Framework for Measuring Program Comprehension

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieurin (Dr.-Ing.)

vorgelegt der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Dipl.-Inform. Dipl.-Psych. Janet Siegmund
geb. am 15. Februar 1982 in Blankenburg

Magdeburg, August 21, 2012

Siegmund, Janet
Framework for Measuring Program Comprehension
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2012.

Abstract

Program comprehension is one of the major human factors in software development. It often makes the difference between success and failure of a software product, because maintenance programmers spend most of their time with understanding code, and because maintenance is the main cost factor in software development. Thus, if program comprehension is not supported properly, time and cost for software development increase significantly.

Although program comprehension has such a paramount importance, researchers and practitioners do not consider it properly, but use plausibility arguments to claim positive or negative effects on program comprehension. However, program comprehension is an internal cognitive process, so we cannot rely on plausibility arguments only, but need to conduct controlled experiments to empirically evaluate what improves and what impairs program comprehension. We believe that one reason for this mismatch between the importance of program comprehension and its insufficient evaluation is the effort of conducting controlled experiments.

Thus, a plethora of new programming paradigms and techniques exist that supposedly improve program comprehension, but have not been evaluated empirically to confirm a positive effect on program comprehension. Hence, although there is effort to improve program comprehension, it is not clear whether this effort has success. To improve the current situation, we define two goals in this thesis: First, reducing the effort for conducting controlled experiments, and, second, creating first empirical evidence how one of the plethora of new approaches, feature-oriented software development, affects program comprehension.

To address our first goal, we first evaluate whether software measures represent a suitable alternative to controlled experiments. Software measures rely only on source-code properties and do not require observing human participants. However, the results indicate that software measure are no reliable substitute for controlled experiments with human participants. Hence, we aim at reducing the effort for controlled experiments with human participants. To this end, we conduct a literature survey of controlled experiments of the last ten years published in seven major journals and conferences of the software-engineering domain. This way, we get an overview of the state of the art of measuring program comprehension, on which we base a framework to support planning, conducting, and replicating experiments. Specifically, we provide a list of confounding parameters for program comprehension, including control techniques, so that researchers know how the validity of experiments is threatened and how they can address the threats. This way, we reduce the major obstacle for conducting controlled experiments with human participants. Furthermore, we develop and evaluate a questionnaire to reliably and conveniently measure programming experience, the most important confounding parameter for program comprehension. Additionally, we provide a tool that helps researchers to design and replicate experiments.

With our second goal, we analyze how a specific programming approach, feature-oriented software development, affects program comprehension. To this end, we conduct a series of controlled experiments, in which we show how different facets of feature-oriented software development affect program comprehension. Specifically, we show

how background colors speed up program comprehension in terms of locating code fragments. Furthermore, we evaluate whether a claimed benefit for program comprehension, which is separation of code along features, really improves program comprehension.

Contents

List of Figures

List of Tables

List of Abbreviations

ESE Empirical Software Engineering

ESEM International Symposium on Empirical Software Engineering and Measurement

fMRI functional Magnetic Resonance Imaging

FSE Symposium on the Foundations of Software Engineering

ICPC International Conference on Program Comprehension

ICSE International Conference on Software Engineering

TOSEM Transactions on Software Engineering and Methodology

TSE Transactions on Software Engineering

Chapter 1

Introduction

Program comprehension is an important human factor in software development. Programmers spend about $50\,\%$ of their time with understanding source code [Standish, 1984; Tiarks, 2011; von Mayrhauser et al., 1997]. Since maintenance is the main cost factor during software development [Boehm, 1981], we can save cost and time during the software-development process by making software comprehendable.

To improve program comprehension, programming techniques came a long way since the development of the first programmable computers around 1945 [Neumann, 1945]. Programming started at a machine-oriented level, in which numbers represented instructions. Then, assembly languages emerged, in which mnemonics refer to instructions [Salomon, 1992]. Over procedural languages, such as Ada [Carlson et al., 1980] and Fortran [Backus et al., 1957], contemporary object-oriented programming was created, which is still a state-of-the-art programming paradigm [Meyer, 1997]. To show the positive effect of object-oriented programming on program comprehension, researchers conducted empirical studies with human participants (e.g., Daly et al. [1995]; Henry et al. [1990]).

To address new requirements that exceed the limits of object-oriented programming, such as variability, extensibility, and customizability, researchers developed novel approaches to implement software. One promising approach is feature-oriented software development, in which *features* are the central elements, instead of objects. A *feature* is "a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option" [Apel and Kästner, 2009]. Features are made explicit to decompose code along them, which supposedly improves program comprehension compared to contemporary object-oriented programming.

So far, there is only little empirical work evaluating the effect of feature-oriented software development on program comprehension, but there are mostly discussions based on plausibility arguments [Apel et al., 2007]. For example, one argument in favor of decomposing code along features is that the amount of information present at the same time is limited, so a developer has to deal with less information. However, as a drawback, when developers need information of other features, they have to trace the information to other files and folders. Thus, limited information can be seen both as benefit and drawback. However, without evaluating how decomposed code affects program comprehension, we do not know whether it is a benefit or a drawback.

To evaluate program comprehension, we need to conduct controlled experiments with human participants, because program comprehension is an internal cognitive process that we cannot observe directly [Koenemann and Robertson, 1991]. Developers understand programs in different ways. If they are familiar with a program's domain, they use top-down comprehension—they derive a general hypothesis about a program and then look for information to confirm or reject this hypothesis [Brooks, 1978; Shaft and Vessey, 1995; Soloway and Ehrlich, 1984]. If developers are not familiar with the domain, they use bottom-up comprehension by analyzing the program statement by statement [Pennington, 1987; Shneiderman and Mayer, 1979]. Thus, to understand the complex process of comprehending source code, we need to observe developers in controlled experiments. Based on the results of such experiments, we can state what effects modern programming techniques, such as feature-oriented software development, have on program comprehension. The more experiments there are, the more reliable and detailed knowledge we can gather.

When we started this project, our initial goal was to evaluate how feature-oriented software development affects program comprehension. However, during our research, we found that there are no common methodologies or tools for conducting experiments. Instead, researchers use their own tools and materials, and only rarely reuse work of other researchers. Thus, there are diverse ways in which researchers conduct controlled experiments, which makes it difficult to compare results of similar experiments, which is necessary to create a common knowledge base.

One reason for this situation is that conducting controlled experiments is time consuming and costly. We need to define an experimental set up, select suitable material and tasks, control for confounding parameters, recruit participants and compensate them for participation, and so on. We believe that this high effort discourages researchers from conducting experiments. We aim at reducing this effort.

Specifically, we define two goals:

- A framework to support controlled program-comprehension experiments.
- A knowledge base that comprises information on the effect of feature-oriented software development on program comprehension.

First, we develop tools and guidelines for conducting experiments that evaluate program comprehension. We refer to them as *framework*, since they are similar to frameworks as used to develop programs, such as Eclipse [Johnson and Foote, 1988]: There are common parts that are required in every comprehension experiment (e.g., presenting material), and user-specific parts that differ in across different experiments (e.g., the kind of material). With our framework, we target the obstacles that prohibit researchers from conducting controlled experiments. Specifically, we give recommendations how to measure program comprehension, assemble and analyze a list of confounding parameters for program comprehension, present a questionnaire to measure programming experience as the most important confounding parameter, and provide a tool to support conducting and replicating comprehension experiments.

Second, we pursue our initial goal to evaluate how feature-oriented software development affects program comprehension. This way, we start creating a knowledge base

about the positive or negative effects of decomposing source code along features. Since feature-oriented software development is a new approach, it is not established in industry yet and is still refined by researchers. Thus, with a sound empirical knowledge base, we can guide the refinement of feature-oriented software development in a direction that improves program comprehension and, in the long run, reduces the time and cost of real-world software development.

For a better overview, we divide this thesis into two parts according to our goals: In the first part, we develop our framework, and in the second part, we present our experiments regarding program comprehension in feature-oriented software development. We pursued both goals in parallel, so work of both parts influenced each other. Thus, the order of the chapters does not reflect the chronological order of our work. Instead, we ordered the chapters according to a coherent story line. Hence, we often use preliminary parts of our framework for our experiments and extended these preliminary parts afterwards.

1.1 Contributions

By addressing our goals, we make the following contributions to the software-engineering community:

1. *Recommendation for measuring program comprehension*.
   Researchers often use software measure to assess program comprehension. In a controlled experiment, we evaluate whether software measures are reliable indicators for program comprehension. Furthermore, we are currently exploring whether we can use functional magnetic resonance imaging to measure program comprehension. Based on the results of both studies, we recommend to conduct controlled experiments to reliably measure program comprehension.

2. *Overview of confounding parameters and control techniques.*
   To reliably measure program comprehension in experiments, we need to control for confounding parameters, which may bias our results. To support researchers in identifying and controlling for confounding parameters, we conducted a literature survey and analyzed how confounding parameters are currently managed. Based on the results of the literature survey and standard control techniques rooted in psychology, we give recommendations how to deal with confounding parameters during designing experiments.

3. *Initial questionnaire to measure programming experience.*
   To control for confounding parameters, we often need to measure them. In our work, we develop a questionnaire to measure programming experience, the major confounding parameter for program comprehension. Our proceeding also serves as recommendation how to create questionnaires to reliably measure certain confounding parameters.

4. *Tool support for comprehension experiments.*
   In comprehension experiments, source code and other material needs to be presented to participants. To support this presentation, we present the tool PROPHET,

which allows experimenters to specify how participants see material, for example, with or without syntax highlighting or with or without a search functionality. PROPHET documents all customizations automatically, so it supports exact replication of experiments. Furthermore, PROPHET logs behavior of participants and can be extended with further functionality.

5. *Empirical evidence of how feature-oriented software development affects program comprehension.*
   With our experiments, we provide first empirical evidence on the benefits and drawbacks of feature-oriented software development on program comprehension. We show that background colors can speed up program comprehension even in large preprocessor-based software. Furthermore, we compare different techniques to separate code along features regarding their effect on program comprehension.

6. *Reusable experimental designs.*
   Furthermore, we designed our experiments to be reusable. To this end, we provide detailed descriptions of our settings and make the tools we used available online at our website (`http://fosd.net/experiments`). Thus, we and others can replicate our experiments, either exact or with slightly modified settings, so that we can extend our knowledge base regarding the effect of feature-oriented software development on program comprehension.

1.2 Outline

In Chapter 2 (*Background*), we briefly introduce general terms and concepts regarding program comprehension, controlled experiments, and feature-oriented software development. This way, we present the most important topics to understand this thesis.

1.2.1 Part I

In Chapter 3 (*Exploring software measures to assess program comprehension*), we describe a controlled experiment to evaluate whether software measures are suitable program-comprehension indicators. We give a detailed explanation of our experimental setting and analysis. The result of this experiment motivates the work presented in the remaining chapters of Part I.

In Chapter 4 (*Confounding parameters for program comprehension*), we present our literature survey to identify confounding parameters for program comprehension. We explain in detail how we selected the papers and extracted relevant information to support other researchers in confirming and extending our list of confounding parameters. Furthermore, we give recommendations on how to manage confounding parameters.

In Chapter 5 (*Measuring programming experience*), we describe how we developed a questionnaire to measure programming experience, the major confounding parameter for program comprehension. We explain in detail the design of the questionnaire based on a literature survey and its evaluation. This way, we enable other researchers to develop reliable questionnaires for other confounding parameters.

In Chapter 6 (PROPHET*: Program-comprehension experiment tool*), we present our tool to support planning, conducting, and replicating comprehension experiments. We describe the requirements we address with our tool based on a literature survey and evaluate whether and how we fulfill these requirements. Furthermore, we describe the architecture of our tool, which we designed such that currently not supported or identified requirements can be addressed.

1.2.2 Part II

In Chapter 7 (*Using background colors to escape the #ifdef hell*), we present our family of three controlled experiments to evaluate whether and how background colors improve program comprehension in preprocessor-based software. With the family of experiments, we show how to start empirical research of an unexplored topic: We start with a small focus and then extend the focus stepwise by replicating experiments with (slightly) modified settings.

In Chapter 8 (*Current projects*), we present the projects we are currently working on. First, we describe the current status of our experiments to evaluate whether we can use functional magnetic resonance imaging to measure program comprehension. Second, we present our experimental setting to compare different techniques to separate code along features. Last, we describe an experiment in which we evaluated whether presenting a software model in a stepwise manner improves its comprehension, compared to presenting a software model all at once.

In Chapter 9 (*Conclusion and future work*), we summarize our contributions and give suggestions for future work.

1.3 Limitation

One important limitation of our research is that we mostly recruited students as participants. Thus, our results are only valid in the context of students. For our results to be valid for professional programmers, we would have to recruit professional programmers. However, they typically get paid for their participation, leading to high costs for conducting experiments. For example, Arisholm and others paid up to $90\,000$ Euros for their experiments with 130 professional Java developers [Arisholm et al., 2002]. Since we do not have these resources, we recruited mostly students, which are typically rewarded by bonus points for their university courses and/or raffles for gift cards.

Of course, recruiting students is discussed controversially, because they do not have the experience and knowledge of professional programmers [Hanenberg, 2010; Höst et al., 2000; Svahnberg et al., 2008; Tichy, 2000]. However, it is often the only choice. Hence, many experiments in the empirical-software-engineering literature are conducted with students. Thus, our results make a useful contribution to the empirical-software-engineering research community. Additionally, since our experimental settings are designed to be reusable, researches who have according resources can replicate our experiments with professionals.

1.4 How to Read this Thesis

In this thesis, we describe numerous experiments in detail to support their replication. However, reading each experiment with all details is not necessary to understand our work. Thus, we provide an overview of each experiment in a table at the beginning of each chapter or section, including pointers to sections that contain more information. Thus, readers who want to get an overview do not need to read the complete experiment descriptions, but can refer to that summary. To present further material to support replication of experiments, we use the project's website (`http://fosd.net`). We decided not to use the appendix, because we often have large Excel sheets with information that cannot reasonably be presented in text documents. Furthermore, we avoid increasing the number of pages of our thesis and can save some trees.

To present the experiments, we use the guidelines developed by Jedlitschka and others [2008]. Thus, we first discuss the objective and hypotheses of an experiment, then the material we used to evaluate our hypothesis, then our sample, and the tasks we used. We describe the conduct of the experiments and whether and what deviations occurred. Then, we analyze our data, starting with descriptive statistics followed by hypothesis testing, and interpret them, after which we discuss threats to validity.

At the beginning of each chapter, we give an outline of its sections to give the reader an overview. Each chapter contains the sections *threats to validity*, *related work*, and *summary*, which we do not mention in chapter overviews.

Furthermore, all experiments have the threat to external validity that we recruited students and generalizability to experts not possible. Since we already discussed this problem, we do not mention it for neither of the experiments anymore.

Last, we give a brief background to understand the main conclusions of our thesis. We present detailed background information specific for a chapter in separate sections, denoted with "Background: [Section Title]". This way, we clearly separate existing work from our own work and allow the reader familiar with this topic to easily skip according sections.

Chapter 2

Background

In this chapter, we present background information regarding program comprehension (Section 2.1), controlled experiments (Section 2.2), and feature-oriented software development (2.3), which is necessary to understand the contributions of this thesis. Since we cover a broad topic, introducing all relevant information would bloat this chapter. Hence, we present only common background information in this chapter, and discuss specific or uncommon techniques we use in a specific chapter in according sections, marked by "Background: [Title of Technique]".

In addition to program comprehension, controlled experiments, and feature-oriented software development, we introduce the software system MobileMedia for the manipulation of multi-media data on mobile devices (Section 2.4), because we use it in most of our experiments.

2.1 Program Comprehension

Program comprehension is an internal cognitive, hypothesis-driven problem solving process that can be defined as

> "the process of understanding a program code unfamiliar to the programmer" [Koenemann and Robertson, 1991].

How this process takes place depends on how much knowledge programmers can use to understand a program. Programmers who have knowledge about a program's domain use their knowledge during the comprehension process. Based on amount of domain knowledge, there are three different kinds of comprehension models: top-down models, bottom-up models, and integrated models. To understand program comprehension, we take a closer look at each of these models.

2.1.1 Top-down Models

If programmers are familiar with a program's domain (e.g., operating systems), they understand programs top down. First, they state a general hypothesis about a program's purpose. To this end, programmers compare the current program with familiar programs

and ideas (e.g., scheduling strategies) of that domain. During that first step, they ignore details and only focus on relevant facets for building the hypothesis.

After stating a general hypothesis, programmers evaluate it by looking at details. They refine their hypothesis stepwisely by defining and refining subsidiary hypothesis, until having a low-level understanding of the source code. During that process, *beacons* (i.e., "sets of features that typically indicate the occurrence of certain structures or operations in the code" [Brooks, 1983], such as identifier names) give hints about the purpose of statements. Furthermore, *program plans* (i.e., "program fragments that represent stereotypic action sequences in programming" Soloway and Ehrlich [1984], such as increments, which indicate loops) can be used to understand groups of statements [Rich, 1987]. Based on domain knowledge, and using beacons and program plans, programmers verify, modify, or reject the general hypothesis.

Examples of top-down models are described by Brooks [1978], Shaft and Vessey [1995], as well as Soloway and Ehrlich [1984].

2.1.2 Bottom-up Models

Without domain knowledge, programmers cannot compare the current program with other programs or search for beacons, because they do not know what they look like. Hence, programmers need to examine source code closely to be able to state hypotheses of a program's purpose. In this case, they start to understand a program by examining details of a program—the statements or control constructs that comprise the program. Statements that semantically belong together are grouped into higher level abstractions, called *chunks*. If enough chunks are created, programmers leave the statement level and integrate those chunks to further higher level abstractions.

For example, if programmers recognize that a group of statements have a high level purpose, they create one chunk and then refer to that chunk (e.g., as "sorting elements in a list"), not the single statements. When further examining the program, programmers combine these chunks into larger chunks (e.g., "implementing scheduling strategies"), until they have a high-level understanding of a program.

Examples of bottom-up models differ on the kind of information that is combined to chunks. For example, Pennington [1987] states that control constructs (e.g., sequences or iterations) are used as base for chunking, whereas Shneiderman and Mayer [1979] describe that chunking begins on the statements of a program.

In practice, programmers do not understand source code solely top down or bottom up, but use both approaches depending on how much knowledge they can use to understand a portion of a program. This is described by integrated models, which we discuss next.

2.1.3 Integrated Models

Integrated models combine top-down and bottom-up comprehension. For example, if programmers have domain knowledge about a program, they form a hypothesis about its purpose. During the comprehension process, they encounter several fragments that they cannot explain using their domain knowledge. Hence, they start to examine the

program statement by statement, and integrate the newly acquired knowledge in the hypotheses about the source code. Usually, programmers use top-down comprehension where possible and bottom-up comprehension only where necessary, because top-down comprehension is more efficient [Shaft and Vessey, 1995].

One example of integrated models is described by von Mayrhauser and Vans [1993]. They divide program 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.

2.1.4 Discussion of Comprehension Models

All program-comprehension models we described were developed at least 20 years ago. To the best of our knowledge, there are no contemporary models explaining program comprehension, but only extensions. For example, Rajlich and Wilde describe how *concepts* play a role in the comprehension process [Rajlich and Wilde, 2002]. Concepts are "units of human knowledge", such as classes or high-level design patterns and are identified by programmers during their comprehension process and supporting top-down comprehension.

We believe that contemporary programming approaches better reflect the human way of thinking, for example, the classification into objects. Furthermore, abstractions, such as inheritance hierarchies, help developers to structure information and increase program comprehension. For example, Daly and others showed that programs using inheritance are better maintainable then programs without inheritance [Daly et al., 1995]. For future work, it would be interesting to evaluate what facets of modern programming paradigms influence program comprehension in what way.

Furthermore, all models assume that developers are familiar with the underlying programming language. Otherwise, an additional learning process takes place, in which developers have to familiarize with the new language. For our experiments, we recruited participants who are familiar with the used programming language.

Being aware of the different kinds of comprehension models is necessary to soundly and reliably measure program comprehension. If we do not consider these differences, results of experiment may be biased and describe something different. For example, if we compare two different programming techniques between two groups of participants, and observe one group of participants with domain knowledge using one technique, and another group of participants without domain knowledge with another technique, we would not only compare the programming techniques, but also top-down with bottom-up comprehension. However, this would bias our results, because top-down comprehension is more efficient, and we cannot be sure what exactly we measured. Thus, taking into account different comprehension models is one important facet in creating sound and reliable experimental designs that measure program comprehension.

2.1.5 Measuring Program Comprehension

So, program comprehension is a complex internal cognitive process—but how can we measure it? In literature, we found four different approaches:

- Think-aloud protocols
- Tasks
- Subjective rating
- Software measures

Think-Aloud Protocols Think-aloud protocols are rooted in cognitive psychology and require participants to verbalize their thoughts [Wundt, 1874]. This way, we can observe the comprehension process. To enable a detailed analysis of participants' thoughts, think-aloud sessions are usually recorded on audio- and/or videotape (or manually by the experimenter).

After recording think-aloud protocols, they have to be analyzed. To ensure objective analysis, experimenters often create coding instructions that describe how to analyze the data [Lang and von Mayrhauser, 1997; Shaft and Vessey, 1995]. For example, Shaft and Vessey defined what statements of participants are hypotheses (e.g., "probably scheduling strategy"), and what statements are inferences (e.g., "must have something to do with scheduling") [Shaft and Vessey, 1995]. Since hypotheses are typical indicators of top-down comprehension, and inferences of bottom-up comprehension, the authors could analyze what comprehension process developers apply. To assure that coding instructions are followed, several researchers should analyze the recorded protocols independently and then compare their answers and discuss disagreements until reaching interpersonal consensus [Someren et al., 1994, p. 45]. To further increase objectivity, researchers who categorize the protocols should not be aware of the hypotheses (e.g., that programmers use top-down comprehension more often) to not intentionally or unintentionally bias the categorization (e.g., categorizing statements as hypothesis, although they are inferences).

Applied correctly, think-aloud protocols provide a sound insight into the comprehension process. However, they are time consuming and costly [Someren et al., 1994]: Sessions of participants need to be recorded and participants' statements need to be categorized carefully according to rules and/or by different persons. Furthermore, only one participant at a time can be evaluated, because if more than one participant talks during comprehending code in the same room, other participants are disturbed. Hence, think-aloud protocols can only be applied if sufficient resources are available.

Tasks During tasks, participants are asked to work with source code, for example, fixing a bug, enhancing the code, or locating certain code fragments. If participants successfully solve a task, they must have understood the code. To analyze comprehension, we can use correctness or response time of a solution [Dunsmore and Roper, 2000]. When using tasks, we can observe a large number of participants at the same time. However,

we do not get information about the comprehension process itself, but only how fast or correct the comprehension process was. Such results are useful, for example, when we analyze whether a certain technique speeds up comprehension or improves correctness.

Subjective Rating When using subjective rating, participants estimate how much they understood of the code, for example, on a five-point scale ranging from "nothing at all" over "about half of the source code" to "the complete source code". However, this can easily be biased, because participants may over- or underestimate how much they understood [Dunsmore and Roper, 2000]. Furthermore, we do not have information about the comprehension process itself.

Software Measures Software measures describe properties of source code. For example, the number of lines of code represents a size measure [Henderson-Sellers, 1995] or cyclomatic complexity describes the number possible execution paths [McCabe, 1976]. The more lines a program has or the more complex it is, the more difficult it is supposed to be understood. Software measures can be computed automatically and are easy to apply, because they do not consider the developer who comprehends source code. However, that is also the problem with software measures, so we cannot be sure how well the comprehension process is assessed. To learn more about whether software measures are suitable to assess program comprehension, we conducted a controlled experiment in Chapter 3.

Think-aloud protocols, tasks, subjective rating, and software measures are the four most commonly used techniques to measure program comprehension. However, we found that in cognitive neuroscience, researchers use functional magnetic resonance imaging to observe cognitive processes since 1991 [Belliveau et al., 1991]. To evaluate the feasibility of functional magnetic resonance imaging as technique to measure program comprehension, we are currently planning an experiment, which we describe in Chapter 8.

Having described program comprehension and its measurement, we discuss how to plan, conduct, and analyze controlled experiments, which are often used to apply any of the techniques measuring program comprehension.

2.2 Experiments

Since this thesis describes controlled experiments and requires knowledge about how to design, conduct, analyze, and interpret experiments, we give an introduction to experiments in this section.

An experiment can be described as a systematic research study, in which experimenters directly and intentionally vary one or more *independent variables* while holding everything else constant and observe the results of the systematic variation (Woodworth [1939]; Wundt [1914]).

The process of conducting experiments can be divided into five stages [Juristo and Moreno, 2001, p. 49]:

1. Objective definition
2. Design
3. Experiment Execution
4. Analysis
5. Interpretation

For better overview, we visualize this process in Figure 2.1. First, the variables and hypotheses of the experiment need to be specified. Second, a detailed plan for conducting the experiment needs to be developed. Third, the experiment needs to be executed according to the plan. Fourth, the data collected during the execution need to be analyzed. Finally, the results need to be interpreted and their meaning for the hypotheses evaluated. In this section, we discuss each step in detail.

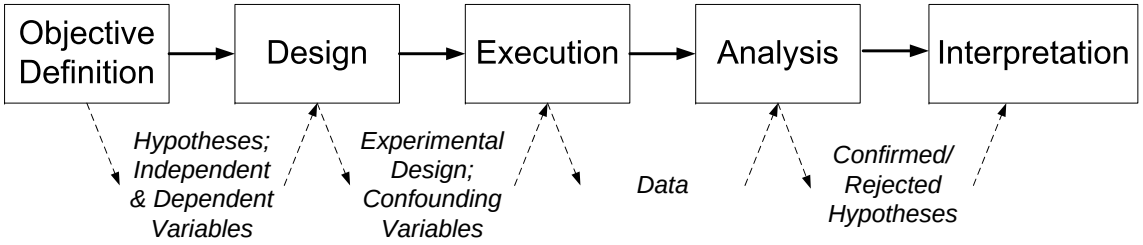


Figure 2.1: *Stages of an experiment.*

2.2.1 Objective Definition

During objective definition, we define variables of interest and hypotheses about the relationship between our variables [Juristo and Moreno, 2001, p. 49]. This enables us to choose a suitable experimental design and analysis methods. If we defined our hypotheses after conducting the experiment, we might find that our material, tasks, and/or participants were not suitable to test our hypotheses, leaving us with useless data. Defining variables and hypotheses is referred to as *operationalization*, because a set of operations is defined with which we measure our variables and test our hypotheses [Bridgman, 1927].

There are three kinds of variables: independent, dependent, and confounding variables. First, *independent variables* are intentionally varied by experimenters and influence the outcome of an experiment [Juristo and Moreno, 2001, p. 60]. They are also referred to as *predictor (variables)* or *factors*. Each independent variable has at least two *levels*, *alternatives*, or *treatments*. For example, when comparing the effect of Java and C++ on program comprehension, the independent variable is programming language with the two levels Java and C++.

Second, *dependent variables* are the outcome of an experiment [Juristo and Moreno, 2001, p. 59] and depend on variations of the independent variable. They are also referred to as *response variable*. In our work, program comprehension is the dependent variable,

because we observe program comprehension, for example, how it is affected by feature-oriented software development.

Third, *confounding variables*, *confounding parameters*, or *extraneous variables* influence the dependent variable besides variations of the independent variable [Goodwin, 1999, p. 143]—they *bias* the dependent variable. To clearly separate independent and dependent variables from confounding variables, we use the term *confounding parameters* from here on. Examples of confounding parameters for program comprehension are programming experience or intelligence. To avoid bias, we need to identify and control for confounding parameters, which we discuss in detail in Chapter 4.

Having defined the variables, we have to specify the *research hypotheses* or *research questions*. Typically, a hypothesis claims benefits or drawbacks (e.g., Java improves program comprehension compared to C++), where as a question does not (e.g., Does Java improve program comprehension compared to C++?). Typically, we state research questions when we have no knowledge about how levels of the independent variable affect the dependent variable or when we can plausibly argue in both directions.

It is important to specify research hypotheses and questions during the objective definition, because they influence the decisions in the remaining stages of the experiment (e.g., participants and analysis methods) [Bortz, 2004, p. 2]. Furthermore, since we define beforehand what questions we address, we avoid "fishing for results" in our data, which may lead to discovering random relationships between variables [Easterbrook et al., 2008].

2.2.2 Design

The next step is to design a plan for conducting the experiment. In this stage, we have to ensure that our experiment is internally and externally valid.

Validity *Internal validity* describes the degree to which the value of the dependent variable can be assigned to the variation of the independent variable [Shadish et al., 2002]. To assure that the result can be attributed to the independent variable, we need to control for confounding parameters. Otherwise, we cannot be sure that our result can solely be attributed to our variation of the independent variable.

External validity describes the degree to which the results gained in one experiment can be generalized to other participants and settings [Shadish et al., 2002]. The more realistic an experimental setting is, the higher its external validity is.

Both kinds of validity are conflicting: Maximizing internal validity means controlling everything, leading to an artificial setting (e.g., only bottom-up comprehension of students of a programming course) and low external validity. Maximizing external validity means a realistic setting (e.g., all comprehension models with students and professional programmers), but now internal validity is minimized. Thus, we need to find a tradeoff between internal and external validity. To this end, we can use a two-staged approach [Shadish et al., 2002]. First, we maximize internal validity in a series of experiments, so that we create a sound knowledge base about what influences program comprehension. Second, we increase external validity in subsequent experiments to test and improve the knowledge base under realistic settings.

Language	Group
Java	1
C++	2

(a) Two groups, between-subjects

Session 1	Session 2
Java	C++

(b) One group, within-subjects

Group	Session 1	Session 2
1	Java	C++
2	C++	Java

(c) Two groups, within-subjects

Table 2.1: *Examples of one-factorial designs.*

Experimental Designs Next, we have to choose an appropriate experimental design, which defines how we apply levels of the independent variable(s) to participants [Juristo and Moreno, 2001]. The most common designs are one-factorial (one independent variable) and two-factorial (two independent variables) designs. There are also three- or more-factorial designs, but they are time consuming and costly, because they require a large sample and specific analysis methods. Thus, we focus on one- and two-factorial designs.

In *one-factorial designs*, we have one independent variable with two or more levels. We can either use a *between-subjects* or *within-subjects* design [Goodwin, 1999, p. 205]. Between-subjects means that we split our sample in two or more groups and then compare these groups. Hence, participants experience only one treatment. Within-subjects means that we apply all treatment to all participants. For illustration, we show a few one-factorial designs in Table 2.1. As example, we use programming language with the levels Java and C++ as independent variable. All designs have benefits and drawbacks. The design in Table 2.1a is simple, but we have to ensure that both groups are comparable, for example, have the same level of programming experience. Furthermore, our sample should be large enough to be split into two groups. Unfortunately, large cannot be defined as a fixed number; in discussions with other researchers, we found that ten participants per groups seem to be acceptable, but five per group not enough. The design in Table 2.1b uses only one group, so it does not require comparable groups, and the sample can be smaller. However, participants might learn from the first session and behave differently in the second session. Furthermore, the order of sessions might influence the behavior of participants. To control for such learning and ordering effects, we can use the design in Table 2.1c, which is more complex and requires a large enough sample size.

In a *two-factorial design*, we have two independent variables. We show an example with programming experience (two levels) as second independent variable in Table 2.2a. Now, we have four groups based on the combination of the levels of the independent variables. Here, we have the same problems as for the one-factorial between-subjects design in Table 2.1a. To address these problems, we can use a within-subjects design, in which each group experiences all four combinations of levels, as shown in Table 2.2b.

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.

Independent Variables	Group
Java/novice	1
Java/expert	2
C++/novice	3
C++/expert	4

(a) Four groups, between-subjects

Independent Variables	Session 1	Session 2	Session 3	Session 4
Java/novice	A	B	C	D
Java/expert	B	C	D	A
C++/novice	C	D	A	B
C++/expert	D	A	B	C

(a) Four groups, between-subjects

Table 2.2: *Examples of two-factorial designs.*

2.2.3 Experiment Execution

In this stage, the experiment is conducted according to the developed plan. Despite all careful planning, deviations can occur, for example, participants who arrive late, missing questionnaires, or program crashes. These deviations should be recorded and described when writing a report, because they can threaten validity of experiments and to avoid that other researchers who replicate an experiment have the same deviations.

2.2.4 Data Analysis

Having conducted the experiment, we need to analyze the data. To apply the correct analysis method, we need to know the scale of our data. Then, we can compute descriptive statistics to describe the data and conduct significance tests to evaluate our hypotheses.

Scale Types

There are four common scale types: nominal, ordinal, interval, and ratio [Fenton et al., 1994]. *Nominal* scales are classification of values, such as male or female. We can assign numbers to the values (e.g., 0 and 1), but they have no quantitative meaning. *Ordinal* scales describe a ranking, for example, the order of participants according to their response time. The ranks indicate only the order of participants. On *interval* scales, numbers have a quantitative meaning, and 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 also referred to as *metric*. From hereon, we do not differentiate between interval and ratio scale, because the analysis methods we need for our experiments do not differentiate between them.

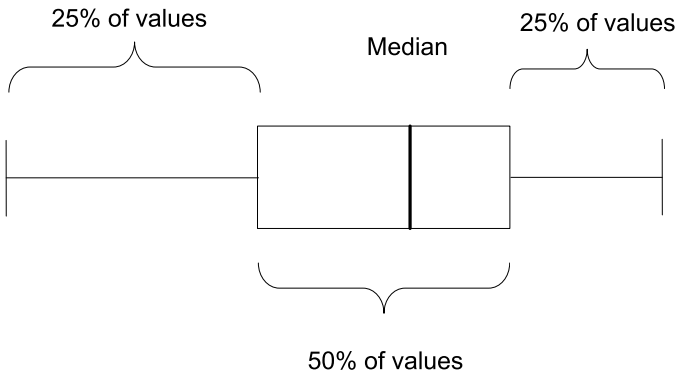


Figure 2.2: *Illustration of a box plot.*

Scale	Frequencies	Median	Mean	Standard deviation	Box plot
Nominal	Yes	No	No	No	No
Ordinal	Yes	Yes	No	No	Yes
Metric	Yes	Yes	Yes	Yes	Yes

Table 2.3: *Scale types and allowed measures.*

Descriptive Statistics

Depending on the scale type, we can compute descriptive statistics [Bortz, 2004, p. 35]. For nominal data, we can compute only *frequencies*, for example, the number of males and females in our sample. For ordinal data, we can compute the *median*, which is the value in the middle of an ordered list of values. 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. If we have metric data, we can compute the *arithmetic mean* (or *mean*, from here on) and *standard deviation*.

Besides computing numbers, we can visualize the distribution of data with a box plot, as shown in Figure 2.2 [Tukey, 1977]. The box contains $50\,\%$ of all values, the thick line shows the median. The whiskers contain the upper and lower $25\,\%$ of the values. We can also draw *outliers* as separate dots (i.e., values that deviate strongly from the median, e.g., more than 2 standard deviations). For better overview, we summarize what measure can be applied to what scale in Table 2.3.

Hypothesis Testing

Having described the data, we can evaluate our hypotheses. To this end, we conduct *significance tests*. For example, if we observe a difference in program comprehension in favor of Java, significance tests evaluate whether this difference is real or occurred randomly. To this end, they compute the *conditional probability* of having observed the result under the assumption that there is no difference between program comprehension of Java and C++. The smaller the conditional probability, the more unlikely the assumption of no

difference is. If the conditional probability is below $5\,\%$, we reject the assumption that there is no difference between Java and C++. The assumption is also referred to as *null hypothesis* (i.e., there is no difference), and the conditional probability as *significance level* or *p value*.

Depending on the scale, we have to select an appropriate test. There are a plethora of significance tests (see, e.g., Anderson and Finn [1996]; Bortz [2004]). We explain only the ones we use in this thesis, and give an overview of when to apply which test in Figure 2.3 at the end of this section.

Nominal Scale—chi^2 Test When comparing frequencies, we typically conduct a chi^2 test [Anderson and Finn, 1996]. It evaluates whether the observed frequencies deviate from expected frequencies. For example, if we have a sample of 7 males and 3 females, and we expect that gender is equally distributed (i.e., 5 males and 5 females, which is the null hypothesis), the chi^2 test evaluates whether the observed frequencies deviate significantly from the expected frequencies. To apply the chi^2 test, expected frequencies must be larger than 5. Otherwise, we have to apply a Fisher's exact test to compensate for small expected frequencies [Fisher, 1922].

Ordinal Scale—Mann-Whitney-U Test When comparing the order of data, we conduct a Mann-Whitney-U test [Anderson and Finn, 1996]. For example, if we order the fictional response times of a task measuring program comprehension with Java and C++ according to their values, the Mann-Whitney-U test analyzes the order of the values (i.e., the ranks, cf. Table 2.4). The null hypothesis is that the sum of ranks is the same for Java and C++. If the sums of ranks differ significantly, we reject the null hypothesis.

Java		C++	
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:	$T_1 = 117$		$T_2 = 93$
Mean:	105.1		98.5

Table 2.4: *Example response times to illustrate the Mann-Whitney-U test.*

Metric Scale—T Test For metric scales, we use the t test [Anderson and Finn, 1996]. For example, we can compare whether the means of response times of Table 2.4 differ

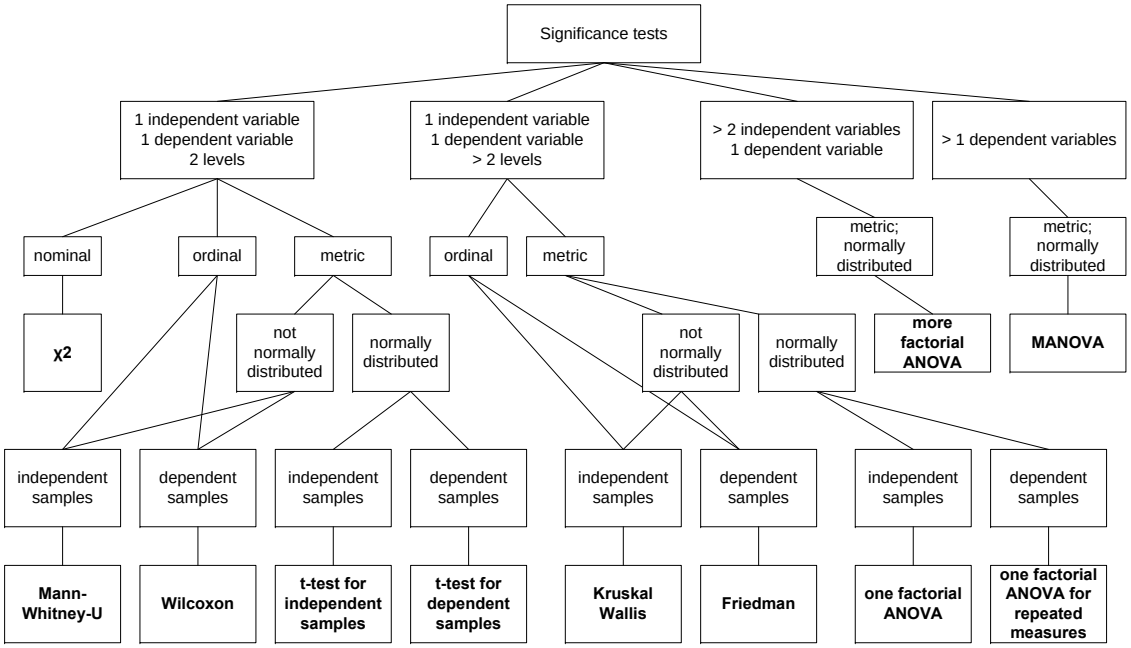


Figure 2.3: *Overview of significance tests.*

significantly. The null hypothesis is that the response times do not differ. The t test requires that our data are normally distributed, for which we can use a Shapiro-Wilk test [Shapiro and Wilk, 1965]. If they are not normally distributed, we have to apply a Mann-Whitney-U test instead. However, if our sample is large enough (at least 30 participants), the t test is robust against not normally distributed data.

These are the most common significance tests. In Figure 2.3, we give an overview of further significance tests and when to apply which test (for more information, see standard statistic books, e.g., Bortz [2004] or Anderson and Finn [1996]). There are also *exploratory methods*, which analyze whether there are certain relationships or patterns in a large amount of data. In Chapter 5, we use such methods and introduce them there.

2.2.5 Interpretation

Computing descriptive statistics and evaluating hypotheses are instruments to analyze the data. Once we accepted or rejected our hypotheses, we have to relate our results to the objective of our experiment. Furthermore, if we obtained unexpected results, we have to search for possible explanations, for example, by exploring our data or consulting similar experiences found in literature.

Reporting data should be strictly separated from their interpretation [American Psychological Association 2009]. This way, results can be presented objectively, which gives the reader the chance to understand our interpretations and conclusions we draw from our results.

2.2.6 Ethical Issues

In experiments with human participants, the outcome of experiments has to be in relation to what participants have to endure during the experiment Wiesing [2003]. In software engineering, this is often not problematic, because participants typically have to implement code or work with new tools. However, when evaluating the efficiency of new teaching methods with students, a between-subjects design is questionable, because either only one group experiences the efficiency of the new teaching method or fails to learn important topics because the new teaching method is unsuitable.

Furthermore, participating in an experiment should not lead to a disadvantage of participants. This is especially important for students, who are often recruited from a university course. Their participation should not affect the completion or grade of the course. To this end, the data of participants are often anonymized. This way, the performance in the experiment cannot be mapped to the students.

2.3 Feature-Oriented Software Development

Feature-oriented software development describes the design and implementation of applications based on features [Apel et al., 2008b; Kästner et al., 2009b]. A feature is a user-visible characteristic of a software system [Clements and Northrop, 2001, p. 114]. Feature-oriented software development provides formalisms, methods, languages, and tools for building variable, customizable, and extendable software [Apel and Kästner, 2009]. One goal of feature-oriented software development is to separate *crosscutting concerns*. A concern is anything a stakeholder might be interested in [Robillard and Murphy, 2007]. It can be the same as a feature, but does not have to be. In our case, the difference between concern and feature is irrelevant, so we use both terms interchangeably.

To illustrate crosscutting concerns, consider a database implementation with the features *Logging* and *Transaction management*. *Logging* is responsible for logging each access to the database, and *Transaction management* is responsible for assuring that if several users access the database at the same time, no inconsistencies occur. Now, every statement that accesses the database belongs to feature *Logging*. However, *Transaction management* also accesses the database, so we need code for both features in the same module. Now, we can either separate code according to feature *Logging*, or according to feature *Transaction management*, but not both at the same time. Feature *Logging* is either *scattered* over the implementation of *Transaction management*, or it is *tangled* with code of *Transaction management*. With two features, scattering and tangling do not appear to be a problem, but real databases consist of considerably more features (e.g., SQLite has 85 features[1]).

Feature-oriented software development summarized techniques to overcome this dilemma and modularize crosscutting concerns. By separating code along features, we create a *software product line*, which allows us to create different *variants* of a product. As running example, we use a simple stack as shown in Figure 2.4. It consists of the features *Base*, *Safe*, *Top*, and *Element*. *Base* is the base implementation and is part of every variant. *Safe* ensures that no elements can be popped from an empty stack. *Top* adds the

[1] `http://sqlite.org/`

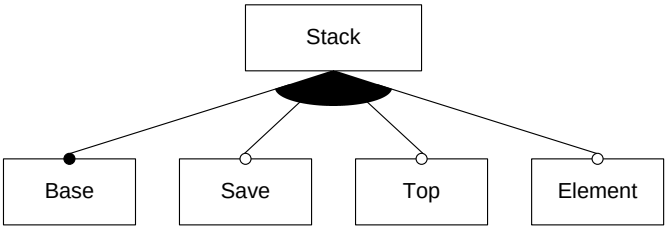


Figure 2.4: *Feature diagram of a stack.*

method `top`, which returns the first element from a stack without removing it. *Element* ensures that only elements with a specified type can be stored in the stack. Now, by combining features, we can create different variants of the Stack, for example, a stack with the features *Base*, *Safe*, and *Element*, or with the features *Base* and *Top*, or with all four features. Thus, we have a customizable implementation, from which we can create different variants without implementing any code.

Typically, feature-oriented programming techniques separate concerns either *physically*, that is, in different modules (e.g., files or folders), or *virtually*, such that source code belonging to a feature (i.e., *feature code*) is annotated accordingly. We discuss both approaches in this section.

2.3.1 Physical Separation of Concerns

There are different ways to separate concerns physically. We focus on the most commonly used techniques: feature-oriented programming [Prehofer, 1997] and aspect-oriented programming [Kiczales et al., 1997]. First, in feature-oriented programming, features are encapsulated in units, called *feature modules*. For illustration, we show a collaboration diagram of the stack including its implementation. Class `Stack` is divided into three *roles*, such that each role contains the implementation of one feature. For each feature, a folder exists, and for each role, the according class is stored in one file in the according folder. For example, folder Top contains one file with class `Stack` that contains only code of the according role. Now, if we select the features *Base* and *Top*, the according roles are combined, for example, by creating inheritance chains [Batory et al., 2004] or by superimposition [Apel and Lengauer, 2008].

Second, in aspect-oriented programming, features are encapsulated in *aspects*, which are stored in separated files and/or folders. An *aspect* contains source code that allows us to alter the behavior of a program during runtime. For illustration, we show how features *Top* and *Safe* can be implemented in Figure 2.6. The implementation of feature *Base* remains the same. When we run the code of feature *Base*, and the method `pop` is executed, the *pointcut* defined in Line 7 captures this event, and the *advice* defined from Line 8 to Line 11 is executed instead, ensuring that no elements can be popped from an empty stack.

Besides feature-oriented and aspect-oriented programming, there are further approaches that separate code physically, for example, Hyper/J [Tarr and Ossher, 2001], CaesarJ [Aracic et al., 2006], and aspectual feature modules [Apel et al., 2008a].

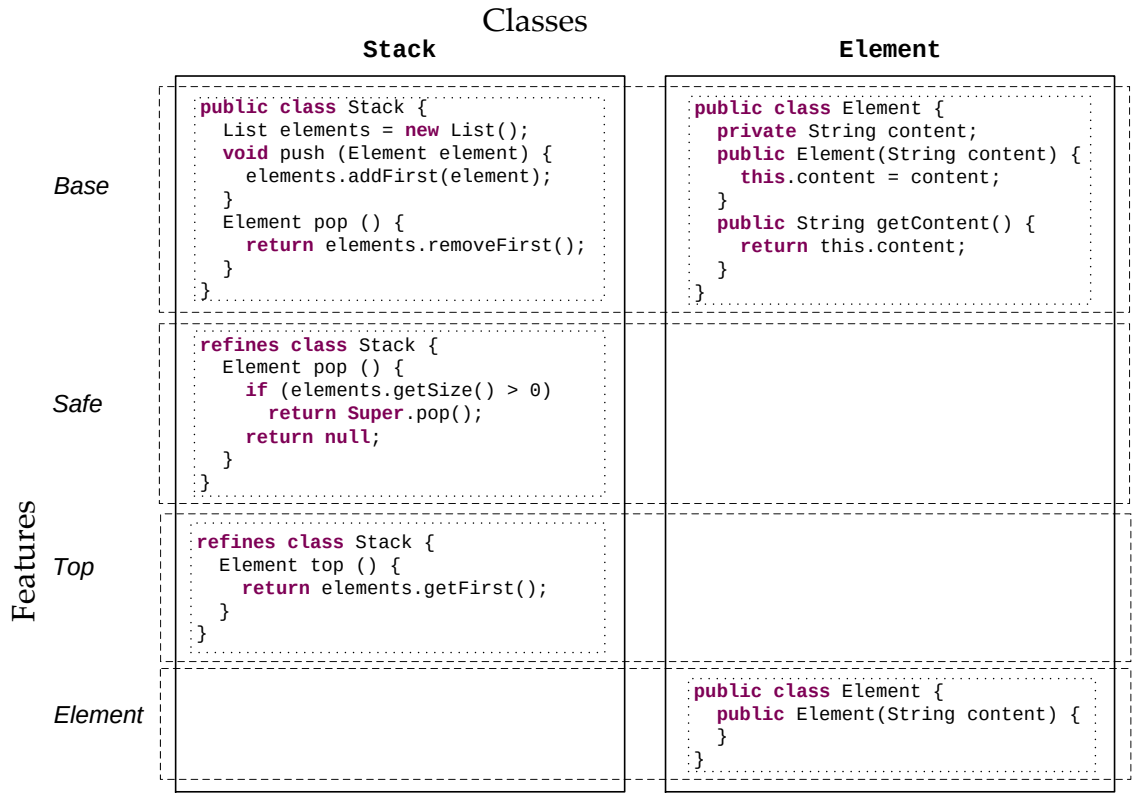


Figure 2.5: *Collaboration diagram of a stack.*

```
1  public aspect Top {
2     public Element Stack.top() {
3        return elements.getFirst();
4     }
5  }
6
7  public aspect Safe {
8     pointcut safePop(Stack stack): execution(Element pop()) && this(stack);
9
10    Element around(Stack stack): safePop(stack) {
11       if (stack.items.size() > 0) return proceed(stack);
12       return null;
13    }
14 }
```

Figure 2.6: *Aspect-oriented implementation of* Safe *and* Top *of the stack example.*

```
1  public class Stack {
2     LinkedList<Element> elements = new LinkedList<Element>();
3
4     public void push(Element element) {
5        elements.addFirst(element);
6     }
7
8     public Element pop() {
9        //#ifdef SAFE
10       if (elements.size == 0) return null;
11       //#endif
12       return elements.removeFirst();
13    }
14
15    //#ifdef TOP
16    public Element top() {
17       //#ifdef SAFE
18          if (elements.size == 0) return null;
19       //#endif
20       return elements.getFirst();
21    }
22    //#endif
23 }
```

Figure 2.7: *Preprocessor-based implementation of the stack example (Antenna).*

2.3.2 Virtual Separation of Concerns

With virtual separation of concerns, feature code is annotated, for example, with preprocessor directives or colors. Based on annotations, views on the source code can be created, thus emulating modules [Kästner et al., 2008]. The most common technique to virtually separate concerns is *preprocessor directives* or *ifdef directives*, which we illustrate with the stack example in Figure 2.7. Code that belongs to feature *Safe* is enclosed in an ifdef (followed the name of the feature) and according endif statement (Line 7 and 9). To create a variant, a preprocessor deletes code of not selected features before compiling. For example, if we do not select feature *Top*, everything from Line 12 to 19 is deleted before compilation.

Similar or in addition to ifdef directives, we can use background colors to annotate feature code. For example, the tool CIDE allows users to assign colors to features [Kästner et al., 2008]. Other approaches are XVCL [Jarzabek et al., 2003], FEAT [Robillard and Murphy, 2003], pure::variants [Beuche et al., 2004], and GEARS [Krueger, 2002].

The advantages and disadvantages of physical and virtual separation of concerns lie—among others—in the grain of feature code and the information that is presented at the same time. First, physical separation of concerns supports a coarse-grained implementation of features, whereas virtual separation of concerns, especially preprocessor directives, supports fine-grained annotations. For example, a single opening bracket can be annotated with an ifdef directive without the corresponding closing bracket. The coarse granularity limits flexibility, but the fine granularity might lead to compilation problems (e.g., when a feature with the opening bracket is deleted, but the closing bracket is still present).

Second, in physical separations, only limited information is presented at once. Since developers are only interested in few features at the same time, the limitation might be beneficial for their comprehension, because they only have to deal with a limited amount of information. However, relevant information might also be missing, in which case developers have to look into different folders, which might impair program comprehension. We discuss this issue in more detail in Section 8.2, when we describe an experiment to evaluate how physical and virtual separation of concerns affect program comprehension.

2.4 MobileMedia

In our experiments, we often use MobileMedia as example of a feature-oriented software system. Thus, we discuss it in this chapter, instead of explaining it for each experiment. MobileMedia is a medium-sized software product line for the manipulation of multimedia data on mobile devices and was implemented by Figueiredo and others with the support of post-graduate students [Figueiredo et al., 2008a]. MobileMedia was developed in parallel in two versions: one version implemented in Java ME with the preprocessor Antenna[2], and one version implemented in AspectJ, an aspect-oriented extension to Java [Kiczales et al., 1997].

MobileMedia was developed in eight releases, of which we use different ones, depending on the purpose of our experiment. In Figure 2.8, we give an overview of the features of the last release. For our experiments, we use the following features:

Sorting: Sorts media according to how often it was viewed

Favourites: Allows users to define and view favorite photos, music, or videos

Video: Plays and captures videos

Music: Plays and captures music

CopyMedia: Copies multi-media data in different albums

SMSTransfer: Sends multi-media data via SMS

AlbumManagement: Allows users to store multi-media data in albums

Note that, since we use different releases of MobileMedia, some features in the experiment descriptions have different names and reduced functionality. For example, in our experiment on background colors, we use the fifth release with a predecessor of feature *CopyMedia*, which is called *CopyPhoto* and only copies photos, but neither music nor video.

There are several benefits in using MobileMedia for our experiments: First, both versions of MobileMedia were code reviewed, such that the same coding conventions have been applied to both versions. Moreover, exhaustive tests were conducted to assure that both versions are comparable. Because of the efforts of the developers (see Figueiredo et al. [2008a] for more details), two comparable versions exist.

[2]http://antenna.sourceforge.net/

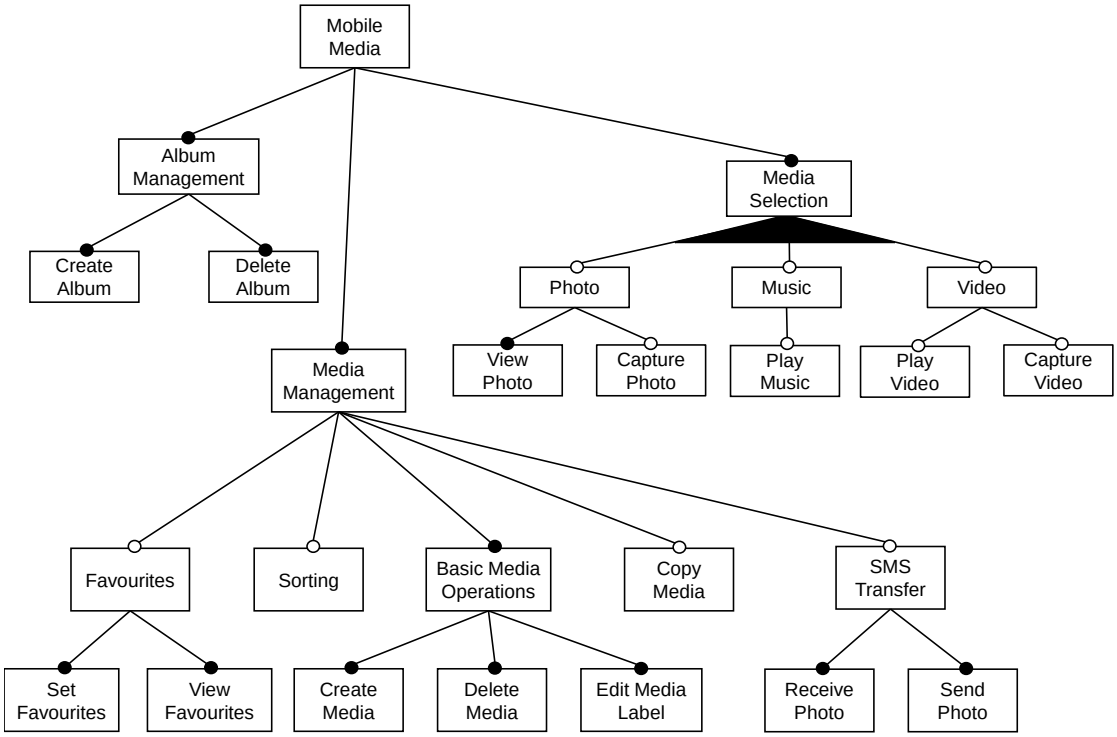


Figure 2.8: *Features of Mobile Media.*

Second, numerous researchers used MobileMedia in their studies [Bertoncello et al., 2008; Bryant et al., 2006; Dyer et al., 2010; Feigenspan et al., 2009; Figueiredo et al., 2008a; Galvâo et al., 2010; Garcia et al., 2005; Greenwood et al., 2007; Kulesza et al., 2006; Molesini et al., 2008; Morin et al., 2009]. Hence, there are a lot of results by other researches, which allow us to relate our work to them and vice versa. Consequently, we contribute to the knowledge base regarding MobileMedia. Furthermore, MobileMedia is a good starting point for generalizing results to other software systems, because numerous different facets have been evaluated thoroughly.

Last, MobileMedia was developed as a software product line. Users can generate different variants of MobileMedia by selecting desired features (e.g., one variant with features *CountViews* and *Favourites*, another variant without both features). To this end, the implementation of a software product line must ensure that there is a mapping of features to the corresponding code units. Thus, we can evaluate program comprehension on the level of features, not only the complete system, allowing us to draw more thorough, low level conclusions (e.g., regarding physically and virtually separated code).

2.5 Summary

In this chapter, we introduced program comprehension, which is an internal cognitive, hypothesis-driven problem solving process. Developers understand code top down or bottom up, depending on their domain knowledge. Thus, program comprehension is a

complex process. To measure program comprehension, researchers typically use think-aloud protocols, tasks, subjective rating, and software measures.

Then, we presented the basics of planning and conducting experiments as well as analyzing and interpreting data. We explained different types of variables (independent, dependent, and confounding) as well as how to assure internal and external validity. To analyze the data, we introduced standard descriptive measures and significance tests to evaluate whether a result occurred randomly or indicates a real difference.

Additionally, we presented feature-oriented software development, a new programming paradigm to modularize crosscutting concerns and implement software product lines. We described physical and virtual separation of concerns and discussed benefits and drawbacks of each approach.

Last, we introduced MobileMedia, a software product line for the manipulation of multi-media data on mobile devices, which we often use in our experiments.

In the next chapter, we evaluate whether we can use software measures to assess program comprehension.

Part I

Framework for Conducting Program-Comprehension Experiments

Chapter 3

Exploring Software Measures to Assess Program Comprehension

This chapter shares content with the ESEM'11 paper "Exploring Software Measures to Assess Program Comprehension" [Feigenspan et al., 2011a], the WSR'11 paper "On the Role of Program Comprehension in Embedded Systems' [Feigenspan et al., 2011c], and the IWDE'12 paper "Program Comprehension in Preprocessor-Based Software" [Siegmund et al., 2012c].

In the previous chapter, we described how program comprehension can be measured. Among others, we presented software measures based on properties of source code. Software measures are easy to apply, because they do not require controlled experiments with human participants. However, their reliability is questionable, because they do not consider human factors, but only source-code properties. Thus, it is unclear whether we can use software measures to assess program comprehension. Thus, we defined the following goal for this chapter:

- Recommendation about the use of software measures to assess program comprehension based on empirical evidence.

If we can find empirical evidence that software measures are related to program comprehension, we could recommend using them to evaluate how new programming techniques, such as feature-oriented software development, affect program comprehension. If we cannot find such a relationship, we show that software measure should not be used as sole indicators for program comprehension.

Since in our work, we are interested in feature-oriented software development as new programming paradigm, we focus on software measures for feature-oriented software systems. Nevertheless, our experimental setup can be reused for other software measures.

In Section 3.1, we take a closer look at software measures and how they might relate to program comprehension to understand why software measures are so popular. In Section 3.2, we describe our experimental setting in detail. This way, we enable researchers to reuse our set up for different software measures and software systems. In Section 3.3,

Reference	Goal	Software Measures
Bryant et al. 2006	Modularization of Pattern Interactions, Composability	CBC, CDC, CDLOC, CDO, DIT, LCOO, LOC, NOA, WOC, Interaction Analysis
Figueiredo et al. 2008a	Modularization, Changeability, Dependencies	CBC, CDC, CDLOC, CDO, DIT, LCOO, VS, LOC, NOA, WOC, Added and Changed Elements, Interaction Analysis
Garcia et al. 2005	Modularization	CBC, CDC, CDLOC, CDO, DIT, LCOO, LOC, NOA, WOC
Greenwood et al. 2007	Stability in the face of Changes	CBC, CDC, CDLOC, CDO, DIT, LCOO, VS, LOC, NOA, WOC, Added and Changed Elements, Interaction Analysis
Kulesza et al. 2006	Modularization, Maintainability	CBC, CDC, CDLOC, CDO, DIT, LCOO, VS, LOC, NOA, WOC
Molesini et al. 2008	Stability in the face of Changes	Added and Changed Elements

CBC: Coupling between Components, CDC: Concern Diffusion over Components, CDLOC: Concern Diffusion over LOC, CDO: Concern Diffusion over Operations, DIT: Depth of Inheritance Tree, LCOO: Lack of Cohesion in Operations, LOC: Lines of Code, NOA: Number of Attributes, VS: Vocabulary Size, WOC: Weighted Operations per Component

Table 3.1: *Software measures used in settings with AspectJ and Java programs.*

we describe correctness and response times of answers, before we explore the relationship of software measures and program comprehension in Section 3.4, so that we get an impression of the nature of a relationship. We discuss whether software measures are reliable indicators for program comprehension in Section 3.5.

3.1 Background: Software Measures

The first software measures were developed to give project managers a tool to observe the progress of their monitored software projects [Henderson-Sellers, 1995, p. 24]. This way, the quality of projects could be evaluated during the development phase and threats to quality or delivery date could be responded to before it was too late.

Today, software measures are also often used in research to evaluate facets of new programming paradigms. For example, there is a large body of work analyzing the effect of aspect-oriented-programming techniques on software, which we summarize in Table 3.1. In this work, researchers compute software measures for different software systems and draw conclusions about the effect of aspect-oriented programming on facets of software quality, such as changeability or maintainability.

So, what are Software Measures? Melton and others define them as

> "measures that are obtainable directly from software documents" [Melton et al., 1990].

```
1  public class GcD {
2     static int number1, number2;
3     public static void main(String[] args) {
4        int temp;
5        do {
6           if (number1 < number2) {
7              temp = number1;
8              number1 = number2;
9              number2 = temp;
10          }
11          temp = number1 % number2;
12          if (temp != 0) {
13             number1 = number2;
14             number2 = temp;
15          }
16       } while (temp != 0);
17       System.out.println("result: " + number2);
18    }
19    public void setNumber1(int num){
20       number1 = num;
21    }
22    public void setNumber2(int num){
23       number2 = num;
24    }
25 }
```

Figure 3.1: *Source code to determine the greatest common divisor.*

They typically describe properties of source code. For better overview, we divide them into three categories: size measures, complexity measures, and concern measures. For each category, we present some measures and illustrate them with an algorithm to determine the greatest common divisor, which we show in Figure 3.1.

Our intention is not to give a complete overview of all measures (there are alone over 100 complexity measures [Zuse, 1991]), but to show the diversity and plausibility of software measures. For a more exhaustive overview of software measures, we recommend Henderson-Sellers [1995], Lorenz and Kidd [1994], or survey papers (e.g., Figueiredo et al. [2005]; Kafura [1985]; Riguzzi [1996]).

3.1.1 Size Measures

Size measures describe the size of a program [Henderson-Sellers, 1995, p. 87]. They were developed to measure the size of fixed-format assembler languages, which do not allow much variability in source code. Hence, solutions to a problem were of similar length.

There are numerous size measures, for example, based on lines of code or on tokens [Henderson-Sellers, 1995, pp. 88]. *Lines of code* describes the number of lines a program consists of. There are several variants of lines of code, such as including blank lines or comment lines. The most common variant is defined by Conte and others:

> "A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line" [Conte et al., 1986, p. 35].

Thus, lines of code for our example is 25.

Tokens are units of which a program consists, for example, variable names or keywords [Henderson-Sellers, 1995, p. 89]. Tokens describing data, such as identifiers, are called *operands*, and tokens describing actions, such as arithmetic symbols and keywords, are called *operators* [Halstead, 1977]. By combining operators and operands, we can define the *vocabulary* of a program (sum of unique operators and operands) or the *length* of a program (sum of total operators and operands). In our example, we have 10 unique operands (e.g., `number1`, `main`) and 17 unique operators (e.g., $<$, $=$), so our vocabulary is 27. The total sum of operands and operators is 36 and 56, respectively, so the length is 92.

Even in contemporary programming languages, it seems plausible that with increasing size, source code is more difficult to understand—simply because there is more code to look at.

3.1.2 Complexity Measures

In the 1970s, complexity of software started to become important, as systems grew larger and maintaining them became difficult [Henderson-Sellers, 1995, p. 57]. To monitor and improve the quality of large software systems, complexity measures were introduced. They try to capture

> "those characteristics of software that affect the level of resources used by a person performing a given task on it" [Henderson-Sellers, 1995, p. 166].

The earliest and still most widely used measure is called *cyclomatic complexity* [McCabe, 1976]. It is based on the control-flow graph of a method and counts the number of possible execution paths. For our example, cyclomatic complexity of the main method is 4: We have one loop `do... while`, two `if` statements, and, per default, each method has a complexity of 1.

Now, there are variants to define the complexity of a file. First, we can use the value of the most complex method, which is 4. Second, we can use the sum of the complexity values of all methods, which is 6. Finally, we can use the average complexity by dividing the sum of complexity values by the number of methods, which yields 2.

Complexity measures are based on the assumption that the more complex code is, the more difficult it is to understand. That sounds plausible, because the more branching statements there are, the more a developer has to consider, which requires more cognitive resources.

3.1.3 Concern Measures

Typical measures to describe concerns of a software system are *concern attributes* and *concern operations*, which represent, respectively, the number of attributes and operations assigned to a concern [Figueiredo et al., 2009]. In our example (cf. Figure 3.1), we have 2 attributes (`number1`, `number2`) and 3 operations (methods `main`, `setNumber1`, and `setNumber2`).

The more attributes and operations a concern defines, the more facets a developer has to consider to understand a concern, which should make it more difficult to understand.

Context	Description	Section
Objective	Analyze the relationship of software measures to program comprehension	3.2.1
Material	MobileMedia in two versions: AspectJ, Java ME with Antenna	3.2.2
Participants	21 graduate students from the University of Passau	3.2.3
Tasks	Maintenance tasks (locating bugs)	3.2.4
Execution	One computer lab; 19" TFT; predecessor of PROPHET	3.2.5
Analysis	Correctness of answers and response time	3.3
Result	No relationship of software measures and program comprehension	3.4

Table 3.2: *Experiment in a nutshell.*

Especially in systems, in which concerns are not encapsulated in a module, understanding should be impaired, because programmers have to trace attributes and operations of a concern across the entire system.

To summarize, software measures appear to be plausible indicators for program comprehension: The more lines of code or branching statements a program has, the more effort a programmer has to invest to understand it. Furthermore, computing software measures is easy, because they can simply be computed based on source-code properties. To this end, tools, such as SourceMonitor[1] or ConcernMorph [Figueiredo et al., 2009] were developed.

However, considering the complexity of the comprehension process, plausibility fades. Program comprehension does not only depend on source-code properties, but also on the person who is working with source code. Thus, it is unclear whether software measures are suitable to assess program comprehension.

3.2 Experimental Design

Since controlled experiments have proven useful to analyze cognitive processes [Goodwin, 1999, p. 100], we conducted an experiment to analyze whether software measures can be used to assess program comprehension. In Table 3.2, we summarize our experiment. In a nutshell, we used two comparable version of MobileMedia with different software measures and observed how students understand both systems. If there is a relationship between software measures and program comprehension, we should observe a difference in comprehension of our students. Our results did not show such a difference, indicating that software measures are not related to program comprehension.

[1] http://www.campwoodsw.com/source\monitor.html

3.2.1 Objective

The objective of our experiment is to evaluate the relationship between software measures and program comprehension. We included four software measures that we introduced in Section 3.1: lines of code as representative of size measure and cyclomatic complexity (average per file) as representative of complexity measure. We used these measures because they are the most widely used measure for size and complexity, respectively [Henderson-Sellers, 1995]. Since with our work, we focus on feature-oriented software development, in which separation of concerns is a key principle, we include both concern measures concern attributes and concern operations. This way, we can evaluate whether software measures specifically designed for feature-oriented software products differ from traditional software measures regarding their relationship to program comprehension.

Since we can argue (and others have) both in favor of and against a relationship between software measures and program comprehension, we state a research question:

RQ: Is there a relationship between software measures and program comprehension?

3.2.2 Material

As material, we use the last release of MobileMedia, because it was most suitable for our objective: The implemented features have considerably different software measures, which should make it easier for us to observe a difference in program comprehension of our participants.

To illustrate the difference in software measures, we give an overview in Table 3.3. We computed lines of code and complexity with SourceMonitor and by hand; concern attributes and concern operations with ConcernMorph [Figueiredo et al., 2009] and by hand. There are large differences between the two versions per feature, which are caused by the different implementation techniques, such that in the AspectJ version, source code is separated into more, but smaller modules.

If software measures indeed describe program comprehension, then this large difference in software measures should be reflected in a large difference in program comprehension. For example, concern *Video* in the Java version has a 5 times higher complexity value, $86\,\%$ more lines of code, and about 4 times more attributes and operations than in the AspectJ version. Hence, software measures suggest that the AspectJ version is more comprehensible than the Java version.

To illustrate the commonalities and differences of both versions, we show code excerpts of each version implementing the same functionality in Figure 3.2. The left part shows the AspectJ version, in which a *pointcut* (Lines 7 to 9) captures the execution of the method `initMenu()` in class `MediaListScreen`. When this method is executed, the *advice* (Lines 12 to 14) is executed, which adds the `sortCommand` to a menu. In the right part of Figure 3.2, we show the Java version, which uses ifdef directives to map the `sortCommand` in class `MediaListScreen`.

Concern	Version	LOC	Complexity	CA	CO
CountViews	AspectJ	319	0.97	2	21
	Java	1 268	3.39	27	41
PhotoAlbum	AspectJ	257	1.29	5	23
	Java	1 771	2.15	49	73
Favourites	AspectJ	257	1.70	3	19
	Java	1 268	3.39	27	41
Video	AspectJ	262	0.96	11	20
	Java	1 892	2.31	45	78
Music	AspectJ	326	1.05	12	24
/MMAPI	Java	2 081	2.32	63	88

LOC: lines of code; CA: concern attributes; CO: concern operations.

Table 3.3: *Software measures per concern.*

To control the level of experience of our participants with a certain tool (e.g., call hierarchy in Eclipse), we implemented our own tool infrastructure, a predecessor of PROPHET, with source-code viewing and a project-browsing component to display the source code. The tool uses Eclipse-like syntax highlighting and shows all files of the software system ordered by packages (similar to the package explorer in Eclipse). We implemented a logging functionality to track each action of our participants during the experiment. We also used this tool to display the descriptions of the tasks and capture the answers of participants.

In addition to the source code, participants got a feature diagram of MobileMedia on a sheet of paper and a mapping of files to concerns (cf. Section 2.3). Participants were familiarized with feature diagrams before the experiment. We provided both to ensure that participants direct their attention to those files that belong to a concern, which allows us to compare software measures and program comprehension at the concern level.

At the end of the experiment, we gave participants a debriefing questionnaire, in which we asked about the perceived difficulty of each task and motivation to solve each task, both on a five-point Likert scale [Likert, 1932]. Furthermore, we encouraged participants to leave comments about the experiment.

3.2.3 Participants

Participants were 21 graduate students at the University of Passau. They were enrolled in the course *Contemporary Programming Paradigms (German: Moderne Programmierparadigmen)*, in which advanced programming techniques, such as AspectJ and preprocessors, were taught and practiced. All participants were aware that they are participating in an experiment and that their performance does not affect their grade for the course.

We used a between-subjects design, so one group worked with the AspectJ version of MobileMedia (AspectJ group), the other group with the Java version (Java group). We

```
1  public privileged aspect CountViewsAspect {
2
3  // 147 additional lines of code
4
5
6    public static final Command sortCommand =
7       new Command("Sort by Views",
8       Command.ITEM, 1);
9
10
11   // pointcut declaration
12   pointcut initMenu(MediaListScreen screen):
13      execution(public void MediaListScreen.
14      initMenu()) && this (screen);
15
16   // advice code
17   after(MediaListScreen screen):
18      initMenu(screen) {
19      screen.addCommand(sortCommand);
20   }
21 // 66 additional lines of code
22 }
```

```
1  public class MediaListScreen
2       extends List {
3  // 39 additional lines of code
4
5    // #ifdef includeCountViews
6    public static final Command
7     sortCommand = new Command(
8     "Sort by Views", Command.ITEM, 1);
9    // #endif
10
11 // 19 additional lines of code
12   public void  initMenu() {
13 // 40 additional lines of code
14
15
16     // #ifdef includeCountViews
17     this.addCommand(sortCommand);
18     // #endif
19     // 6 additional lines of code...
20   }
21
22 }
```

Figure 3.2: *Comparison of AspectJ and Java version of MobileMedia. Left: AspectJ version, showing pointcut expression and advice code. Right: Java version, showing #ifdefs to annotate code fragments.*

decided against a within-subjects design, because the experiment would have lasted too long, leading to fatigued participants. Additionally, participants could have learned from the version they started with, so we could not be sure how confounded our result would be. Nevertheless, since both versions of MobileMedia were designed to be comparable, our setting allows us to draw sound conclusions without stressing our participants too much.

To form two comparable groups, we measured programming experience with a predecessor of the questionnaire that we describe in Chapter 5. In short, we asked participants to estimate their experience with several programming languages and paradigms on a five-point Likert scale, as well as the size of projects they have worked with. A low value in the questionnaire (minimum: 5) indicates no programming experience; the higher the value is, the more programming experience participants have (high value: 60, the scale is open ended). The mean programming experience of the AspectJ group is 41.9 (standard deviation: 10.6), and of the Java group 40.8 (standard deviation 10.5). We had 21 participants, of which one was female—she was in the Java group. Altogether, there were 10 participants in the Java group and 11 participants in the AspectJ group.

To account for the possibly more complex nature of AspectJ, we did not introduce bugs in highly syntax-specific code. Since AspectJ is an extension to Java, its syntax is based on Java, and introduces additional elements, such as *pointcuts* (roughly similar to pattern matching) and *advice* (roughly similar to Java syntax). We introduced the bugs only to advice code and made sure that the claimed benefits of aspect-oriented programming for program comprehension, such as separation of concerns, could still be measured. Since participants did not have to implement any source code or understand com-

Task	Bug description	Concern
1	When creating/converting media, the counter how often a medium was shown, is always set to 0.	*CountViews*
2	If a picture of a photo album should be displayed (action "View"), nothing is displayed, although the photo album is not empty.	*PhotoAlbum*
3	Although several pictures are set as favorites, the action "View favorites" is not displayed in the menu. The developer claims, he implemented the according functionality.	*Favourites*
4	The option to play a video (command "Play video") is not displayed.	*Video*
5	If you click in the menu on "Play music", no music is played, although the according functionality is implemented.	*Music/ MMAPI*
6	If you click on "View favorites" in the menu, no favorites are shown, although favorites exists and the according functionality is implemented.	*Favourites*

Table 3.4: *Overview of Tasks.*

plex pointcut declarations, a thorough understanding of AspectJ syntax is not necessary in our experiment.

3.2.4 Tasks

We created six maintenance tasks and gave participants a bug description as a user might provide it for each task. In addition, we provided the concern in which the bug occurred, to ensure that participants focus on concern code. Second, we opened for each task all files that belong to the according concern. However, participants could open all other files of MobileMedia, if they thought it was necessary (e.g., to trace a method call). Thus, we can evaluate program comprehension on the concern level. To solve a task, participants should locate the position at which the bug occurs (file and line), state why the bug occurs, and suggest a solution. We used all information to decide whether a bug was identified correctly. Additionally, we measured the time participants needed to solve a task (referred to as response time). For all tasks, we carefully introduced bugs into the source code.

We present all tasks in Table 3.4 to give an overview. To better understand of the nature of bugs, we describe the cause of the first bug in detail. It was caused by setting a variable that counts the number of views to 0, instead of setting it to the correct value. In the AspectJ version, the bug was located in aspect *CountViewsAspect* in Line 223, where the code stated `mediaData.setNumberOf Views(0)` instead of `mediaData.setNumberOfViews(numberOfViews)`. In the Java version, the bug was located at a corresponding position in class `MediaUtil`. In Line 151, the code stated `ii.setNumberOfViews(0)` instead of `ii.setNumberOfViews(numberOfViews)`.

In addition to these six tasks, we designed a warming up task to let participants familiarize with the experimental setting. In the AspectJ version, participants should count the number of pointcuts of the concern *PhotoAlbum*, in the Java version how often the command `includeFavourites` of the concern *Favourites* occurs. To have comparable effort for both tasks, the same number of files was opened and had to be looked at and about the same number of occurrences existed. This task is not included in the analysis.

3.2.5 Experiment Execution

The experiment was conducted in July 2010 instead of a regular lecture session in a lab room with Linux computers and 19" screens. We gave an introduction to all participants, in which we explained important facets of the experiment and repeated facts from their programming course relevant for the experiment as a reminder. After the introduction, participants were seated at a computer and started to work on the tasks on their own. Four experimenters regularly checked that participants worked as planned. After participants were finished, they were instructed to raise their arm, so that we can give them the debriefing questionnaire. After completing the questionnaire, participants were instructed to leave quietly without disturbing the others. There are no deviations to report.

3.3 Experiment Results

In this section, we present the results of our experiment regarding program comprehension. To evaluate program comprehension, we measured correctness and response time of an answer.

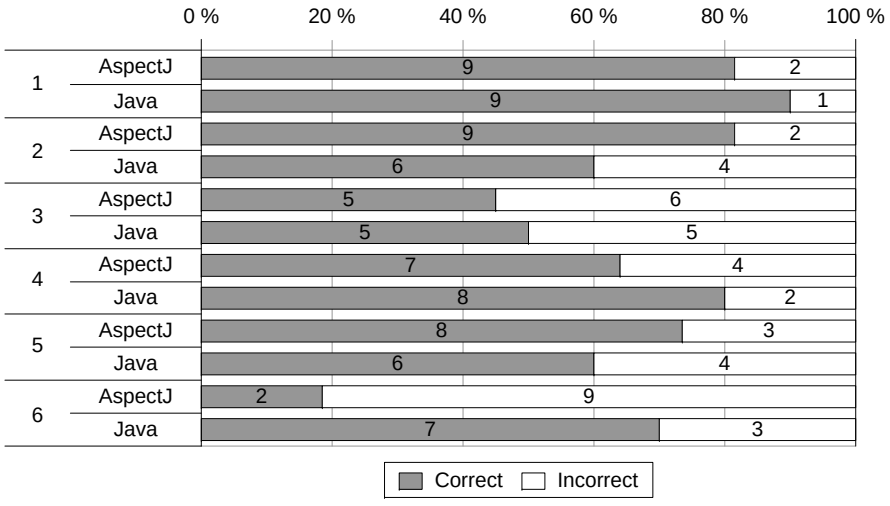
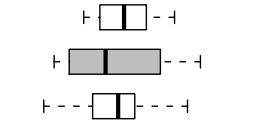


Figure 3.3: *Number of correct answers per group and task.*

Correctness In Figure 3.3, we give an overview of the number of correct answers per task and group. For most of the tasks, both groups have about the same number of

Variable	Version	Distribution	Mean	N	t/U	p
Task 1	AspectJ		956.55	11	1.032	0.321
	Java		883.6	10		
Task 2	AspectJ		875.91	11	-0.476	0.646
	Java		1056	10		
Task 3	AspectJ		807.64	11	20	0.151
	Java		508.8	10		
Task 4	AspectJ		669.64	11	36	0.397
	Java		471.5	10		
Task 5	AspectJ		406.82	13	22	0.18
	Java		683.1	10		
Task 6	AspectJ		631.73	11	–	–
	Java		459.4	10		
		0 10 20 30				

Only for the significance tests, we omitted response times for wrong answers; there are too few data for Task 6 to conduct a significance test; t/U: test values; t for Tasks 1 and 2, U for Tasks 3 to 5

Table 3.5: *Response time of participants.*

correct solutions. For the sixth task, there is a large difference: Only two participants of the AspectJ group entered the correct solution.

To evaluate whether there are significant differences in the number of correct solutions, we conducted Fisher's exact test. We cannot conduct the χ^2 test, because expected frequencies are smaller than 5. Only for the last task, we found a significant difference in the number of correct solutions ($p = 0.03$). For all other tasks, the p values range from 0.361 to 1.

Response Time In Table 3.5, we show the mean response times of participants for each task. For the Tasks 2 and 5, participants of the AspectJ group were faster; for the remaining tasks, participants of the Java group were faster. The largest differences appear in Task 5 in favor of the AspectJ group, and in Task 3 in favor of the Java group.

For completeness, we show the response times for all tasks in Figure 3.4. The difference in response time is negligible with $2\,\%$ (1.2 minutes, compared to almost 1.5 hours for all tasks). A t test shows that the difference is not significant (t value: 0.1715, p value: 0.866).

Before conducting significance tests for response times, we have to consider whether a task was solved correctly or not, because response times differ for correct and incorrect solutions [Yellott, 1971]. For example, a participant might enter a wrong answer deliberately, for example, to be finished with a task, which would bias the response time. We could compute efficiency measures, for example, combinations of correctness and response times, but it is not clear what they mean. Hence, we excluded response times

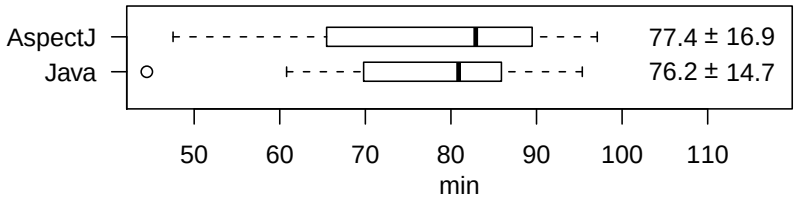


Figure 3.4: *Response time of participants for all tasks. The numbers indicate mean and standard deviation.*

of wrong answers from the analysis. For the last task, this leaves us with only 2 values for response times in the AspectJ group, so we cannot conduct a significance test for this task.

To check whether the observed differences in response time are significant, we conducted a t-test for Task 1 and 2, and a Mann-Whitney-U test for Tasks 3 to 5, in which the response times are not normally distributed. In Table 3.5, we show the distribution of response times for each task (including wrong answers) and the results of the significance tests (without wrong answers). All p values are larger than 0.05, meaning that none of the differences is significant.

Taking both, the results for correctness and response time into account, we found no significant differences for program comprehension between the AspectJ and Java version, except for the correctness of one task. We believe there are two reasons responsible for the difference in correctness: First, participants of the AspectJ group estimated this task as more difficult (U value: -3.029; $p < .05$), and their motivation as lower (U value: -3.079; $p < .05$). Second, when we told participants the group assignment, several participants of the AspectJ group complained, but none of the Java group. We believe that both affected the performance negatively, resulting in fewer correct answers for the last task.

3.4 Software Measures and Program Comprehension

Having found (almost) not differences in program comprehension, we can conjecture that both versions are equally comprehendable, which is in contrast to what the software measures suggest (i.e., that the AspectJ version is more comprehendable).

After evaluating our research question, we go one step further and explore our data for a possible relationship. During this process, we refine the computation of software measures by including the behavior of participants, such that we can have a detailed look on how software measures and program comprehension could correlate. This data exploration may sound like "fishing for results". However, since we defined a research question (i.e., is there a relationship between software measures and program comprehension?), exploring our data is a legitimate step to answer this question. Furthermore, we defined the modification of software measures before starting the exploration, so we do not juggle with the numbers until finding something interesting. With our exploration, we provide some insights into possible relationships and concrete research hypotheses for future experiments.

Version	LOC	Complexity	CA	CO
AspectJ	6 717	1.6	477	182
Java	5 397	2.0	271	165

LOC: lines of code; CA: concern attributes; CO: concern operations.

Table 3.6: *Software measures of MobileMedia.*

3.4.1 Software Measures of Complete System

Although software measures are often calculated in terms of concerns, we start in a general way by comparing the entire application for completeness. In Table 3.6, we present an overview of software measures for the complete system. The AspectJ version has more lines of code, more attributes, and more operations. In contrast, the complexity value is smaller in the AspectJ version. Since we did not observe a significant difference in program comprehension that reflects this difference in software measures, we cannot confirm a relationship between software measures and program comprehension on the level of the complete program. Next, we look at the concern level.

3.4.2 Software Measures in Terms of Concerns

Software measures for the entire system do not necessarily reflect the subsystem analyzed for a specific task. Hence, we compare the software measures in terms of concerns with program comprehension as we observed it. In Table 3.7, we show the software measures of both systems in terms of concerns. The software measures for each concern of the AspectJ version are smaller, that is, suggest better comprehensibility. This means that the AspectJ group should make fewer errors and be faster for every task. However, we could not find such a difference. Only for correctness of the last task, we discovered a significant difference, but in favor of the Java group (opposite of what the measures suggest). Hence, we cannot confirm that software measures and program comprehension correlate when considering the concerns participants worked with.

One might argue that when participants worked on a task, they did not only look at files that belong to the according concern, but opened other files, as well. Hence, we should compute software measures based on the *files participants actually looked at*. Since we logged what participants did during working on a task, including opening files, we are able to compute software measures based on the files participants looked at.

3.4.3 Software Measures Related to Files

To compute the software measures related to files for a single task, we determined *personal* software measures for each participant and computed their mean in three steps. We describe this aggregation process for one task. First, we extracted all files a participant looked at and determined the software measure for each file. Second, we computed the average complexity value, and the sum for lines of code, concern attributes, and concern operations, respectively, of all files. Hence, each participant has an own *personal* value for complexity, lines of code, concern attributes, and concern operations. Finally, we

Feature/Task	Version	LOC	Complexity	CA	CO
CountViews	AspectJ	319	0.97	2	21
Task 1	Java	1 268	3.39	27	41
PhotoAlbum	AspectJ	257	1.29	5	23
Task 2	Java	1 771	2.15	49	73
Favourites	AspectJ	257	1.70	3	19
Task 3 & 6	Java	1 268	3.39	27	41
Video	AspectJ	262	0.96	11	20
Task 4	Java	1 892	2.31	45	78
Music/MMAPI	AspectJ	326	1.05	12	24
Task 5	Java	2 081	2.32	63	88

LOC: lines of code; CA: concern attributes; CO: concern operations.

Table 3.7: *Software measures per feature.*

computed these personal measures for each participant and averaged over the personal software measures of all participants. We repeated this process for each task.

We summarize the results of the adapted software measures in Table 3.8. The difference of software measures between both versions is smaller now for most tasks and measures. This indicates that, if we take into account what participants actually did, software measures better reflect program comprehension. However, the differences between software measures are still too large, compared to the fact that we did not observe significant differences in program comprehension.

Now, participants looked at one file only for a few seconds, but several minutes at another file. Hence, the *time a participant looked at a file* should also be considered, because the file at which a participant looked longer should have more influence on the personal software measure. Since we logged the time participants looked at a file, we can compute weighted personal software measures.

3.4.4 Software Measures Weighted with Response Time

To compute weighted software measures for a single task, we again computed personal software measures for each participant, and additionally took into account the time a participant spent with a file. We describe this approach for a single task. First, we divided the time a participant spent with each file by the complete time for a task. Second, we multiplied this value with the according software measures for a file. Thus, if a participant looked at a complex file only for a few seconds, the weighted value for complexity is low. Hence, for each file and each participant, we got *personal weighted* values for lines of code, complexity, concern attributes, and concern operations, respectively. Finally, we proceeded as described in the previous section for the computation of software measures related to files, based on the *personal weighted* software measures.

Task	Version	LOC	Complexity	CA	CO
1	AspectJ	440.18	1.12	8.64	29.64
	Java	1 290.30	2.95	28.50	39.70
2	AspectJ	409.36	1.38	7.36	30.55
	Java	1 658.20	2.24	44.90	66.70
3	AspectJ	369.18	1.66	5.55	25.27
	Java	1 149,50	3.04	27.50	34.80
4	AspectJ	481.00	1.02	15.36	32.91
	Java	1 210.30	3.07	31.70	44.40
5	AspectJ	382.82	1.33	12.27	28.27
	Java	1 896.30	2.58	55.70	74.50
6	AspectJ	501.27	1.45	8.82	33.09
	Java	1 167.90	3.48	24.20	36.30

LOC: lines of code; CA: concern attributes; CO: concern operations.

Table 3.8: *Software measures per task.*

In Table 3.9, we present the mean of weighted software measures for each task. The difference between the software measures of both versions got smaller again, compared to the unweighted values. Especially the complexity values are all smaller than 1, which indicates that complexity can be an appropriate measure if we consider participants' behavior. The smaller difference for the other software measures aligns better with the results of our experiment, as well.

Another interesting observation for weighted software measures is that the weighted value for concern operations is smaller in the Java version; most likely, because participants of the AspectJ group spent most of their time in aspects with a large concern-operations value, because those contained most of the implementation of a concern. In contrast to the AspectJ group, the Java group looked at more files per task, so the time per file is considerably smaller. Multiplying this small value with the concern-operations value results in smaller weighted concern-operations values, and, thus, in a smaller weighted value.

Finally, one might argue that we should also take into account the methods of each file participants looked at, because a very complex method may be somewhere in a file where a participant did not even look. Unfortunately, this is difficult to assess reliably in an experimental setting without eye-tracking software. We could take code displayed on a screen at any time (start line – end line) as indicator, but this does not allow us to deduce at which method on the screen participants looked, or whether they just scrolled through the code. Nevertheless, this would be an interesting challenge for future experiments, for example, by using an eye-tracking system.

Task	Version	LOC	Complexity	CA	CO
1	AspectJ	210.07	0.50	2.26	15.88
	Java	254.82	0.67	3.47	6.35
2	AspectJ	134.82	0.28	3.14	12.15
	Java	232.33	0.31	3.83	7.10
3	AspectJ	176.31	0.53	2.74	14.47
	Java	293.34	0.87	6.56	6.73
4	AspectJ	145.64	0.27	5.13	11.89
	Java	162.95	0.87	10.06	3.96
5	AspectJ	207.43	0.60	6.26	15.88
	Java	250.16	0.29	5.44	6.95
6	AspectJ	174.77	0.52	2.67	13.89
	Java	310.22	0.85	4.67	7.61

LOC: lines of code; CA: concern attributes; CO: concern operations.

Table 3.9: *Weighted software measures per task.*

3.5 Discussion

So, can we use software measures to measure program comprehension? Although we refined software measures, such that they better fit the behavior of participants, none of the refinements was entirely satisfactory. The values of lines of code, concern attributes, and concern operations still differed considerably, which is not reflected by program comprehension as we measured it. For complexity, we found that the weighted value is similar for both versions (all smaller than 1). For all other software measures, we cannot confirm a relationship between software measures and program comprehension.

The reader may have noticed that using the observed data to refine software measures eliminates the benefit of easy computation: Instead of basing the computation of measures solely on source code, we included what participants did. This approach is not feasible in practice, because we cannot predict which developers work with the source code or how long they look at what file. Hence, adapting software measures is not a practical way to improve the predictive power of software measures.

Thus, we have to conclude that we cannot use software measures to assess program comprehension. Even personal software measures to not predict the comprehensibility of source code. Instead, software measures should only be used for what we know they describe, for example, the size of a program, the complexity of a method, or the size of a concern. If we need to evaluate the comprehensibility of software, there is no way around controlled experiments.

Nevertheless, for initial research on a new concept, plausibility discussions with software measures are helpful to establish research hypotheses regarding benefits and drawbacks. However, such hypotheses should be evaluated empirically eventually. This helps us to discover possible hidden relationships, to describe and evaluate claimed benefits of

a concept more easily, and to gain a more thorough understanding of the relationship of software measures and program comprehension. Furthermore, our results and proceedings can act as inspiration to develop and evaluate new software measures that better describe comprehensibility of source code.

3.6 Threats to Validity

Threats to internal validity are caused by the low experience of our participants with AspectJ and by exploring the data. Threats to external validity are rooted in using only one software system, four software measures, and limiting our study to program comprehension.

3.6.1 Internal Validity

One problem of our study is the experience of our participants with AspectJ. They were introduced to AspectJ in the course they were enrolled in, whereas they worked with Java since they started to study. To diminish the influence of experience with AspectJ, we made sure that for understanding the cause of bugs, participants did not need a deep understanding of AspectJ syntax. To further reduce the influence of AspectJ experience, we did not let participants implement source code, but only explain the problem and suggest a verbal solution. This way, participants did not have to implement AspectJ code. Hence, the experience of participants with AspectJ was sufficient for our purpose.

Another issue is that we explored the data, which can easily drift off to "fishing for results". However, we did not exploit our data until we found interesting results, but made some reasonable, well-defined refinements to the computation of software measures. Our data exploration is rather a benefit, because we obtained some insights of a possible relationship of software measures and program comprehension, which should be evaluated in further experiments.

3.6.2 External Validity

For our study, we used only one software system. However, using MobileMedia has the benefit that our results are comparable with numerous results of other researchers, who also used MobileMedia in their work. Consequently, the generalizability to other research with MobileMedia is given, but not the generalizability to other software systems. Thus, the restriction to MobileMedia is both a benefit and a drawback for external validity.

A further restriction is that we only used four software measures. Our results are only applicable to these software measures. To limit this restriction, we used a representative measure of every category we described in Section 3.1. This allows us to carefully draw conclusions for the categories of software measures. Nevertheless, to be able to state a relationship of other software measures (e.g., coupling and cohesion) to program comprehension, they should also be evaluated in a carefully designed experiment. Here, we showed what a carefully designed setting looks like.

Furthermore, we only evaluated how program comprehension and software measures could correlate. We cannot generalize from program comprehension to other software quality facets, such as maintainability and design stability, and their relationship to software measures, which was the focus of numerous studies (cf. Table 3.1).

3.7 Related Work

There is a lot of work regarding software measures. We already mentioned one line of research that develops and tests measures to evaluate quality properties of aspect-oriented software systems [Figueiredo et al., 2008a,b; Greenwood et al., 2007; Molesini et al., 2008]. In this work, several software projects are evaluated with the developed software measures. For example, Figueiredo and others [2008a] assess the design stability of MobileMedia based on software measures. To this end, MobileMedia was developed in two versions in several scenarios, while with every scenario, the program was extended. Based on software measures, both versions were compared. However, the studies did not include the behavior of human participants to assess quality properties.

On the other hand, there is also empirical research with human participants regarding program comprehension, in which properties of source code, such as depth of inheritance hierarchies [Daly et al., 1995], comment style [Prechelt et al., 2002], and identifier styles [Sharif and Maletic, 2010] were evaluated regarding their effect on comprehensibility. This is similar to our work, in which we assessed the comprehensibility of two systems. However, we did not evaluate whether several facets of source code influence program comprehension. Instead, we were only interested in whether we could observe a difference in comprehensibility.

Another line of work evaluates empirically the benefit of aspect-oriented programming compared to object-oriented programming [Hanenberg et al., 2009]. In an experiment, participants implemented crosscutting code into a small target application, one implemented in AspectJ, the other in Java. Depending on the kind of code changes, AspectJ had a positive or negative influence on the development time of participants. Like in our experiment, students were recruited as participants, which were introduced to AspectJ, however, without a link to software measures.

Another research topic that addresses the costs produced by maintenance are bug prediction models [Bettenburg and Hassan, 2010; Zimmermann et al., 2007]. In such models, properties of source code and change logs are analyzed so that possible bugs can be predicted. This is similar to computing software measures based on source code to describe its comprehensibility. However, the goal is to predict future bugs. Bettenburg and Hassan [2010] analyzed social facets of the concurrent-version-system entries six months before and after a major Eclipse release and derived a regression model based on those logs. Zimmermann and others [2007] analyzed complexity measures of several Eclipse releases and found that the more complex code is, the more defects it has. It could be an interesting step to combine bug prediction models with software measures, such that the prediction, where bugs might occur, can be used to adapt software measures.

3.8 Summary

Software measures are often used to assess comprehensibility of source code. However, software measures are based on properties of source code and do not consider the developer. In this chapter, our goal was to give recommendations about using software measures as program-comprehension indicators based on empirical evidence.

To fulfill our goal, we conducted a controlled experiment, in which we gave two groups of participants two versions of MobileMedia with different software measures. If there is a relationship between software measures and program comprehension, we should also see a difference in the comprehension between both groups.

We focused on measures for feature-oriented software systems, because feature-oriented software development is the focus of our research, and because software measures are especially used to evaluate quality facets of feature-oriented software systems.

We did not find any differences, indicating that the software measures we used are not suitable to measure program comprehension. Even after refining software measures by taking into account the files participants looked at and how long they looked at each file, we did not find a relationship of software measures and program comprehension (except for one complexity measure). Thus, we cannot recommend software measures as indicators for program comprehension of software systems. Furthermore, the effect of feature-oriented software development (and other modern programming paradigms) on program comprehension should not solely be evaluated based on software measures. Instead, we recommend to conduct controlled experiments, in which we take into account the developer.

To reduce the effort of controlled experiment, we developed several tools and guidelines, which we present in the following chapters.

Chapter 4

Confounding Parameters for Program Comprehension

This chapter shares content with a submitted article.

In the previous chapter, we concluded that we need to conduct controlled experiments to reliably measure program comprehension. To this end, we need to control for confounding parameters, which threaten the validity of results and may lead to false conclusions. To control for confounding parameters, we need to identify them first.

Based on our experience with planning experiments and based on discussions with other researchers and students, we found that identifying and controlling for confounding parameters is one of the major obstacles for conducting experiments. Since we aim at reducing the effort for conducting controlled experiments with our work, we start by reducing the effort of identifying and controlling for confounding parameters. To this end, we conducted a literature survey of controlled experiments, which we present in this chapter. Our goal is the following:

- An extensive list of confounding parameters for program comprehension and control techniques.

With an extensive list, researchers do not have to identify confounding parameters, but can simply consult our list and decide for each parameter whether it has an important influence or not. Furthermore, with recommendations how to control for a confounding parameter, we support researchers in selecting a suitable control technique and, thus, in producing valid experimental results.

To have a better impression of the relevance of our goal, we discuss the importance of confounding parameters in Section 4.1. In Section 4.2, we describe the selected journals and conferences, how we selected relevant papers, and how we extracted confounding parameters. This way, we enable researchers evaluate the suitability of our approach and extend our work based on the same approach. To give an overview of the state of the art and derive recommendations for its improvement, we discuss how researchers control for confounding parameters in Section 4.3. In Section 4.4, we describe five common techniques to control for confounding parameters to enable researchers to select appropriate

control techniques. In Section 4.5, we describe all 39 confounding parameters we identified and discuss how each of them can be controlled. We give recommendations how to manage confounding parameters in Section 4.6.

4.1 Importance of Confounding Parameters

When conducting experiments, we have to ensure internal and external validity (cf. Section 2.2.2). When recruiting human participants, this is especially problematic, because there are considerable inter-individual differences: Participants have different experience with programming, different intelligence, different familiarity with different domains, and so on. If we do not consider these differences, our results are most likely biased. That is, variations in the outcome are not caused by the intentional variations of the independent variable, but by unintentional variations of the confounding parameters (i.e., the inter-individual differences). To reduce this bias, we can restrict our population, for example, to professional programmers only, but even in this population, there are considerable differences in development and maintenance time McConnell [2011]; Sackman et al. [1968]. Thus, controlling for confounding parameters is not as straightforward as it appears.

In the context of psychology, many confounding parameters for experiments are well known, because experimentation has a long history [Wundt, 1874]. For example, during evaluation of productivity of employers in the Hawthorne Works, experimenters found that higher illumination increased productivity [Roethlisberger, 1939]. However, not the higher illumination caused the increase, but simply the observation of the employees, which was discovered in subsequent observations. In the context of program comprehension, there are in addition to the well known parameters other specific confounding parameters, such as programming experience or domain knowledge.

To avoid bias due to confounding parameters, we can either increase our sample size or reduce variation in confounding parameters. When increasing the sample size, we need to consider the number of independent variables and their levels. For example, if we have two independent variables with two levels each, we have four experimental groups (cf. Section 2.2.2). For each group, we need about 10 participants to sufficiently control the bias, leading to a sample size of 40 participants. However, there are not only two confounding parameters, but considerably more. In our review, we identified 39. Assuming two levels per parameter, we have $2^39 = 549\,755\,813\,888$ experimental groups. Thus, increasing the sample size is not feasible. Even if we select only the five most important parameters, we would still need $2^5 \times 10 = 320$ participants to produce sound results. This sounds more feasible, but typically, it is difficult to find that many participants for a study.

Thus, it is more feasible to reduce the variation of a confounding parameter. To this end, we control their influence, for example, by ensuring comparable groups of participants or by including only one level, such as low programming experience. The first step in controlling for confounding parameters is identifying them. Thus, with fulfilling our goal, we support researchers in producing sound experimental results.

4.2 Methodology of Literature Survey

Since our goal is to provide an extensive list of confounding parameters, we selected the following broad, representative set of journals and conferences:

Empirical Software Engineering (ESE)

Transactions on Software Engineering and Methodology (TOSEM)

Transactions on Software Engineering (TSE)

International Conference on Program Comprehension (ICPC)[1]

International Conference on Software Engineering (ICSE)

Symposium on the Foundations of Software Engineering (FSE)

International Symposium on Empirical Software Engineering and Measurement (ESEM)

We selected ESE as leading platform for empirical research in the field of software engineering. We included TOSEM and TSE as leading journals in software engineering. ICPC is the leading conference for research regarding program comprehension. ICSE and FSE are the leading conferences on software engineering. Additionally, we chose ESEM as platform in the empirical-software-engineering domain. From each journal and conference, we considered all papers published between 2001 and 2010.[2] Hence, we have a representative set of journals and conferences.

From these journals and conferences, we selected all papers that report an experiment. Since there are different kinds of experiments and only certain kinds are relevant for our survey, we give a short overview of different types of experiments (see, e.g., Sjøberg et al. [2005]).

Background: Types of Experiments In general, a setting in which a treatment is deliberately applied to a group of participants is called an *experiment*. We can describe the following types of experiments:

- Randomized experiment
- Quasi experiment
- Correlational study
- Case study

First, if participants are randomly assigned to treatment and control condition(s), an experiment is referred to as *randomized experiment*. Second, in a *quasi experiment*, participants are not assigned randomly to conditions, for example, when groups are already

[1] ICPC was a workshop until 2005.

[2] ESEM started in 2007, so we have papers of only four years for this conference.

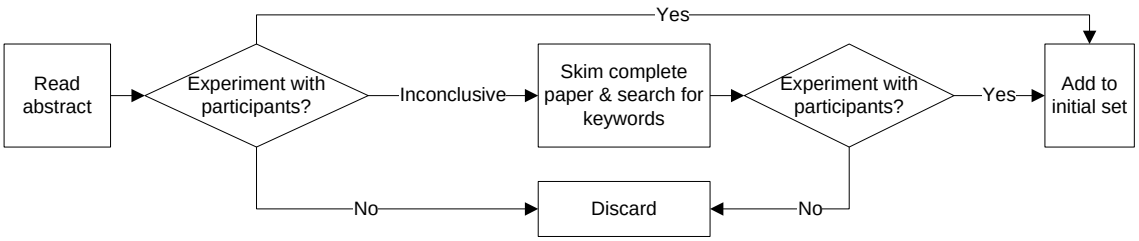


Figure 4.1: *Approach to select papers that describe experiments with participants.*

present (which is often the case in studies conducted in companies). Third, in a *correlational study*, size and direction of relationships among variables are observed, often on existing data. Fourth, in *case studies*, only one or few participants are observed and the outcome has a qualitative nature.

For our survey, we include all types of experiments except for correlational studies that observe only existing data and do not recruit human participants. For example, Bettenburg and others [2010] analyzed the commit data of the development of an Eclipse version six months before and after its release to identify how commit comments help to predict bugs. Since this experiment was not conducted with human participants, we excluded it. For simplicity, we do not differentiate different kinds of experiments from here on.

To extract relevant papers from the selected journals and conferences, we used the following procedure: First, we read the abstract of a paper. If the abstract describes an experiment with human participants, we added the paper to our initial set; if not, we discarded it. If the abstract is inconclusive, we skimmed through the paper for any information that indicates the conduct of an experiment. Furthermore, we searched the paper with a fixed set of keywords: (programming) experience, expert, expertise, professional, subject, and participant. Those keywords are typical for program-comprehension experiments with human participants. Based on the skimming and search result, we either added a paper to our initial set or discarded it. To have a better understanding of our approach, we visualize it in Figure 4.1. As result of this selection process, we have an initial set of 291 papers.

As next step, we read each paper of our initial set completely. During that process, we discarded some papers, because the described experiment was too far away from program comprehension. Before discarding a paper, we discussed whether it is relevant for us until we reached inter-personal consensus. When in doubt, we included a paper, because it is better to have more parameters in our list than missing one. This way, researchers can decide whether a parameter is relevant or not. We excluded 133 irrelevant papers, so we had 158 papers in our final set. On the project's website, we have a current list of all extracted papers, including the ones we discarded. In Table 4.1, we show how many papers from which journal and conference we extracted and included in the final set.

As last step, we extracted confounding parameters. To this end, we included variables that authors categorized as confounding variables (e.g., some authors listed these

Source		2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	Sum
ESE	All	24	18	15	15	19	21	24	26	24	16	**202**
	Extr.	2	9	5	8	10	7	7	4	7	3	**62**
	Final	1	5	1	4	3	3	3	2	5	2	**29**
TOSEM	All	11	15	13	10	12	12	15	21	13	13	**135**
	Extr.	0	0	0	0	1	0	3	0	0	3	**7**
	Final	0	0	0	0	0	0	2	0	0	0	**2**
TSE	All	66	73	88	72	68	61	55	53	50	48	**634**
	Extr.	5	6	6	5	5	3	2	5	3	5	**45**
	Final	4	4	5	3	3	3	1	5	2	3	**33**
ICPC	All	28	24	22	21	24	23	22	21	22	16	**223**
	Extr.	2	3	3	8	8	3	4	5	4	3	**43**
	Final	2	2	3	3	7	3	4	5	4	3	**36**
ICSE	All	47	48	42	58	44	36	49	56	20	52	**482**
	Extr.	8	10	8	1	11	12	9	8	9	12	**88**
	Final	0	3	3	4	3	4	1	3	4	5	**30**
FSE	All	29	17	42	25	32	25	63	31	39	34	**337**
	Extr.	0	1	0	0	1	2	3	3	1	1	**12**
	Final	0	1	0	0	0	2	2	3	1	0	**9**
ESEM	All	-	-	-	-	-	-	45	29	44	30	**148**
	Extr.	-	-	-	-	-	-	12	3	11	8	**34**
	Final	-	-	-	-	-	-	6	5	1	7	**19**

All: All papers of the source in the according year. Extr.: Extracted papers in our initial set. Final: Papers in the final set (after discarding non-relevant papers).

Table 4.1: *Overview of all, included, and extracted papers by year and journal/ conference.*

variables in a table or used terms like "Our confounding parameters are" to describe them). Furthermore, we included variables that followed terms like "To control for", "To avoid bias due to", or "A threat to validity was caused by", because such a variable was treated as confounding.

The complete selection and extraction process was conducted by the author and a research assistant (Jana Schumann, University of Magdeburg). To reduce bias, we checked the selection and extraction work of the other researcher on random samples, as suggested by Kitchenham and Charters [2007]. We discuss the validity of this approach more detailed in Section 4.7. To give an impression of the effort of the selection and extraction process, we estimated the time we needed, which is about 63 work days (à 8 hours). When extending our list, researchers can use these numbers to estimate their effort.

4.3 State of the Art

To get an impression of the state of the art, we present insights of how confounding parameters are managed in literature. The main findings are:

- There is no systematic way to describe confounding parameters
- Researchers use different ways to measure and control for the same confounding parameter
- Only a fraction of identified confounding parameters are considered in each paper

We discuss each of the findings in detail in this section.

4.3.1 Describing Confounding Parameters

In our survey, we found that confounding parameters are described at different places in the papers, which typically are the following [Jedlitschka et al., 2008]:

- Experimental design
- Analysis
- Interpretation
- Threats to validity

In experimental design, authors describe the setting of an experiment, including material, participants, and means to control confounding parameters. In the analysis, the authors present the data analysis, for example, means, standard deviations, and statistical tests. After the analysis, the results of the experiment are interpreted and set in relation to the research hypotheses or questions. Finally, authors discuss the validity of the experiments.

In Table 4.2, we give an overview in which parts confounding parameters were introduced. Column "N" contains the total amount of how often parameters were mentioned in each section; mean denotes the average, relative amount of parameters of all papers mentioned in the according section. Most parameters were discussed during the experimental design. This is not surprising, because in this stage, means to manage confounding parameters are typically discussed. Next, in part threats to validity, 17 % of the confounding parameters are described. In this part, authors mostly describe confounding parameters, how they could have threatened the validity of the experiments, and how they controlled for a parameter so that the threat to validity is minimized. Only a small part of the parameters is described in the analysis and interpretation.

Thus, the major part of confounding parameters is introduced in the experimental design. However, about a fifth of the confounding parameters was not mentioned in experimental design, although we specify in the design phase how we manage confounding parameters (cf. Section 2.2). As a consequence, readers of a report might overlook a parameter, because it is mentioned only implicitly. Thus, they cannot be sure whether

Part	N	Mean
Experimental design	716	79.14 %
Analysis	22	1.78 %
Interpretation	19	1.91 %
Threats to validity	199	17.17 %

N: number of mentions of paramater; Mean: relative amount of mentions of parameters

Table 4.2: *First mentions of a parameter per part of the experiment description.*

all relevant parameters have been considered. Consequently, readers might misjudge the soundness of an experimental design.

Furthermore, we found different ways of describing confounding parameters and according control techniques. In some papers, confounding parameters were introduced in a table (along with the other experimental variables), which often did not necessarily include all confounding parameters, but some were introduced as threat to validity. In other papers, confounding parameters were introduced with phrases like "Our confounding parameters are" or "To control for". Additionally, some parameters were not described as confounding, but authors only described that they measured it. This unsystematic way makes it difficult fully understand an experimental design.

4.3.2 Measuring and Controlling for Confounding Parameters

There are various ways to measure and control the influence of a confounding parameter.[3] For example, we found several different ways for measuring and controlling for programming experience: It was kept constant by recruiting only students or creating two groups with comparable level of programming experience. To create comparable groups, researchers had to measure programming experience, which they realized with different indicators, such as the years participants have been programming, participants' level of education (e.g., undergraduate vs. graduate student), self estimation, or supervisor estimation. Often, authors wrote that they controlled for a parameter, but did not specify how.

The different means of measuring and controlling for confounding parameters can make the comparison of experiments difficult. For example, when we try to compare programming experience based on years participants have been programming, and based on the level of education, it is likely that we compare different things, because an undergraduate student may have been programming for several years, whereas a graduate student may have started programming when starting to study. Comparing experiments gets even more problematic when we do not know how a parameter was controlled for. Thus, researchers might not be able to fully understand and replicate an experiment.

[3] In Section 4.5, we discuss control techniques and parameters in detail. Here, we give only an example.

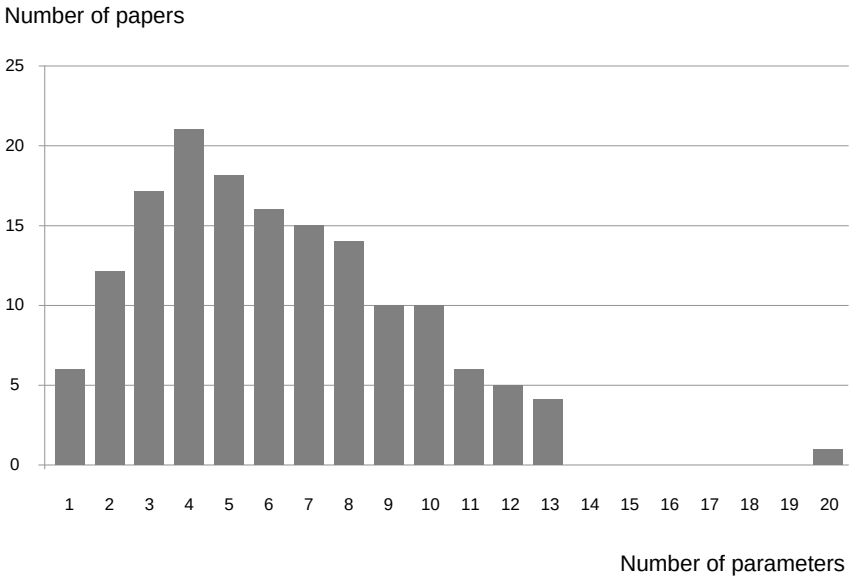


Figure 4.2: *Number of parameters mentioned per paper.*

4.3.3 Number of Confounding Parameters

In our survey, we found 39 confounding parameters. However, only a fraction is mentioned in each paper. In Figure 4.2, we give an overview of how many papers mentioned how many parameters. Most of the papers mentioned 10 or less parameters; one paper named 20 parameters. We believe that most authors controlled more parameters than they actually described, but that space restrictions prohibited mentioning each parameter and how it was controlled. However, knowing whether and how researchers managed confounding parameters helps to evaluate the soundness of an experiment.

To summarize, there is effort to control for confounding parameters. However, reporting this effort is too unsystematic, so it is difficult to evaluate the soundness of an experimental design. To address the identified problems, we give recommendations in Section 4.6.

4.4 Background: Techniques to Control for Confounding Parameters

Experimentation in psychology has a long history [Wundt, 1874]. Hence, all control techniques are based on psychological research and have proven useful in numerous experiments. Since we give recommendations how to control for confounding parameters, we present typical control techniques in this section, which are the following:

- Randomization
- Matching

- Keep confounding parameter constant
- Use confounding parameter as independent variable
- Analyze the influence of confounding parameters on result

For better illustration, we describe the control techniques with the confounding parameter *programming experience* as example (we go into more detail in Section 4.5.1.2). For each technique, we first explain it, and then discuss advantages and disadvantages regarding the criteria sample size, measurement requirement, effort of applying it, and generalizability of results.

4.4.1 Randomization

Using randomization, a confounding parameter is randomly assigned to experimental groups. This way, the influence of confounding parameters is spread evenly across experimental groups and is comparable in all groups [Goodwin, 1999]. For example, to create two comparable groups regarding programming experience, we toss a coin to assign all participants to groups. Since the group assignment is random, there is no systematic bias. That is, the coin toss does not assign more experienced participants to one group, and less experienced participants to another group. Hence, both groups are comparable, or homogeneous, regarding programming experience.

For randomization to be effective, we need a large enough sample size. Otherwise, chances of considerably different group sizes or heterogeneous groups are too high. Unfortunately, large cannot be defined as a fixed number. Assigning 30 participants to two experimental groups seems reasonably large, but assigning 30 participants to six experimental groups may be too small to ensure homogeneous groups. Thus, the more experimental groups there are, the more participants we need. In our own experience and discussions with other researchers, we found that five participants per group are too few, but ten seem to be acceptable. The benefit of randomization is that it does not require measuring a parameter and that applying it requires almost no effort. However, the generalizability of results depends on our sample: If we only recruit novices programmers, our results are only applicable for novice programmers; if we recruit participants with all levels of programming experience, our results are valid for all levels.

Thus, randomization is the most convenient way to control for a confounding parameter, because it does not require measurement of a parameter.

4.4.2 Matching

With matching, we measure a confounding parameter and assign participants to experimental groups, such that both groups have about the same size and same level regarding the confounding parameter [Goodwin, 1999]. To illustrate matching, we show fictional values for programming experience of participants in Table 4.3. The participants are ordered according to the quantified programming-experience value. Now, we assign Participant 5 to Group A, Participant 7 to Group B, Participant 1 to Group B, and Participant 10 to Group A again. We repeat this process until all participants are assigned to groups.

Participant	Value	Group
5	67	A
7	63	B
1	62	B
10	59	A
8	57	A
6	57	B
3	53	B
2	50	A
9	45	A
4	43	B

Table 4.3: *Fictional programming-experience values and according group assignments.*

This is also called as the odd-even-even-odd principle. We can also combine randomization and matching, such that we randomly assign blocks of participants (e.g., the first four, the second four, and the last two) to groups A and B.

The benefit of matching is that a smaller sample size suffices, because we control how we create groups: We can assign four participants to two groups, such that both groups have the same level of experience. However, the drawback is that we have to measure a confounding parameter, which increases the effort of matching compared to randomization. The more difficult a parameter is to measure, the higher the effort of applying matching. For example, for intelligence, there are a number of definitions and different tests [Jäger et al., 1997; Raven, 1936; Wechsler, 1950], which take time to conduct (e.g., 3h for the BIS [Jäger et al., 1997]) and a suitable one has to be chosen. Additionally, not for every parameter a suitable test exists. For example, there is no accepted way to measure programming experience. Thus, depending on the parameter, the effort for matching increases. The generalizability depends on the selected sample, like for randomization.

4.4.3 Keep Confounding Parameter Constant

When keeping a confounding parameter constant, we have exactly one level of this parameter in our experimental design (or at least exclude outliers) [Markgraf et al., 2001]. For example, to keep programming experience constant, we can measure programming experience and recruit participants only with a certain value. Alternatively, we can recruit participants from a population of which we know that a parameter has only one level. For instance, when we recruit freshmen, we can assume that their programming experience is roughly at one low, comparable level. There might be some students who started to program earlier, but we can exclude them by requiring that participants did not start programming before they started to study.

Like for matching, smaller samples suffice when keeping a confounding parameter constant. Whether a parameter needs to be measured depends on the parameter: If we recruit only freshmen, we can assume a low level of programming experience. Hence, if we minimize the effort of measuring a parameter, keeping it constant does not require

more effort than applying randomization. As a drawback, generalizability is reduced, because our results are only applicable to the selected level of the parameter, for example, low programming experience.

4.4.4 Use Confounding Parameter as Independent Variable

We can include a confounding parameter as independent variable in our experimental design [Markgraf et al., 2001]. This way, we can manipulate it and control its influence. For example, we can recruit participants with high and low programming experience, such that our results are applicable to people with high and low programming experience.

As a drawback, we need a large sample size, because we have more experimental groups (cf. Section 4.1). Regarding measurement, the same counts as for keeping a parameter constant: When we can apply heuristics to ensure certain levels (e.g., freshmen as novice programmers, senior software developers as expert programmers), the effort regarding measurement is low. A further drawback regarding the effort is that we need more complex research hypotheses and a more complex experimental design: If our initial design was one factorial, we now need a two-factorial design, leading to more complex analysis. However, the benefit is that generalizability is high, depending on the number of levels.

4.4.5 Analyze the Influence of Confounding Parameter on Result

When we cannot assign participants to experimental groups, we can analyze the influence of a confounding parameter after the experiment [Shadish et al., 2002]. This is often necessary when we recruit participants from companies, because we cannot assign participants to different companies. In this case, we can measure a parameter and analyze its influence on the result after conducting the experiment or excluding certain levels of a parameter (i.e., a variant of the technique to keep a parameter constant). For example, we can measure whether participants of Company A have a different level of programming experience than participants of Company B and then evaluate whether these differences influenced our result.

As benefit, a smaller sample size suffices to apply this technique. As a drawback, we need to measure a parameter, which may lead to high effort. The generalizability of results depends on the parameter and its values: If we find that only novice programmers participated, our results only count for novice programmers. If also experts participated, our results can be applied to both levels of programming experience.

4.4.6 Summary of Control Techniques

These five techniques are the most common control techniques. There are also other techniques that are specific for a confounding parameter. For example, when controlling for ordering effects (i.e., when the order of tasks or treatments influences results), we can use a one-factorial, within-subjects design with two groups (cf. Table 2.1c). We describe these techniques when we explain a corresponding parameter.

Technique	Sample size	Requires measurement	Effort to apply technique	Generalizability
Randomization	large	no	low	depends on sample
Matching	small	yes	depends on parameter	depends on sample
Constant	small	depends on parameter	depends on measurement	limited
Independent	large	depends on parameter	high	good
Analyzed afterwards	small	yes	depends on parameter	depends on parameter

Table 4.4: *Approximate benefits and drawbacks of control techniques.*

When we cannot apply any control technique, we should explain why and how that influences the validity of experiments. If the influence of a parameter is evidently negligible, we can even ignore it, but make this decision deliberately, not haphazardly.

In Table 4.4, we summarize the control techniques and their benefits and drawbacks. Note that we cannot give a general recommendation on which control technique to use, because that depends on the circumstances of the experiment. If the sample is small, we should avoid randomization and using a parameter as independent variable. If measuring a confounding parameter is too time consuming, we should use randomization. If we want to have high external validity, we should not keep a parameter constant. Thus, for each experiment and parameter, we need to carefully decide which control technique we use.

4.5 Identified Confounding Parameters

In this section, we present the confounding parameters we extracted. For a better overview, we divide confounding parameters into two categories: personal and experimental parameters. Personal parameters are related to the person of the participants, for example, programming experience or intelligence, and experimental parameters are related to the experimental setting, for example, tasks or source code. Personal parameters always influence the behavior of participants, whereas experimental parameters only influence the behavior of participants because they take part in an experiment. We found 16 personal and 23 experimental parameters. To have an understanding of the role of each parameter, we describe it and how it can influence the result, give an overview of how it was controlled and measured, and give recommendations how to manage it.

In addition to controlling for a confounding parameter, we often found in our review that authors *discussed* how a parameter could have influenced the result, or that authors just mentioned a parameter, but *did not specify* whether they controlled it or whether it had an influence. Additionally, some parameters can be *avoided*, for example, the influence

of participants by the experimenter toward confirming a hypothesis (i.e., the Rosenthal effect).

When describing how a parameter was controlled for in literature, we start with the five techniques we presented, then state when it was discussed, not specified, or avoided. Then, we mention other techniques authors used that are specific for an according parameter. We also denote the frequency of how often a technique was applied in round brackets following the technique, (like this). If only one paper used a technique, we cite the paper instead.

4.5.1 Personal Parameters

In Table 4.5, we summarize how often each of the 16 personal confounding parameters on program comprehension was considered.

Parameter	ESE	TOSEM	TSE	ICPC	ICSE	ESEM	FSE	Sum
Personal background (Section 4.5.1.1)								
Color blindness	0	0	0	1	0	0	0	**1**
Culture	0	0	2	1	0	0	0	**3**
Gender	0	0	3	4	1	0	0	**8**
Intelligence	0	0	2	4	1	0	0	7
Personal knowledge (Section 4.5.1.2)								
Ability	12	2	12	7	5	4	2	**44**
Domain knowledge	3	0	5	4	0	0	0	**12**
Education	8	1	6	15	8	3	0	**41**
Programming experience	24	2	25	23	22	11	5	**112**
Reading time	0	0	0	3	0	1	0	**4**
Personal circumstances (Section 4.5.1.3)								
Attitude toward study object	0	0	1	1	0	0	0	**2**
Familiarity with study object	19	2	17	10	10	12	6	**76**
Familiarity with tools	5	2	9	8	9	1	3	**37**
Fatigue	8	0	5	0	2	5	0	**20**
Motivation	12	0	10	7	3	2	0	**34**
Occupation	0	0	0	3	0	1	0	**4**
Treatment preference	0	0	0	3	1	2	0	**6**

Table 4.5: *Number of personal confounding parameters mentioned per journal/conference.*

For better overview, we present a summary of how each parameter was controlled in Table 4.6. We discuss each parameter and how it was controlled detail in this section. To have a better overview, we divide personal parameters further into the groups *personal background*, *personal knowledge*, and *personal circumstances*. Within each group, we discuss the parameters in alphabetic order.

Parameter	Ran.	Mat.	Con.	Ind.	Ana.	Dis.	Not sp.	Other
Personal background (Section 4.5.1.1)								
Color blindness	0	0	0	0	0	1	0	0
Culture	0	0	1	0	0	2	0	0
Gender	1	0	4	0	2	1	0	0
Intelligence	0	1	4	0	0	0	2	0
Personal knowledge (Section 4.5.1.2)								
Ability	7	8	4	6	12	2	6	0
Domain knowledge	1	0	2	1	1	4	3	0
Education	0	1	25	2	4	4	3	1
Programming experience	8	8	23	17	7	14	32	2
Reading time	0	0	1	0	2	1	0	0
Personal circumstances (Section 4.5.1.3)								
Attitude toward study object	0	0	0	0	2	0	0	0
Familiarity with study object	2	1	51	0	7	3	11	0
Familiarity with tools	1	0	32	0	1	0	3	0
Fatigue	0	7	1	0	2	2	0	7
Motivation	3	1	17	0	3	3	4	3
Occupation	0	0	2	0	2	0	0	0
Treatment preference	0	0	3	0	0	3	0	0

Ran.: Randomization; Mat.: Matching; Con.: Kept constant; Ind.: Used as independent variable; Ana.: Analyzed afterwards; Dis.: A parameter was discussed; Not sp.: A parameter was not specified; Other: Other control technique than mentioned

Table 4.6: *Control techniques for personal confounding parameters.*

4.5.1.1 Personal Background

Personal background describes parameters that have a fixed value for a participant, that is, with which participants are born and that hardly change during life time.

Color blindness Persons who are unable to perceive certain colors, for example, red and green, are referred to as color blind [Goldstein, 2002]. When colors play a role in an experiment, for example, when participants see source code with syntax highlighting or when the effectiveness of background colors is analyzed, color-blind participants might respond different or slower than other participants. Furthermore, they might be unable to solve a task if they cannot distinguish relevant colors.

Color blindness was considered in only one experiment, in which it was discussed as threat to validity [Jablonski and Hou, 2010].

To control for color blindness, randomization is not suitable, because only a small fraction of the population are color blind and it interacts with gender (i.e., females are less frequently color blind) [Goldstein, 2002]. For example, if we have 20 participants, maybe one or two are color blind, who might easily be assigned to the same group. Since it is easy to measure color blindness, either by asking participants or by applying the

Ishihara test [Ishihara, 1972], we recommend to measure it, if relevant. Then, we can choose how we deal with color-blind participants (e.g., excluding them, assigning them to an experimental group in which colors do not play a role).

Culture Culture refers to the origin of participants. This can affect the outcome, because different cultures (especially Western compared to Asian cultures) often have different ways to solve a problem.

In literature, we found three papers that mentioned culture. In two of them, the influence of different cultures was analyzed afterwards, and in one, culture was kept constant [Lui et al., 2008]. Although none of the other papers mentioned culture, we believe most of them held it constant, because they recruited students from one university (or even one class) or one company.

Since culture is easy to measure by asking participants to which culture they belong, we recommend to assess it and then decide what to do with participants. However, we have to be careful not to discriminate against a participant. For example, when evaluating the effectiveness of new teaching methods, it might be unethical to exclude participants because they have a different culture. In this case, we can also let participants complete the experiment and then exclude the data set from the analysis. However, we have to ensure that all experimental groups have a comparable size after exclusion.

Gender Gender of participants might influence program comprehension, for example, because boys were more encouraged than girls to play with computers. Thus, male participants may be more familiar with computers.

In literature, gender was kept constant (4), randomized [Vitharana and Ramamurthy, 2003], and analyzed afterwards (2). In one paper, gender was discussed as threat to validity [Ko and Uttl, 2003].

Regarding controlling for gender, the same arguments apply as for the culture of participants: It is easy to measure, but we should avoid discriminating against participants.

Intelligence[4] The ability to solve problems, memorize material (e.g., using working memory capacity), recognize complex relationships, or combinations thereof is referred to as intelligence [Jäger et al., 1997; Raven, 1936; Wechsler, 1950]. Intelligence can influence program comprehension, such that the higher problem-solving skills or memory skills participants have, the better they are able to understand source code.

In our literature survey, intelligence was kept constant (4), such that the memory capacity was not exceeded. To this end, material was either presented on paper to participants (so participants could look up information and did not have to memorize everything), or the number of items was small enough to avoid memory overload. In one paper, intelligence was analyzed afterwards (measured with an intelligence test) [Ko and Uttl, 2003]. In one other paper, a within-subjects design was used, such that the same group of participants received all experimental treatments [Lui et al., 2008]. In another

[4]There are voices that say intelligence is rather something learned than something inborn. Thus, we could also classify it as personal knowledge.

paper, intelligence was measured, but not specified how [Vitharana and Ramamurthy, 2003].

There are numerous tests to measure intelligence, but they are time consuming [Jäger et al., 1997; Raven, 1936; Wechsler, 1950]. Hence, we recommend randomization. However, when certain facets of intelligence, such as memory capacity, are assumed to have a significant influence on program comprehension, we recommend to measure that facets and apply a suitable control technique (e.g., using it as independent variable).

4.5.1.2 Personal Knowledge

Personal knowledge describes parameters that are influenced by learning and experience. These parameters change, but rather slowly over a period of weeks, months, or years.

Ability Ability as a general term describes skills or competence of participants. The more and higher ability participants have (e.g., regarding implementing code or using language constructs), the better they may comprehend source code. Unfortunately, authors only rarely specified what they mean with ability. Based on the descriptions in the papers, we believe that ability refers to how skilled participants are with the study object, such as writing code or UML modeling.

Authors used randomization (7), matching (8), kept it constant (4), used it as independent variable (6), and analyzed it afterwards (12). Additionally, authors discussed it (2) or did not specify how they controlled for ability (2). To measure ability, authors often used grades of courses participants were enrolled in.

Since ability is not clearly specified, we cannot give recommendations beyond rules of thumb. When it is easy to measure or has a considerable influence, we should measure it; otherwise, we should randomize it.

Domain knowledge The familiarity of participants with the domain of the study object, for example, databases, is called domain knowledge. It influences whether participants use top-down or bottom-up comprehension. Usually, top-down comprehension is faster than bottom-up comprehension, because developers can compare source code with what is in their memory and use beacons, plans, or concepts. With bottom-up comprehension, developers have to analyze each statement, which inherently takes more time.

In our survey, domain knowledge was randomized [Vitharana and Ramamurthy, 2003], kept constant (2), used as independent variable [O'Brien and Buckley, 2001], and analyzed afterwards [Ko and Uttl, 2003]. Furthermore, it was discussed (4) and not specified (3). To measure domain knowledge, author's asked participants or assumed familiarity based on the courses participants were enrolled in.

Since domain knowledge has a strong influence on program comprehension, we should usually avoid randomization. Instead, we should measure it, for example, by asking participants how familiar they are with relevant domains and exclude participants that are too familiar or unfamiliar or apply an according training.

Education The education of participants describes the topics participants learned during their studies, but not the status of participants' studies (e.g., whether participants are freshmen or graduate students). If students visited mostly programming courses, their skills are different from students who mostly visited database or graphical-user-interface courses, in which programming may not be the primary content.[5]

In literature, education was controlled with matching [Svahnberg and Wohlin, 2005], kept constant (25), used as independent variable (2), and analyzed afterwards (4). In one paper, participants with different education were treated as different samples [Thelin, 2004]. Furthermore, authors discussed its influence (4) or did not specify how they managed it (3). Often, participants of one course were recruited to keep it constant. In some other cases, authors asked participants the courses they completed.

To control for education, we can ask participants the courses they were enrolled in, but we cannot be sure what was taught in these courses. Instead, we recommend to ask what participants learned that might be relevant for the experiment. For example, if UML is relevant, we can ask whether they completed courses in which they learned UML. Additionally, we can use randomization if we are not sure what might be relevant for an experiment.

Familiarity with study object/tools Participants have different levels of experience with the evaluated concepts or tools, such as Oracle or Eclipse, which is referred to as familiarity with study object/tools. Familiarity with study object appears to be the same as domain knowledge. However, looking closer, they slightly differ, such that domain knowledge describes the domain of a study object (e.g., databases), and familiarity with study object the object itself (e.g., Oracle as one concrete database system). If participants are familiar with the study object or tools, then they do not need as much cognitive resources as unfamiliar participants. Thus, they might perform better. We summarize familiarity with study object and tools, because they are closely related. Numbers for both variables are presented together and separated with a slash (<numbers for study object>/<numbers for tools>).

In our survey, familiarity with study object/tools was controlled with randomization (2/[Sarma et al., 2008]), matching ([Itkonen et al., 2007]/0), kept constant (51/32), and analyzed afterwards (7/[Walenstein, 2003]). Furthermore, authors discussed its influence (3/0) or did not specify how they controlled it (11/3). Often, participants were trained to assure a comparable level of familiarity. To measure familiarity, participants were asked how familiar they are or a pretest was conducted.

Since familiarity with study object/tools is easy to measure, we recommend to assess it, for example, by asking participants or conducting a pretest. We can also reliably keep it constant by recruiting unfamiliar participants or train participants to have the same level of familiarity.

Programming experience Programming experience describes the experience participants had so far with writing and understanding source code. The more source code participants have seen and implemented, the better they can adapt to comprehending source

[5] Of course, the specific contents of courses depend on the country and specific university.

code, and the higher the chance is that they will be more efficient in experiments [McConnell, 2011; Sackman et al., 1968].

Most of the papers of our survey took programming experience into account (112). Authors used randomization (8), matching (8), kept it constant (23), used it as independent variable (19), and analyzed it afterwards (7). Additionally, authors discussed the influence of programming experience (14) or did not specify how they managed it (32). To measure programming experience, authors often assessed the years participants have been programming, the status of education, and self estimation.

Programming experience has an important influence in program-comprehension experiments. Hence, we should not use randomization, but measure programming experience and apply a suitable control technique (e.g., matching, using it as independent variable). To this end, we developed a programming-experience questionnaire based on self estimation (cf. Chapter 5).

Reading time Participants may differ in their reading time, that is, how fast they can read. The faster they are, the more they can read in a given time interval. Consequently, they may be faster in understanding source code.

In literature, reading time was kept constant [Sharif and Maletic, 2010], analyzed afterwards (2), and discussed as threat to validity [Xie et al., 2007]. To measure it, authors always used an eye tracker.

Reading source code is only a minor part in the comprehension process, because statements are short compared to normal text and because source code is often formatted to improve readability. Thus, we recommend to use randomization or ignore it, unless authors believe that reading time has a significant influence on the result. To measure reading time without an eye tracker, we can let participants read a text and stop the time and confirm that participants read the text with comprehension questions.

4.5.1.3 Personal Circumstances

Parameters in this category describe how participants feel at the time of the experiment. These parameters can change rapidly, that is, within minutes. For these parameters, matching is not feasible, because they can change during the experiment.

Treatment preference Treatment preference refers to whether participants prefer a certain treatment, such as a new tool. This can affect performance, such that participants who do not like a new tool are not willing to work with it as they are supposed to.

In our survey, treatment preference was kept constant (3), analyzed afterwards (2), and discussed (3). To measure it, authors asked participants which treatment they prefer. To keep it constant, authors of one paper designed a context-neutral task and compared the performance with other treatment tasks.

To control for treatment preference, we recommend to measure it (e.g., by self estimation of participants) and analyze its influence afterwards, with which we had good experience in our experiments. Furthermore, experimenters should take care not to influence participants to prefer one treatment over the other.

Fatigue If participants get fatigued, they lose concentration. This occurs especially in long experiments and could affect performance of participants, such that the error rate increases toward the end of the experiment.

In literature, fatigue was controlled for with matching (7), kept constant [Ko et al., 2006], and analyzed afterwards (2). Furthermore, session duration was short enough to avoid fatigue (7). In two papers, it was discussed. To measure it, authors used self estimation or the performance of participants (e.g., whether the error rate increases). For matching, authors assessed fatigue before the experiment. To keep it constant, authors deleted data indicating poor performance from the analysis.

The best way to control for fatigue is to avoid it by having short enough sessions. Humans typically can work concentrated for about 90 minutes [Jensen, 1998]. After that, attention decreases. In our experience, sessions that last longer than 2 hours, including introduction and debriefing, are too exhausting. If sessions need to be longer, we can randomize the order of tasks (i.e., use randomization), so that not the same tasks are completed at the end of the experiment, or split sessions.

Motivation Participants may have different levels of motivation to participate in the experiment. If participants are not motivated, it may affect their performance negatively [Mook, 1996].

In our survey, motivation was controlled for using randomization [Knodel et al., 2008], kept constant (17), or analyzed afterwards (3). Furthermore, authors used different task orders [Baniassad et al., 2003], short enough session duration [Sfetsos et al., 2009], or included the performance in the experiment as part of a participant's grade to assure high motivation [Sharif and Maletic, 2009]. In another paper, the tasks built up on each other, so participants were motivated to perform well for a task [Biffl and Halling, 2003]. Additionally, the influence of motivation was discussed (3) or not specified (4). To measure motivation, authors asked participants to estimate their motivation. To keep motivation constant, authors offered rewards for the best-performing participant(s) to motivate participants to show their best performance.

To control for motivation, we recommend to assure high motivation. To this end, we can recruit participants on a voluntary basis, not make participation mandatory to complete a course, because participants who volunteer are typically motivated. To further increase motivation, good performance can be rewarded. Additionally, we can ask participants how motivated they were and analyze its influence afterwards.

4.5.2 Experimental Parameters

Experimental parameters are related to the experiment and its setting. We found 23 parameters, which we summarize in Table 4.7. In Table 4.8, we give a summary of how each parameter was controlled. For a better overview, we divide experimental parameters into four categories: *subject related*, *technical*, *context related*, and *study object related*.

Parameter	ESE	TOSEM	TSE	ICPC	ICSE	ESEM	FSE	Sum
Subject related (Section 4.5.2.1)								
Evaluation apprehension	0	0	1	1	0	0	0	2
Hawthorne effect	9	1	3	2	2	5	0	22
Process conformance	15	1	10	4	5	8	1	44
Study-object coverage	2	0	0	1	0	1	0	4
Ties to persistent memory	0	0	0	1	0	0	0	1
Time pressure	7	0	4	1	0	2	0	14
Visual effort	0	0	0	1	0	0	0	1
Technical (Section 4.5.2.2)								
Data consistency	0	0	1	0	0	0	0	1
Instrumentation	8	0	8	2	0	1	0	19
Mono-method bias	2	0	1	0	0	0	0	4
Mono-operation bias	2	0	1	1	0	0	0	3
Technical problems	0	0	0	2	0	2	0	2
Context related (Section 4.5.2.3)								
Learning effects	15	0	14	16	7	9	4	65
Mortality	0	0	1	0	0	0	1	2
Operationalization of study object	1	1	0	0	0	0	0	2
Ordering	5	0	7	8	2	2	3	27
Rosenthal	10	1	2	3	3	5	0	24
Selection	11	1	6	1	2	2	1	24
Study-object related (Section 4.5.2.4)								
Content of study object	5	1	1	9	0	2	1	19
Language	7	2	14	23	13	7	6	72
Layout of study object	4	0	2	7	0	3	1	17
Size of study object	14	1	19	15	9	6	3	67
Tasks	6	0	6	14	5	4	2	37

Table 4.7: *Experimental confounding parameters.*

4.5.2.1 Subject-Related parameters

Subject-related parameters are caused by participants and only emerge because participants take part in an experiment. In this way, they differ from personal parameters, which are always present.

Evaluation apprehension Evaluation apprehension refers to the fear of being evaluated. This may bias responses of participants toward what they perceive as better. For example, participants could judge tasks easier than they actually think to hide from the experimenter that they had difficulties. Another problem might be that participants cannot show their best performance, because they feel frightened.

Parameter	Ran.	Mat.	Con.	Ind.	Ana.	Dis.	Not sp.	Other
Subject related (Section 4.5.2.1)								
Evaluation apprehension	0	0	0	0	0	0	0	2
Hawthorne effect	0	0	1	0	0	6	0	15
Process conformance	0	3	4	0	9	6	0	21
Study-object coverage	0	0	3	1	0	0	0	0
Ties to persistent memory	0	0	0	1	0	0	0	0
Time pressure	0	0	6	0	2	4	0	2
Visual effort	0	0	0	1	0	0	0	0
Technical (Section 4.5.2.2)								
Data consistency c	0	0	0	0	0	0	0	1
Instrumentation	0	0	5	0	1	11	0	2
Mono-method bias	0	0	0	0	0	0	0	3
Mono-operation bias	0	0	0	0	0	1	0	2
Technical problems	0	0	4	0	0	0	0	0
Context related (Section 4.5.2.3)								
Learning effects	5	21	5	1	12	7	2	4
Mortality	0	0	0	0	0	2	0	0
Operationalization of study object	0	0	0	0	0	1	0	0
Ordering	5	13	1	0	3	5	0	0
Rosenthal	0	0	0	0	0	6	0	18
Selection	3	4	1	0	2	11	1	2
Study-object related (Section 4.5.2.4)								
Content of study object	2	1	7	0	1	3	0	7
Language	1	0	48	0	2	7	6	0
Layout of study object	0	1	3	6	1	2	1	2
Size of study object	0	1	1	1	1	4	2	59
Tasks	2	8	2	1	4	10	2	5

Ran.: Randomization; Mat.: Matching; Con.: Kept constant; Ind.: Used as independent variable; Ana.: Analyzed afterwards; Dis.: A parameter was discussed; Not sp.: A parameter was not specified; Other: Other control technique than mentioned

Table 4.8: *Control techniques for experimental confounding parameters.*

In our survey, evaluation apprehension was avoided by anonymizing the data [Mohan and Gold, 2004] and by assuring participants that their performance does not affect their grade for the course [Do et al., 2010]. However, we believe that considerably more authors anonymized the data, but did not mention it, because anonymization is often used to ensure that participants have no disadvantage from participation.

To control for evaluation apprehension, we recommend to assure anonymity of participants. Additionally, we can encourage participants to answer honestly by clarifying that only honest answers are of value for us. Furthermore, with randomization, the effect of evaluation apprehension should be homogeneous between groups.

Hawthorne effect The Hawthorne effect is closely related to evaluation apprehension. It describes that participants behave differently in experiments, because they are being observed [Roethlisberger, 1939]. Like evaluation apprehension, we may observe different behavior compared to observing participants in their real environment, thus threatening validity. The difference to evaluation apprehension is that participants do not change their behavior because they are afraid of being evaluated, but because they are observed.

In our survey, the influence of the Hawthorne effect was discussed (6). Furthermore, to avoid it, authors did not reveal their hypotheses to participants (15) or used a context-neutral task [Ellis et al., 2007]. It was measured once by comparing the performance in a context-neutral task to performance in treatment tasks.

To control for the Hawthorne effect, the best way is to avoid it by not letting participants know that they take part in an experiment. However, this can raise ethical problems and must be discussed with an ethics committee to ensure fair treatment of all participants. In most cases, not telling participants that they take part in an experiment is impossible, because we need to instruct participants. Thus, we recommend not to reveal hypotheses to participants, such that they cannot bias their performance in favor of or against our hypotheses. After the experiment, we recommend to reveal participants the hypotheses, so that they understand what their data is used for.

Process conformance If participants follow the instructions, they maintain process conformance. If participants deviate from their instructions, for example, searching the internet for solutions or giving subsequent participants information about the experiment, the results may be biased.

In our survey, process conformance was kept constant (4) and analyzed afterwards (9). Furthermore, participants were observed to assure process conformance (15), were encouraged to stick to the instructions (2), or signed a consent not to disclose any information (3). In one experiment with several sessions, participants were not allowed to take any material home [Briand et al., 2005], and three experiments used different tasks for participants seated next to each other. Additionally, the influence of process conformance was discussed (6) or not specified [Gupta and Jalote, 2007]. To measure process conformance, authors often asked the participants how well they followed the instructions. To keep it constant, authors deleted data from participants who deviated from the instructions.

To control for process conformance, we recommend to tell participants exactly what they have to do and observe that they are following these instructions. If observation is not possible (e.g., when participants are allowed to work at home), we should asked participants how well they followed the protocol and analyze the effect of deviations afterwards. This may threaten internal validity, but increases external validity.

Study-object coverage Study-object coverage describes how much of the study object was covered by participants. If a participant solved half as much tasks as another participant, the results could mean something different, because, for example, the slower participant was more thorough.

In our survey, study-object coverage was kept constant (3) or used as independent variable [Mouchawrab et al., 2007]. To keep it constant, authors excluded participants who did not complete all tasks.

Since we can easily determine study-object coverage, we recommend to assess it and apply suitable control techniques (e.g., keeping it constant by excluding participants).

Ties to persistent memory If experimental material has links to persistent (or long-term) memory of participants, it has ties to persistent memory. If source code has no ties to persistent memory and working memory becomes flooded (e.g., because of long variable names or long method calls), comprehension may be impaired.

Ties to persistent memory was relevant in only one study [Binkley et al., 2008]. It was measured in terms of the usage of identifiers: Identifiers often used in packages were assumed to have ties to persistent memory, whereas program or domain identifiers have no ties to persistent memory.

The influence of ties to persistent memory is not always important. Only in comprehension experiments, in which the memory capacity plays a crucial role, we should consider it. To measure it, we can categorize material into having ties or not having ties to persistent memory, for example, based on participants' domain knowledge.

Time pressure If participants feel rushed to complete the experiment in a given time frame, they experience time pressure. This can bias the performance, such that participants make more errors when time is running out.

Time pressure was kept constant (6) and analyzed afterwards (2). Furthermore, authors used a realistic setting [Arisholm et al., 2007] and encouraged participants to work as fast as possible [Hannay et al., 2010]. In four papers, it was discussed as confounding parameter. To measure it, authors asked participants whether they experienced time pressure. To keep it constant, authors did not set a time limit for a task.

To control for time pressure, we recommend to keep it constant. Since it is difficult to create the same time pressure for all participants, we should set no time limit for an experiment. If a time limit is unavoidable, we should analyze the influence of time pressure afterwards (e.g., by asking participants).

Visual effort Visual effort describes the number and length of eye movements to find the correct answer. The more effort a task has, the longer it takes to find the correct answer.

Visual effort was relevant in only one experiment [Sharif and Maletic, 2010]. It was controlled by analyzing the eye movements of participants recorded with an eye tracker.

Visual effort cannot be measured upfront. The best way to control for visual effort is to use an eye tracker. If it is not important, we can ignore it.

4.5.2.2 Technical Parameters

Technical parameters are related to the experimental set up, such as the tools that are used.

Instrumentation The instruments used in the experiment, such as questionnaires, tasks, or eye trackers are referred to as instrumentation. This can influence the result, especially when instruments are not carefully designed or unusual for participants.

In literature, instrumentation was kept constant (5) and analyzed afterwards [Otero and Dolado, 2002]. Furthermore, authors conducted pilot studies [Güleşir et al., 2009] and evaluated the instruments based on design principles [Dzidek et al., 2008]. Additionally, the influence of instrumentation was discussed (11). To keep it constant, authors often used comparable systems and tasks (without further specifying how they ensured comparability).

To control for instrumentation effects, we need to carefully choose and design the instruments we use. For example, when we use a questionnaire, we should use one that is validated to measure what it intends to measure. If there is no questionnaire, we should carefully design our own by consulting literature and/or experts on that topic. Furthermore, we recommend to conduct pilot tests to ensure that the instruments are appropriate and applicable.

Data consistency Data consistency refers to how consistent the data of the experiment are. For example, when paper-based answers of participants are digitalized, answers can be forgotten or transferred wrong. Inconsistent data can bias the results, because we analyze something different than we measured.

In our survey, only one paper controlled for data consistency [Biffl and Halling, 2003]. To this end, data digitalized from paper to computer were checked by two independent reviewers.

To avoid inconsistent data, we recommend to check data, especially when transcribing paper-based data to a digital form.

Mono-method bias If we use only one measure to assess a variable, we have a mono-method bias (e.g., only response time of programming tasks to measure program comprehension). If that measure is badly chosen, the results are biased. For example, when participants wanted to finish a task independent of correctness, response time is not a suitable indicator.

In three papers, we found that authors explicitly mentioned and controlled for mono-method bias by using different measures for comprehension. However, we believe that more authors controlled for mono-method bias, but did not mention it.

To control for mono-method bias, we recommend to always use at least two measures for comprehension. For example, we can use correctness and response time of tasks and/or an efficiency measure as combination of both.

Mono-operation bias Mono-operation bias is related to mono-method bias; it refers to an underrepresentation of the evaluated construct, for example, when we use only one task to measure comprehension. If that task is not representative, then our results may be biased. For example, a task can be designed such that it confirms a hypothesis.

In our survey, mono-operation bias was controlled for by using different tasks [Torchiano, 2004] or a representative task (2). Furthermore, it was discussed [Porras and Guéhéneuc, 2010].

To avoid mono-operation bias, we should include several tasks that are representative for the evaluated construct. To ensure representativeness, we can consult literature or experts in the domain.

Technical problems During experiments, technical problems can occur, for example, a computer crash or missing questionnaires for participants. This may bias the results, such that participants have to repeat a task or that we miss answers of participants.

In literature, four papers kept the influence of technical problems constant, such that they excluded participants from the analysis when a problem occurred.

The best way to manage technical problems is to avoid them where possible. Hence, we recommend to always have some extra questionnaires, papers, or pens. If we cannot avoid them, such as computer crashes, we should analyze the effect on the result and, if necessary, exclude data from the analysis.

4.5.2.3 Context-Related Parameters

Context-related parameters are typical problems that occur in nearly all experiments, such as participants that drop out or learn from experimental tasks.

Learning effects Typically, participants learn during the session of an experiment, which is called learning effects. This is especially problematic in within-subject designs, because participants could learn from the first application of a treatment.

In our survey, learning effects were controlled for by randomization (5), matching (22), keeping it constant (5), using it as independent variable [Mouchawrab et al., 2007], and analyzing it afterwards (12). Furthermore, it was discussed (7), or not specified (2). To measure it, authors often compared the performance of subsequent tasks. To keep it constant, authors conducted a training before the experiment, such that participants learned mostly during the training, not during the experiment.

To control for learning effects, we can conduct a training before the experiment, so that the effect is minimized. Furthermore, we recommend to use different tasks in different sessions, so that participants cannot learn too much from a session. Additionally, we can use matching by applying a between-subjects design to avoid learning effects or measure them.

Mortality Mortality occurs when participants drop out from an experiment. This is especially a problem in multi-session experiments, where participants have to return for sessions. Mortality may influence the result, because participants may not drop out randomly, but, for example, only low-skilled participants because of frustration caused by perceived difficulty of the experiment.

Only two papers discussed the effect of mortality on their result.

To control for mortality, we should avoid it where possible, for example, by splitting the experiment in as few sessions as possible. If we need multiple sessions, for example, to avoid fatigue due to too long sessions, we should encourage participants to return, for example, by rewarding participants for completing all sessions.

Operationalization of study object If the study object is not suitably operationalized (i.e., its measurement is unsuitable), our results may be biased. For example, to measure program comprehension, we can use the correctness of solutions to tasks, but the number of opened files is not a good indicator. If we use an inappropriate measure, then not the study object, but something else is measured.

In our survey, we found that the operationalization of study object was discussed (2). However, we found that authors typically carefully operationalized the study object without explicitly discussing whether their operationalization is suitable.

To avoid bias due to inappropriate operationalization, we recommend to consult literature and/or experts to find the optimal operationalization of the study object.

Ordering The order in which tasks or experimental treatments are applied may influence the result, which is referred to as ordering effect. If the solution of one task automatically leads to the solution of subsequent tasks, but not the other way around, a different order of these tasks leads to different results.

In our survey, ordering was controlled for by randomization (5), matching (13), kept constant [Vokáč et al., 2004], and analyzed afterwards (3). In five other papers, the effect of ordering was discussed. In none of the papers, authors described how they measured the influence of the ordering effect.

To control for ordering effects, we recommend to randomize the order or use matching by applying an appropriate experimental design (e.g., between-subjects). If we cannot randomize the order (e.g., when tasks built up on each other), we should analyze whether and how the order influenced our result.

Rosenthal effect The Rosenthal effect refers to that the experimenter influences consciously or subconsciously the behavior of participants [Rosenthal and Jacobson, 1966]. This can influence the result, especially when we assess participants' opinion about a new technique or tool, such that participants rate it more positive.

In literature, none of the standard control techniques was used. Instead, authors discussed the effect (6), were careful not to bias participants (4), were objective (i.e., they did not develop the technique under evaluation) (4), or used standardized instructions (6). Furthermore, the objectivity of the material was evaluated by several reviewers (3). The Rosenthal effect is difficult to measure. Instead, authors used different techniques to avoid it.

To avoid the Rosenthal effect, we recommend to use a standardized, neutral, well-defined set of instructions for the experimenter and ensure that these instructions are followed.

Selection If the selection of participants for an experiment is not representative, bias may occur, such that the conclusions are not applicable to the intended population. For example, if we select students as participants, we cannot apply results to programming experts. Selection influences some of the personal confounding parameters (e.g., programming experience, motivation). Thus, if we avoid selection bias, we reduce the bias due to some personal parameters. However, avoiding selection bias is not sufficient to control for the according personal parameters.

In our survey, selection was controlled for by using randomization (3), matching [Dunsmore et al., 2002], kept constant [Porras and Guéhéneuc, 2010], and analyzed afterwards (2). Furthermore, authors used block designs (3), special analysis techniques (2), discussed it (11), or did not specify how they controlled it [Thelin et al., 2004]. Selection bias is difficult to measure. Thus, authors used techniques to avoid it.

To control for selection bias, we have to ensure to select a representative sample. The best way to do so is to randomly recruit participants from the intended population. However, this is not feasible in most cases (e.g., we cannot recruit all students who start to learn Java from all over the world, because we lack the resources). Typically, we recruit participants from one university or company. Hence, our results are only applicable for similar universities or companies. Thus, we must be aware of that limitation and should communicate that as threat to validity, even though it limits external validity of an experiment.

4.5.2.4 Study-Object-Related Parameters

Study-object-related parameters describe properties of the study object, such as its size.

Content of study object Experimenters should ensure that the content of the study object, that is, what source code or models are about, is comparable between different treatments. Otherwise, it may bias the results, such that one content is more difficult to comprehend. For example, when we compare the comprehensibility of object-oriented with imperative programming based on two programs, we need to make sure that both programs differ only in the paradigm, not the language or the functionality they are implementing.

In literature, the content of study object was controlled for by using randomization (2), kept constant (7), and analyzed afterwards [Vokáč et al., 2004]. Furthermore, authors used a realistic setting (6), compared realistic to non-realistic tasks [Binkley et al., 2009], and let two reviewers evaluate that the content is comparable [De Lucia et al., 2010]. In two papers, authors discussed how the content of study object may have influenced the result. The effect of the content of study object is difficult to measure. Instead, authors relied on their own or expert estimation to ensure that there is no bias caused by inappropriate content.

To control for content of study object, we recommend to use the same content or standardized material whenever possible. In other cases, we should consult at least two reviewers who evaluate whether the content of the study object is comparable.

Language Language refers to the underlying programming language of the experiment. We could also summarize language under *familiarity with study object* or *content of study object*, but we decided to keep it separate, because we work with program comprehension, for which the underlying programming language has an important influence. If participants work with an unfamiliar programming language, their performance is different than if they work with a familiar language, because they need additional cognitive resources for understanding the unfamiliar language.

In our survey, we found that authors used randomization [Vitharana and Ramamurthy, 2003], kept the language constant (48), and analyzed its influence afterwards (2). Furthermore, authors discussed the influence of language (7) or did not specify how they controlled for it (6). To measure language, authors most often used self estimation of participants. To keep it constant, authors recruited participants who were familiar with a language.

To control for language, we recommend to keep its influence constant. This reduces external validity. However, if we use a common language, such as Java, results still count for a large number of cases. To measure it, we can ask participants to estimate their experience or conduct a pretest.

Layout of study object The study object can be presented using different layouts. For example, source code can be formatted according to different guidelines or not formatted consistently, or different UML models can have different layouts. This may influence the comprehension of participants, because they have to get used to the layouts.

In our survey, the layout of the study object was kept constant (3), used as independent variable (6), or analyzed afterwards [Ng et al., 2006]. Furthermore, authors used a real-world setting [Matthijssen et al., 2010] or a specific experimental design [Sajaniemi and Prieto, 2005]. In two papers, authors discussed the influence of the layout of study object, and in one paper, authors did not specify how they controlled it [Qattous et al., 2010]. To ensure comparable layouts, authors used their common sense to avoid influences due to differences in the layout.

To control for layout of study object, we recommend to keep the layout constant across different treatments. Furthermore, we should use a realistic layout, such as standard formatting styles for source code. This way, we do not threaten external validity.

Size of study object Study objects can differ in their size, for example, the number of lines of source code or the number of elements in a UML model. The larger an object is, the more time participants need to work with it. If objects of different treatments differ in their size, the results of the experiment are also influenced by different sizes, not different treatments.

In literature, we found that authors used matching [Kuzniarz et al., 2004], kept it constant [Quante, 2008], used it as independent variable [Lemon et al., 2007], and analyzed it afterwards [Vitharana et al., 2003]. Furthermore, authors discussed its influence (4), mentioned the size (59), or did not specify how they managed it (2). To measure the size of the study object, authors used lines of code, number of files/classes, or number of elements in a UML model.

To control for size of study object, we recommend to keep it constant in different treatments. If that is not possible, we should make sure that the different size of study objects is as similar as possible, such that it does not influence the outcome of the experiment.

Task Task describes how tasks can differ, for example, in difficulty or complexity. If the difficulty of tasks for different treatments is not the same, then the difficulty would bias the results.

In our survey, we found that authors used randomization (2), matching (8), kept it constant (2), used it as independent variable [Dias-Neto and Travassos, 2009], or analyzed it afterwards (4). Furthermore, authors used realistic tasks (5), discussed its influence (10), or did not specify how they managed it (2). The influence of task cannot be measured directly. Instead, authors used their own estimation to ensure comparable tasks.

To avoid influence due to tasks, we recommend to carefully design tasks with the help of literature and/or experts. If possible, we should use standardized tasks.

4.5.3 Concluding Remarks about Confounding Parameters in Literature

To summarize, there are numerous confounding parameters for program comprehension. There is no general control technique to control all parameters, but depending on the circumstances of the experiment, the most suitable control technique(s) needs to be chosen. In this section, we gave recommendations for typical program-comprehension experiments as we encountered them in our literature survey.

The categorization we used here serves as overview and should not be seen as absolute. For example, intelligence can be defined as something that is learned rather than inborn. Thus, we could also categorize intelligence as personal knowledge.

Furthermore, it might seem unsettling that some parameters, such as mono-operation bias, are considered in only few studies. However, we believe that authors controlled for parameters more often than we found in our survey, but did neither explicitly nor implicitly mentioned it. Thus, the actual number of how often confounding parameters are controlled should be higher than we found.

Additionally, some parameters appear very similar, such as domain knowledge and familiarity with study object. However, instead of describing similar parameters as one, we kept them separate to have a broad overview and enable experimenters to look at variables from different points of views. This way, we hope that experimenters can better decide whether and how a parameter influences the outcome of an experiment.

4.6 Recommendation

In this section, we give recommendations on how to manage confounding parameters. In short, they are the following:

- Decide whether a confounding parameter is relevant for an experiment
- Select an appropriate control technique

- Describe all confounding parameters explicitly in the design part of a report
- Report whether and how confounding parameters are measured and controlled for

First, we have to decide whether a parameter has an influence in an experiment. Unfortunately, we cannot give general recommendations on which parameter is relevant, because that depends on the circumstances of the experiment. Thus, we recommend to check the list of parameters we identified and provide in Appendix 10 when designing an experiment. By systematically considering each parameter, researchers can evaluate whether it is relevant for an experiment or not. Since this step significantly influences the validity of an experiment, we need to put great care in the decision whether a parameter is relevant or not. Thus, we recommend to discuss it in a group of at least two experimenters who are familiar with the study object, its domain, and experiments in general. This way, we assure to have several relevant opinions about whether a parameter is relevant. If we cannot reach an agreement, we recommend to treat a parameter as confounding to minimize possible bias on our results. If we decide that a parameter has no influence, we can ignore it.

Second, the same great care should be put into the selection of an appropriate control technique. If we decide that a confounding parameter is relevant, but choose an inappropriate control technique, our results would be biased. To support the decision for a suitable control technique, we gave an overview of how other researchers controlled a parameter and gave general recommendations for each parameter.

Third, experimenters should describe all confounding parameters in the design part of a report and should explicitly define it as a confounding parameter. For example, Jedlitschka and others suggest to explain hypotheses and variables in one section as part of the experiment planning [Jedlitschka et al., 2008]. We recommend to list confounding parameters also in this section. This way, authors enable other researchers to easily perceive which confounding parameters authors considered as relevant for an experiment.

Finally, we should report whether and how we controlled for a confounding parameter. To this end, we recommend to use a pattern similar to the one described in Table 4.9. We illustrate this pattern with the parameters programming experience, the Rosenthal effect, and ties to persistent memory. We mention each parameter, provide an abbreviation to reduce the space we need to refer to it, describe the control technique(s) and the explain why we applied it, and describe how we measured it or ensured that it is sufficiently controlled and why we measure/ensure it that way. With this pattern, we enable other researchers to easily perceive which parameter we considered as relevant, how we controlled and measured it and why. Thus, researchers can easily evaluate the soundness of our experimental design. Furthermore, we can support replication, because relevant information for confounding parameters is presented at one defined location.

We are aware that most reports on experiments have space restrictions. To avoid incomplete descriptions of confounding parameters, we recommend to give only a short description of the most important parameters in the report, and provide the complete list of parameters, according control techniques, and measurements in accompanying material (e.g., on a website or a technical report) using the same pattern. This way, reports do not become bloated, but all relevant information is available.

Parameter	Control technique		Measured/Ensured	
	How?	Why?	How?	Why?
Programming experience (PE)	Matching	Major confound	Education level	undergraduates have less experience than graduates
Rosenthal effect (RE)	Avoided	Reliable	Standardized instructions	Most reliable
Ties to persistent memory (Ties)	Ignored	Not relevant	—	—

Table 4.9: *Pattern to describe confounding parameters.*

4.7 Threats to Validity

Internal validity is threatened by the selection and extraction process of articles and parameters and by our list of keywords. External validity is threatened by the selection of journals, conferences, and issues.

4.7.1 Internal Validity

The selection of articles and extraction of parameters may be biased. In our survey, we had one reviewer selecting the papers, the other reviewer extracting the confounding parameters. To minimize bias, we checked the work of the other reviewer on random samples in alignment with suggestions of [Kitchenham and Charters, 2007]. That is, the reviewer who selected the papers checked the extraction process, and the reviewer who extracted the parameters checked the selection process. Thus, we sufficiently controlled this threat.

Furthermore, the list of keywords ((programming) experience, expert, expertise, professional, subject, participant) may lead to incorrectly excluding a paper. However, these keywords are typical for experiments. Additionally, we used these keywords in conjunction with skimming the paper to minimize the threat of falsely discarding a paper. Furthermore, we excluded several papers of our initial set, so we do not have irrelevant papers in our final set. Thus, our final set of papers is representative, so we minimized the threat caused by the selection of keywords.

4.7.2 External Validity

As for every literature survey, the selection of journals, conferences, and issues as well as the data extraction may be biased. First, we selected three journals and four conferences that are the leading publication platform in their field. However, seven sources may be not enough for a literature survey. To reduce this threat, we selected a broad spectrum, so that we also included more general sources in the area of software engineering, not only sources for empirical research. Additionally, we could have considered a larger time frame, but 10 years is enough to get at least a starting point for an exhaus-

tive overview of confounding parameters. In future work, we can consider additional journals, conferences, and years to extend our list.

Additionally, it is unlikely that we extracted all relevant confounding parameters for program comprehension. Although we had a broad set of papers, there might be parameters missing. For example, the size of the monitor on which the study object is presented might influence the result, or the operating system, because participants are used to a different one than what is used in the experiment. Thus, our list can be extended. To minimize the number of missed parameters, we set the selection and extraction criteria for papers and confounding parameters as broad as possible. Thus, our list provides a good foundation for creating sound experimental designs.

4.8 Related Work

Based on work in psychology, Wohlin and others provide a checklist of confounding parameters for software-engineering experiments [Wohlin et al., 2000]. This is a good starting point for experiments. However, the application of this list was not evaluated based on a literature survey. Furthermore, this checklist applies for experiments in software engineering in general, not specific to comprehension experiments.

There are a lot of literature surveys about experiments in software engineering. For example, Sjøberg and others conducted a survey about the amount of empirical research in software engineering [Sjøberg et al., 2005]. Dybå and others analyzed the statistical power in software-engineering experiments [Dybå et al., 2006]; Kampenes and others analyzed the conduct of quasi experiments [Kampenes et al., 2009]; Budgen and others analyzed how UML and different facets of it have been studied empirically [Budgen et al., 2011]. In addition to these meta studies, Kitchenham and others conducted tertiary studies, for example, about systematic reviews in software engineering [Kitchenham et al., 2009]. Our work is similar, in that we draw conclusion about the status of empirical research in software engineering. However, we focus on the facet of confounding parameters and give recommendations how to manage them.

4.9 Summary

Program comprehension is an internal cognitive process and should be evaluated based on controlled experiments with human participants. One of the major obstacles for conducting experiments is identifying and controlling for confounding parameters. To reduce this obstacle, we set as goal for this chapter to provide a list of confounding parameters. This way, we give researchers a tool to design reliable and valid experiments.

To fulfill our goal, we conducted a literature survey of seven major journals and conferences of the last ten years. We identified 39 personal and experimental parameters. We described how they were controlled for in literature and gave recommendations how to manage them in experiments. Researchers can consult our list when identifying confounding parameters for program comprehension. Hence, we reduced the effort for planning experiments.

Furthermore, we gave an overview of the state of the art, indicating that confounding parameters are managed in an unsystematic way. To improve this situation, we recommend to systematically decide whether a parameter is relevant and what control technique(s) to apply. Additionally, we recommend to explicitly describe a confounding parameter and its control technique in a report. This way, researchers can evaluate the soundness of an experimental design and know where to find information about confounding parameters.

Chapter 5

Measuring Programming Experience

This chapter shares content with the ICPC'12 paper "Measuring Programming Experience" [Feigenspan et al., 2012c].

In the previous chapter, we presented a list of confounding parameters based on a literature survey. In this chapter, we describe programming experience, one of the most important and most often considered confounding parameter for program-comprehension experiments: Novice programmers need considerable more time to implement and maintain programs than expert programmers [McConnell, 2011; Sackman et al., 1968]. Currently, there is no valid way to measure programming experience. Thus, we defined the following goal for this chapter:

- Questionnaire to reliably and conveniently measure programming experience.

We aim at a questionnaire, because, in psychology, they have proven as a useful, easy-to-apply, and often used instrument to measure numerous facets of the human mind, for example, personality traits [Myers and McCaulley, 1985]. Thus, if we can give researchers a convenient instrument to measure programming experience, we can reduce the effort for designing program-comprehension experiments.

We developed the questionnaire based on the literature survey presented in the previous chapter, so that we consider the opinion of different researchers and increase acceptance of the questionnaire. We present a summary of the literature survey and results in Section 5.1. In Section 5.2, we present the questionnaire to measure programming experience, so that the reader can get an impression of it. To evaluate our questionnaire, we conducted a controlled experiment, in which we let participants complete the questionnaire and compare the answers with the performance in program-comprehension tasks. We present the experiment in Section 5.3 and the results in Section 5.4. Furthermore, we conduct an exploratory analysis to identify the most relevant questions of our questionnaire and to find a model that describes programming experience in Section 5.5. To enable other researchers to measure programming experience and develop their own questionnaire, we give recommendations in Section 5.6.

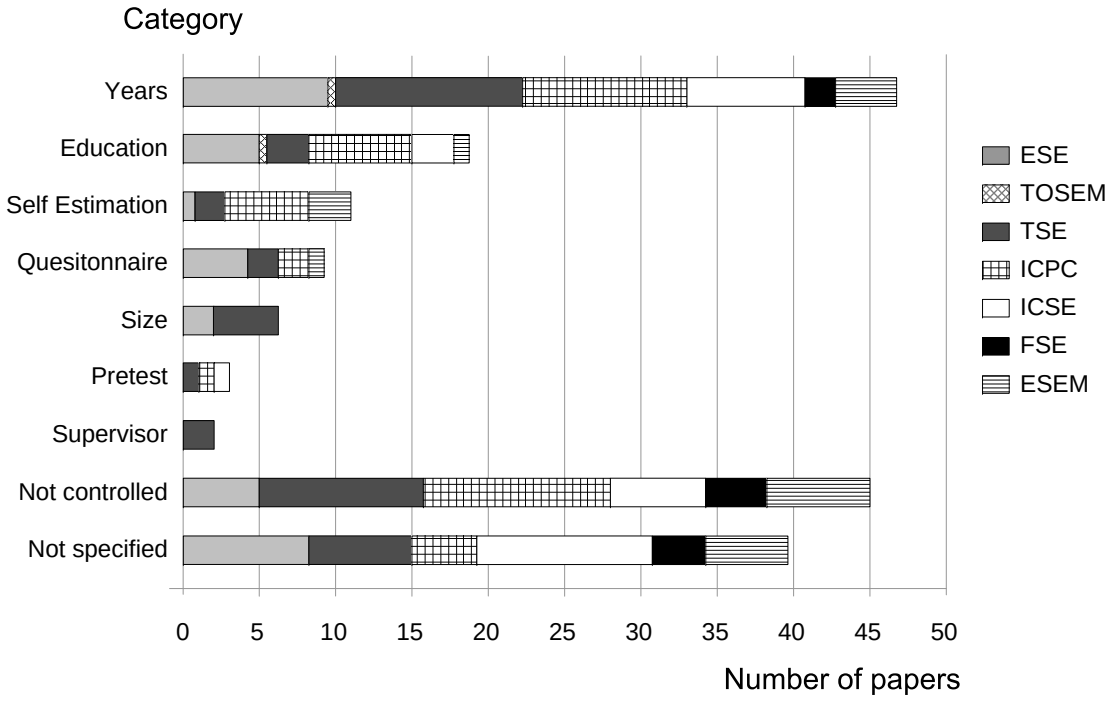


Figure 5.1: *Operationalization of programming experience.*

5.1 Literature Survey

To evaluate how other researchers measure programming experience, we consulted the paper of our literature survey we presented in the previous chapter. The only difference is that we focused on programming experience and its measurement. Hence, we do not explain the selection and extraction process here, but present only the results.

We found several ways to measure programming experience, which we divide into 7 categories and summarize in Figure 5.1. Furthermore, we found that authors did not specify how the measured programming experience or did not control for it. The categories are not disjoint: When authors combined indicators from different categories, the according paper counts for each category.

1. *Years:* In many papers ($47/29\,\%$), the years participants were programming at all or programming in a company or certain language was used to measure programming experience. For example, Sillito and others asked the number of years participants were programming professionally [Sillito et al., 2008].

2. *Education:* The education of participants was used to indicate their experience in 19 ($12\,\%$) of the reviewed papers. Education includes information such as the level of education (e.g., undergraduate or graduate student) or the grades of courses. For example, Ricca and others recruited undergraduate students as low-experience and graduate students as high-experience participants [Ricca et al., 2007].

3. *Self estimation:* In 12 (7 %) papers, participants were asked to estimate their experience themselves (e.g., with Java or object-oriented programming). For example, Bunse let his participants estimate their experience on a five-point scale [Bunse, 2006].

4. *Unspecified questionnaire:* Some authors (9/6 %) applied a questionnaire to assess programming experience. For example, Erdogmus and others let participants fill out a questionnaire before the experiment [Erdogmus et al., 2005]. However, none of the authors specified what the questionnaire looked like.

5. *Size:* The size of programs participants have written was used as indicator in 6 (4 %) papers. For example, Müller [2004] asked how many lines of code the largest program has that participants had implemented.

6. *Unspecified pretest:* In 3 (2 %) experiments, a pretest was conducted to assess participants' programming experience. For example, Biffl and Grossmann [Biffl and Grossmann, 2001] used a pretest to create three groups of skill levels (excellent, medium, little). However, none of the authors specified what the pretest looked like.

7. *Supervisor:* In two experiments (1 %), in which professional programmers were recruited as participants, the supervising managers estimated the experience of participants [Arisholm et al., 2007; Hannay et al., 2010].

8. *Not specified/not controlled:* Often, the authors state that they measured programming experience, but did not specify how (39/24 %). Even more often (45/28 %), authors did not mention programming experience at all, which may threaten the validity of the corresponding experiments.

The measurement of programming experience is diverse, which could threaten the validity of experiments, because researchers use their own definition of programming experience without validating it. Thus, we cannot be sure whether researchers really captured programming experience or something else.

Additionally, when authors do not specify how they measured programming experience, it is difficult to compare experiments. For example, if in one study graduate students were categorized as expert programmers, and in another study as novice programmers, then expert programmer means different things. Now, if we compare experts of both studies, we actually do not have comparable programming experience, although the studies suggest that. Thus, when we compare results of both studies without knowing that expert describes different levels of programming experience, we might draw wrong conclusions. This is especially problematic in meta analyses, in which conclusions are drawn based on a number of experiments.

Another interesting observation is that, in none of the papers, we found a definition of programming experience, but authors only described how they measured it. There seems to be an implicit consensus of what programming experience is. To make this understanding explicit, we asked four programming experts to define programming experience. In summary, most experts had difficulties finding a clear, explicit definition. During discussions, the following definition emerged:

> "Programming experience describes the amount of acquired knowledge regarding the development of programs, so that the ability to analyze and create programs is improved."

In future work, we plan to evaluate whether this definition holds when consulting a representative set of programming experts. For now, since our goal is to measure programming experience, not define it, we adopt this definition for our work.

5.2 Programming-Experience Questionnaire

Most measurements of programming experience that we found in literature can be performed as part of a questionnaire. Only pretest and supervisor estimation require additional effort, but are also rarely used in our analyzed papers. Hence, we excluded both categories from our analysis. Furthermore, we excluded the category *unspecified questionnaire*, because the contents of questionnaires were not specified in our analyzed papers.

We designed a single questionnaire, which includes questions of the remaining categories *years*, *education*, *self estimation*, and *size*. For each category, we selected multiple questions that we found in literature. Additionally, we added questions that we found in previous experiments and during talking to programming experts to be related to programming experience, but were not mentioned in literature. This way, we aim at having a more exhaustive set of indicators for programming experience and, consequently, a better measurement of programming experience. During our analysis, we can exclude questions that prove irrelevant. Some questions are specific to students; when working with different participants (e.g., professional programmers), they need to be adapted.

In Table 5.1, we summarize our questionnaire. The version of the questionnaire we used in our experiment is available at the project's website. We also show the scale of the answers, that is, how participants entered their answers. In column "Abbreviation", we show the abbreviation of each question (the first letter encodes the category), which we use in the remainder of this chapter.

5.2.1 Years

Questions of this category mostly referred to how many years participants were programming in general and professionally. Programming in general describes the time since participants started programming, which includes hello-world-like programs (*y.Prog*). Professional programming describes when participants earned money for programming, which typically requires a certain experience level (*y.ProgProf*). In our questionnaire, we asked both questions. We believe that both are an indicator for programming experience, because the longer participants are programming, the more source code they implemented and, thus, the higher their programming experience should be.

5.2.2 Education

This category contains questions that assess educational facets. We asked participants to state the number of courses they took in which they implemented source code (*e.Courses*)

Category	Question	Scale	Abbreviation
Self estimation	Programming experience	1 to 10	s.PE
	Programming experience compared to experts	1 to 5	s.Experts
	Programming experience compared to class mates	1 to 5	s.ClassMates
	Experience with Java/Prolog/C/ Haskell	1 to 5	s.Java/s.Prolog/ s.C/s.Haskell
	Number of further languages with at least medium experience	Integer	s.NumLanguages
	Experience with functional/imperative/logical/object-oriented programming	1 to 5	s.Functional/ s.Logical/ s.Imperative/ s.ObjectOriented
Years	Years of programming	Integer	y.Prog
	Years of programming professionally	Integer	y.ProgProf
Education	Year of enrollment	Integer	e.Years
	Number of programming courses	Integer	e.Courses
Size	Size of professional projects	<900, 900–40k, >40k	z.Size
Other	Age	Integer	o.Age

Integer: Answer is an integer; 1 to 10/5: scale from 1 to 10/5, 1 meaning very inexperienced, 10/5 meaning very experienced. The abbreviation of each question encodes also the category to which it belongs.

Table 5.1: *Questions to assess programming-experience.*

and the year in which they enrolled (*e.Years*), recoded into number of years participants were enrolled. The number of courses roughly indicates how much source code participants have implemented. With the years participants are studying, we get an indicator of the education level: The longer participants have been studying, the more experience they should gained through their studies.

5.2.3 Self Estimation

In this category, we asked participants to estimate their own experience level. With the first question, we asked participants to estimate their programming experience on a scale from 1 to 10 (*s.PE*). We did not clarify what we mean by programming experience, but let participants use their intuitive definition of programming experience to not use a definition that felt unnatural. We used a 10-point scale to have a fine-grained estimation. In the remaining questions, we used a five-point scale, because a coarse-grained estimation is better for participants to estimate their experience in these more specific questions.

Next, we asked participants to relate their programming experience to experienced programmers (*s.Experts*) and their class mates (*s.ClassMates*) to let participants think more thoroughly about their level of experience.

Additionally, we asked participants how familiar they are with certain programming languages. We chose Java (*s.Java*), C (*s.C*), Haskell (*s.Haskell*), and Prolog (*s.Prolog*) as common languages and because they are taught at the universities our participants were enrolled at. The more programming languages participants are familiar with, the more they have learned about programming in general and their experience should be larger. Furthermore, experience with the underlying programming language of the experiment can be assessed. Beyond that, we asked participants to state the number of programming languages in which they are experienced at least to a medium level (*s.NumLanguages*). This way, we can assess familiarity with many languages without listing each of them. The same counts for familiarity with different programming paradigms (*s.Functional, s.Logical, s.Imperative, s.ObjectOriented*).

5.2.4 Size

We asked participants with professional experience about the size of their projects (*z.Size*). We used the categorization into small (< 900), medium (900–$40\,000$), and large ($> 40\,000$) based on the lines of code according to von Mayrhauser and Vans [1995]. The larger the size of projects, the more experienced participants should be, because they saw more code and need to have a certain skill level.

In addition, we also included the age of participants (*o.Age*) in the questionnaire, because it might be possible that the older participants are, the more code they have seen, and the more experienced they should be. This way, we aim at having a more exhaustive understanding of programming experience.

5.3 Empirical Validation

Constructing and validating a questionnaire is a long and tedious endeavor that requires several (replicated) experiments [Peterson, 2000]. In this chapter, we start this process.

There are different ways to validate a questionnaire. We could recruit programming experts and novices as participants and compare their answers in the questionnaire. Since we know there is a difference in the experience between both groups, we should also see a difference in the questionnaire. Another way is to compare the answers in the questionnaire with performance in tasks that are related to programming experience. The benefit is that we do not need different groups of participants—one group is sufficient. We used the latter way with a group of students, because we found in our survey that they are often recruited as participants in software-engineering experiments. Hence, they represent an important sample.

For better overview, we present the most important information of the experiment in Table 5.2.

Since we recruit students, we expect only little variation for some questions (e.g., o.Age). We asked these questions anyway to have a more exhaustive data set. Of course,

Context	Description	Section
Objective	Evaluate constructed programming-experience questionnaire	5.3.1
Material	9 small programs, such as a sorting algorithm; 1 variant of MobileMedia; Questionnaire	5.3.2
Participants	128 undergraduate students from the universities of Passau, Marburg, and Magdeburg	5.3.3
Tasks	Determining the output of a method	5.3.4
Execution	One computer lab per university; 17 to 22" TFT; PROPHET	5.3.6
Analysis	Correctness of answers; correlations of correctness and answers in questionnaire; stepwise regression to extract relevant questions; exploratory factor analysis to explore model of programming experience	5.4
Result	Self estimation suitable to measure programming experience; five-factor model of programming experience	5.6

Table 5.2: *Experiment in a nutshell.*

further experiments with different groups of participants (e.g., professional programmers) are necessary, which we plan to do in future work. To this end, we can reuse our experimental design.

5.3.1 Objective

With our experiment, we aim at evaluating how the questions relate to programming experience. To this end, we need an indicator for programming experience to which we can compare the answers of our programming-experience questionnaire. Hence, we designed programming tasks that participants should solve in a given time. For each task, we measure whether participants solve a task correctly and how long they need to complete a task. The first underlying assumption is that the more experienced participants are, the more tasks they solve correctly. Since experienced participants have seen more source code compared to inexperienced participants, they should have less trouble in analyzing what source code does and, hence, solve more tasks correctly. The second assumption is that experienced participants are faster in analyzing source code, because they have done it more often and know better what to look for.

As we are starting the validation, we have no hypotheses about how our questions relate to the performance in the programming tasks. Instead, we have a research questions:

RQ: Do the questions of our questionnaire correlate with programming experience?

```
1  public class Class1 {
2     public static void main(String[] args) {
3        int array[] = {14,5,7};
4
5        for (int counter1 = 0; counter1 < array.length; counter1++) {
6           for (int counter2 = counter1; counter2 > 0; counter2--) {
7              if (array[counter2 - 1] > array[counter2]) {
8                 int variable1 = array[counter2];
9                 array[counter2] = array[counter2 - 1];
10                array[counter2 - 1] = variable1;
11             }
12          }
13       }
14
15       for (int counter3 = 0; counter3 < array.length; counter3++)
16          System.out.println(array[counter3]);
17       }
18    }
19 }
```

Figure 5.2: *Source code for the first task.*

5.3.2 Material

As material, we selected typical algorithms presented in introductory programming lectures. This way, we match the average experience level of undergraduates, our participants. When replicating this experiment with programming experts, the tasks should be matched to the high level of experience. For illustration, we show the source code of the first task in Figure 5.2. The source code sorts an array of numbers, so the correct answer is "$5, 7, 14$". The remaining algorithms were roughly similar (except for two): Two algorithms implemented a stack and five a linked list.

In two algorithms, one involved command-line parameters and one was a running variant of MobileMedia (i.e., without information of features in the code) of the fifth release. We included these two algorithms to identify highly experienced participants among second-year undergraduate students, since some students start to program before their studies. We expected that only highly experienced participants should be able to sufficiently understand these algorithms in the given time. In Appendix 10.3, we present source code of the first 9 tasks.

All source code was in Java, the language that participants were most familiar with.

To present the questionnaire, tasks, and source code, we used our tool infrastructure PROPHET (cf. Chapter 6).

5.3.3 Participants

Participants were recruited from the University of Passau (27), Philipps University Marburg (31), and University of Magdeburg (70), so we had 128 participants in total. All universities are located in Germany. Participants from Passau and Marburg were in the end of their third semester and attended a course on software engineering. Participants from Magdeburg were at the beginning of their fourth semester and from different courses. The level of education of all participants was comparable, because no courses took place

between semesters and participants had to complete similar courses at all universities. All participants were offered different kinds of bonus points for their course (e.g., omitting one homework assignment) for participating in the experiment independent of their performance. All participants took part voluntarily, were aware that they participated in an experiment, and could quit anytime without consequences. Data were logged anonymously.

Since we recruited participants from three different universities, we actually have three different samples. However, only the question s.ClassMates is specific for each university, because participants can only compare themselves to the students of their university. A Kruskal-Wallis test to evaluate whether there are significant differences in the answer of s.ClassMates between the three universities was not significant ($\chi^2 = 1.275$, df = 2, p = 0.529) [Anderson and Finn, 1996]. Furthermore, we selected the tasks to be typical examples of what students learn in introductory programming courses at their universities. Hence, we can treat our three samples as one sample.

5.3.4 Tasks

We had ten different algorithms. For the first nine, participants should determine what executing the algorithms would print. Furthermore, participants should explain what the source code is doing.

In the last task, we used MobileMedia, in which we introduced a bug into class `AddPhotoToAlbum`, such that a variable holding the path of a photo was initialized with `null` instead of the actual path. Thus, a `NullPointerException` would be thrown. Participants got a bug description explaining the behavior and should locate the cause of the bug, explain why it occurs, and suggest a solution.

An answer was correct when it was result of running the program, ignoring whitespace (or the correct bug location). When an answer diverged from the expected result, a programming expert analyzed participants' explanation of the source code and decided whether the answer could be counted as correct.

We had 10 tasks so that only experienced participants would be able to complete all tasks in the given time of 40 minutes, which we confirmed in a pretest with PhD students from the University of Magdeburg. This way, we can better differentiate between high and low experienced participants. To make sure that participants are not disappointed with their performance in the experiment, we explained that they would not be able to solve all tasks, but should simply proceed as far as possible within the given time. Another way would be to have no time limit and let participants work until they completed all task. However, we had a time slot of only one hour, including introduction and debriefing, so we specified a time limit for all tasks.

5.3.5 Confounding Parameters

In alignment with our suggestion of managing confounding parameters, we summarize the most important confounding parameters for our study and describe how we controlled them. In Table 5.3, we give an overview.

Parameter	Control technique		Measured/Ensured	
	How?	Why?	How?	Why?
Personal parameters				
Ability	Randomization	No resources for measurement		
Domain knowledge	Constant	Limited resources	Asked participants	Reliable
Education	Constant	No training necessary to ensure familiarity with study object	Participants of similar courses	Comparable knowledge
Intelligence	Randomization	No resources for measurement		
Motivation	Analyzed afterwards; voluntary participation	Can change during experiment	Asked participants	Reliable
Experimental parameters				
Evaluation apprehension	Avoided	To get actual performance	Anonymized data	Performance not biased
Hawthorne effect	Avoided	Reliable	Not revealed hypotheses	Performance not biased
Learning effects	Accepted	Experienced participants should learn faster	Similar tasks	Observe learning effect
Operationalization of study object	Avoided	Reliable	Suitable based on experts	Learned from others
Rosenthal	Avoided	Reliable	Standardized instructions	Reliable
Selection	Accepted	No according resources	Interpretation restricted to selected sample	
Technical problems	Avoided	Reliable	According data excluded	Repitition not feasible
Time pressure	Accepted	Time constraints	Told participants that completing all tasks is impossible; should work as fast and correct as possible	Minimized time pressure

Table 5.3: *Selection of confounding parameters and applied control techniques.*

For intelligence and ability, we used randomization, because we had not time to reliably measure it before the experiment. To control for education, we kept it constant by recruiting students from similar courses. This way, we ensure that participants learned similar topics during their studies and have the same familiarity with the algorithms we used in our experiment. We kept domain knowledge constant by obfuscating identifier names. This way, we ensure bottom-up comprehension. Since we need a large sample size, we cannot test top-down and bottom-up comprehension, because that would significantly increase the required sample size. For motivation, we asked participants afterwards how motivated they were to solve a task, because the level of motivation can change during the experiment; additionally, participation was voluntary.

To control for evaluation apprehension, we anonymized the data, so that we get the actual performance of participants. We avoided the Hawthorne effect by not revealing our hypotheses; thus, participants could not bias their performance. For learning effects, we expect that more experienced participants should learn faster; thus, some tasks were similar, so more experienced participants should solve more tasks correctly. To have a suitable operationalization of study object, we consulted programming experts, who confirmed that programming experience should correlate with the number of correct answers in our tasks.

To avoid the Rosenthal effect, we carefully developed standardized instructions for the one experimenter who instructed participants. We accepted bias due to the selection of our sample, because we do not have resources to have a heterogeneous sample (e.g., with programming experts). Thus, we interpreted the results only in the context of our sample (i.e., undergraduate students). Finally, we avoided bias due to technical problems by excluding the affected question of the questionnaire. We could not collect the answers of that question afterwards, because participants were then aware of the hypotheses, which could have introduced the Hawthorne effect.

We present the remaining parameters in Appendix 10.2. Since the parameters and control techniques are similar for all our experiments, we summarize the confounding parameters of all experiments to avoid redundancy.

5.3.6 Experiment Execution

The experiments took place in January and April 2011 at the Universities of Passau, Marburg, and Magdeburg as part of a regular lecture session. First, we let participants complete the programming-experience questionnaire without knowing its specific purpose. Then, we gave participants an introduction about the general purpose and proceeding of the experiment, without revealing our goal. The introduction was given by the same experimenter each time. After all questions were answered, participants worked on the tasks on their own. Since we had time constraints, the time limit for completing the tasks was 40 minutes. After time ran out, participants were allowed to finish the task they were currently working on. Two to three experimenters checked that participants worked as planned. After the experiment, we revealed the purpose of this experiment to the participants.

One deviation occurred, in that we had a technical error for the presentation of the programming-experience questionnaire, such that we could not measure s.PE for all par-

No.	Question	Distribution	Completed
1	s.PE		70
2	s.Experts		126
3	s.ClassMates		127
4	s.Java		124
5	s.C		127
6	s.Haskell		128
7	s.Prolog		128
8	s.NumLanguages	0 6	118
9	s.Functional		127
10	s.Imperative		128
11	s.Logical		126
12	s.ObjectOriented		127
13	y.Prog	0 25	123
14	y.ProgProf	0 7	127
15	e.Years	0 9	126
16	e.Courses	0 20	123
17	z.Size		128
18	o.Age	19 40	128

Completed: number of participants who completed this task.

Table 5.4: *Answers in questionnaire.*

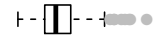
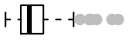
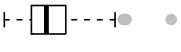
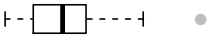
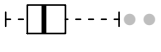
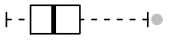
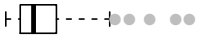
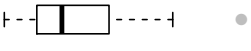
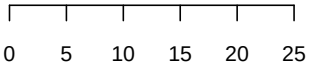
ticipants. Hence, we only have the answer of 70 (of 128) participants for this question. We discuss the implication in Section 5.5.

5.4 Experiment Results

First, we describe descriptive statistics to get an overview of our data. Second, we present how each question correlates with the performance in the tasks. This way, we get an impression of how important each question is as indicator for programming experience in our sample.

5.4.1 Descriptive Statistics

In Table 5.4, we show the answers participants gave in our questionnaire. The median for s.PE varies between 2 and 3, which we would expect from second-year undergraduate students. In general, participants felt very inexperienced with logical programming and experienced with object-oriented programming. The median of how long participants are programming is four years, but only few participants said they were programming for more than ten years. Although participants took a second-year undergraduate course,

Task	Response time in minutes		Completed	Correct	
	Distribution	Mean		Absolute	Relative
1		4.44	124	70	55 %
2		3.65	123	90	70 %
3		5.02	121	97	76 %
4		6.17	117	22	17 %
5		4.06	118	46	36 %
6		4.72	111	40	31 %
7		2.34	92	31	24 %
8		4.1	82	69	54 %
9		1.94	78	11	9 %
10		9.64	30	22	17 %
					
	0 5 10 15 20 25				

Completed: number of participants who completed this task;
Correct: number of participants with correct solution.

Table 5.5: *Response time for each task.*

some participants were enrolled for more than three years,[1] which could also explain why some participants completed numerous courses in which they had to implement source code.

In Table 5.5, we give an overview of how participants solved the tasks. Column "Mean" contains the average time in minutes of participants who completed a task. Since not all participants finished all tasks, they cannot be interpreted across tasks. We discuss the most important values. Task 10 took the longest time to complete (on average, 9.6 minutes). This is caused by size of the source code of MobileMedia for the last task with 2 800 lines of code. To solve Task 9, participants needed on average 1.9 minutes; most likely, because its source code consisted of only 10 lines. However, only 11 participants solved it correctly. To solve this task, participants must be familiar with command-line parameters, which may not be typical for the average second-year undergraduate student. Considering the correctness of Task 4, we see that only 22 participants solved it correctly. In this task, elements were added to an initially empty linked list, such that the list is sorted in a descending order after the insertion. In most of the wrong answers, we found that the order of the elements was wrong. We believe that participants did not analyze the insert algorithm thoroughly enough and assumed an ascending order of elements.

In Figure 5.3, we show the number of correctly solved tasks per participant. As we expected, none of our participants solved all tasks correctly (cf. Section 5.3.2). Especially

[1] The German system allows participants to take courses in a somewhat flexible order and timing.

the last two tasks (Task 9: command-line parameters; Task 10: bug fix in MobileMedia) required an experience level beyond that of second-year undergraduate students. More than half of the students (72) solved two to four tasks correctly. Taking into account the time constraints (40 minutes to solve 10 tasks), it is not surprising that the number of correctly solved tasks is that low.

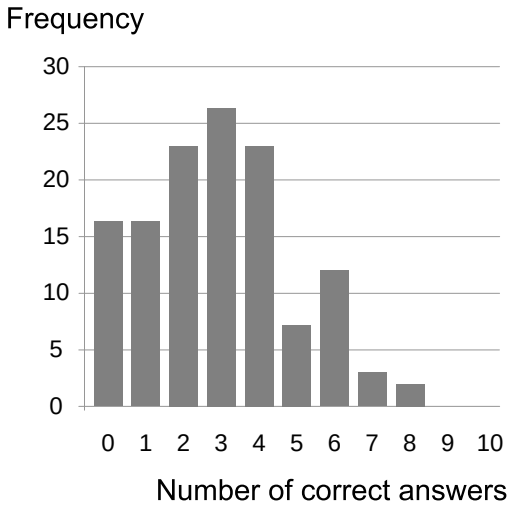


Figure 5.3: *Number of correct answers for all tasks.*

5.4.2 Correlations

In Table 5.6, we give an overview of the correlation of the number of correct answers with the answers of the questionnaire. Since we correlate ordinal data, we use the Spearman rank correlation [Anderson and Finn, 1996]. For about half of the questions of self estimation, we obtain small to strong correlations.[2] The highest correlation with number of correct answers has s.PE. The lowest significant correlation is with s.NumLanguages. Regarding y.Prog and y.ProgProf, we have medium correlations with the number of correct answers. E.Years does not correlate with the number of correct answers. For the remaining questions, we do not observe significant correlations.

For completeness, we show the correlations of response time with each of the questions of our questionnaire in Table 5.7. Only 23 correlations, of 180, are significant, which is in the range of coincidence, given the common significance level of 0.05. Since there are so many correlations, a meaningful interpretation is impossible without further analysis, for example, a factor analysis. However, such analysis typically requires a larger number of participants. Since we have a decreasing number of participants with each task, we leave analyzing the response times for future experiments.

In the next section, we discuss which questions are good indicators for programming experience.

[2]Correlations can beare categorized as small (± 0.1 to ± 0.3), medium (± 0.3 to ± 0.5), or strong (± 0.5 to ± 1) [Cohen, 1988].

No.	Question	ρ	N
1	s.PE	0.539	70
2	s.Experts	0.292	126
3	s.ClassMates	0.403	127
4	s.Java	0.277	124
5	s.C	0.057	127
6	s.Haskell	0.252	128
7	s.Prolog	0.186	128
8	s.NumLanguages	0.182	118
9	s.Functional	0.238	127
10	s.Imperative	0.244	128
11	s.Logical	0.128	126
12	s.ObjectOriented	0.354	127
13	y.Prog	0.359	123
14	y.ProgProf	0.004	127
15	e.Years	-0.058	126
16	e.Courses	0.135	123
17	z.Size	-0.108	128
18	o.Age	-0.116	128

ρ: Spearman correlation; N: number of participants;
gray cells denote significant correlations ($p < .05$).

Table 5.6: *Spearman correlations of number of correct solutions with answers in questionnaire.*

No.	Question	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9	Task 10	Number of participants
1	s.PE	-0.279	-0.417	-0.042	0.004	-0.002	0.016	0.014	-0.182	0.071	0.085	68 – 27
2	s.Experts	-0.300	-0.177	0.047	-0.026	0.006	-0.075	-0.217	-0.004	0.206	0.131	122 – 40
3	s.ClassMates	-0.189	-0.401	-0.084	-0.065	-0.053	-0.059	-0.163	-0.061	0.161	0.100	123 – 40
4	s.Java	0.029	-0.066	-0.154	-0.022	-0.066	0.003	-0.040	0.145	-0.170	-0.222	105 – 34
5	s.C	-0.175	-0.124	0.018	0.027	0.126	-0.108	-0.056	-0.052	0.043	0.108	123 – 40
6	s.Haskell	-0.171	-0.109	-0.144	-0.113	-0.014	-0.216	-0.153	-0.183	0.019	0.158	124 – 40
7	s.Prolog	-0.174	-0.141	-0.079	-0.104	-0.027	-0.039	0.076	-0.239	-0.047	0.146	124 – 40
8	s.NumLanguages	-0.295	-0.339	-0.131	-0.121	-0.027	-0.103	-0.035	-0.090	0.232	0.168	115 – 34
9	s.Functional	-0.148	-0.150	-0.150	-0.004	-0.017	-0.204	-0.120	-0.217	0.027	0.175	123 – 40
10	s.Imperative	-0.283	0.331	-0.033	-0.089	-0.06	-0.129	-0.296	-0.156	0.126	0.043	124 – 40
11	s.Logical	-0.209	-0.105	-0.158	-0.136	-0.022	-0.014	0.058	-0.257	-0.191	0.108	122 – 40
12	s.ObjectOriented	-0.084	-0.232	-0.008	0.012	-0.093	-0.034	0.025	-0.060	0.156	0.082	123 – 40
13	y.Prog	-0.241	-0.379	-0.144	-0.071	0.010	-0.113	-0.258	-0.159	0.273	0.180	120 – 38
14	y.ProgProf	-0.217	-0.196	-0.012	-0.119	-0.130	0.071	-0.274	-0.022	0.044	-0.010	123 – 39
15	e.Years	-0.032	0.001	0.018	-0.152	0.059	0.047	-0.119	0.037	-0.092	-0.173	122 – 40
16	e.Courses	-0.146	-0.088	-0.040	-0.062	0.071	0.028	-0.053	-0.004	0.268	0.058	120 – 38
17	z.Size	-0.155	-0.160	-0.057	-0.134	0.059	0.003	-0.201	0.046	0.000	-0.023	124 – 40
18	o.Age	0.036	0.014	0.110	0.082	0.131	0.102	-0.081	0.090	0.059	0.010	124 – 40

Gray cells denote significant correlations ($p < .05$).

Table 5.7: *Spearman correlations of response times for each task with answers in questionnaire.*

5.5 Exploratory Analysis

In this section, we explore the data further to extract relevant questions for measuring programming experience and to develop a model describing programming experience. For this analysis, we excluded question s.PE, because only 70 participants answered this question (cf. Section 5.3.6). Alternatively, we could have removed participants who did not answer this question from the analysis, but this would have made our sample too small for the exploratory analysis.

5.5.1 Stepwise Regression

So, which questions are the best indicators for programming experience? The first obvious selection criterion is to include all questions that have at least a medium correlation (> 0.30) with the number of correctly solved task, because they are typically considered as relevant. However, the questions themselves might correlate with each other. For example, the s.ClassMates correlates with s.ObjectOriented with 0.552. Hence, we can assume that both questions are not independent from each other. If we used both questions as indicator, we would overestimate the relationship of both questions with programming experience; that is, we would count the common part of both questions twice, although we should count it only once. To account for the correlations between questions, we use *stepwise regression* [Lewis-Beck, 1980].

Background: Stepwise Regression With stepwise regression, we build a model of the influence of the questions on the number of correct answers in a stepwise manner. We start by including the question with the highest correlation, which, in our case, is s.ClassMates. Then, we consider the question with the next highest correlation, which is y.Prog. Using this question, we compute the *partial correlation* with the number of correct answers, describing the correlation of two variables cleaned from the influence of a third variable [Cohen and Cohen, 1983]. Thus, the correlation of y.Prog with the number of correct answers, cleaned from the influence of s.ClassMates, is computed. If this cleaned correlation is high enough, the question is included, else it is excluded. The goal is to include questions with a high partial correlation with the number of correct answers, such that as few questions as possible are selected to have a model as parsimonious as possible. This is repeated with all questions of the questionnaire.

In Table 5.8, we show the results for our questionnaire. With stepwise regression (specifically, we used stepwise as inclusion method), we extracted two questions: self-estimated experience compared to class mates (s.ClassMates) and experience with logical programming (s.Logical). The higher the Beta value, the larger the influence of a question on the number of correctly solved tasks. The model is significant ($F_{2,45} = 8.472, p < .002$) and the adjusted R^2 is 0.241, meaning that we explain $24.1\,\%$ of the variance in the number of correct answers with our model.

Hence, the result of the stepwise-regression algorithm is that questions s.ClassMates and s.Logical contribute most to the number of correct answers: The higher participants estimate their experience compared to class mates and their experience with logical pro-

Question	Beta	t	p
s.ClassMates	0.441	3.219	0.002
s.Logical	0.286	2.241	0.030

Table 5.8: *Resulting model of stepwise regression.*

gramming, the more tasks they solve correctly. We believe that stepwise regression extracted s.ClassMates, and not s.Experts, because we recruited students as participants and the tasks are taken from introductory programming lectures. Hence, if participants estimate their experience better than their class mates, they should be better in solving the tasks.

Why was s.Logical extracted and not s.Java, which is closer to our experiment? We believe that the reason is that our participants learn Java as one of their first programming language and feel somewhat confident with it. In contrast, learning a logical programming language is only a minor part of the curriculum of all three universities. Hence, if students estimate that they are familiar with logical programming, they may have an interest in learning other ways of programming and pursue it, which increases their programming experience. However, we need to confirm the results of stepwise regression in future experiments.

The model we received from stepwise regression describes Beta values, which are weights for each question. For example, if a participant estimates a 4 in s.ClassMates (more experienced than class mates) and a 2 in s.Logical (unfamiliar with logical programming), the resulting value for programming experience is $0.441 * 4 + 0.286 * 2 = 2.336$ (for simplicity, we omitted a constant to add as part of the model).

Hence, we identified two questions that explain $24.1\,\%$ of the variance of the number of correct answers. For our sample, these two questions explain programming experience best. We could include more questions to improve the amount of explained variance, but none of the questions contribute a significant amount of variance. Since a model should be parsimonious, stepwise regression excluded all other questions. Thus, for our sample, these two questions provide the best indicators for programming experience.

5.5.2 Exploratory Factor Analysis

Furthermore, to look for a pattern in our questions, we analyzed whether questions in our questionnaire correlate. To this end, we conducted an *exploratory factor analysis* [Anderson and Rubin, 1956].

Background: Exploratory Factor Analysis The goal of an exploratory factor analysis is to reduce a number of observed variables to a small number of underlying *latent* variables or *factors* (i.e., variables that cannot be observed directly). To this end, the correlations of the observed variables are analyzed to identify groups of variables that correlate among each other. For example, the experience with Haskell and functional programming are very similar and might be explained by a common underlying factor. The result of an

exploratory factor analysis is a number of factors that summarize observed variables into groups. The meaning of the factors relies on interpretation.

Furthermore, we obtain correlations or *factor loadings* of the variables in our questionnaire with identified factors. By convention, factor loadings that have an absolute of smaller than 0.32 are omitted, because they are too small to be relevant [Costello and Osborne, 2005]. There are *main loadings*, which are the highest factor loading of one variable, and *cross loadings*, which are all other factor loadings of a variable that have an absolute of more than 0.32. The higher the main loading and the smaller the number and values of cross loadings, the more unambiguously the influence of one factor on a variable is. If a variable has many cross loadings, it is unclear what it exactly measures and more investigations on this variable are necessary in subsequent experiments.

In Table 5.9, we show the results of our exploratory factor analysis. The first factor of our analysis groups the variables s.C, s.ObjectOriented, s.Imperative, s.Experts, and s.Java. This means that these variables have a high correlation amongst each other and can be described by this factor. Except for s.Experts, this seems to make sense, because C and Java and the corresponding paradigms are similar and often taught at universities. Hence, if participants have Java experience, they are most likely familiar with C, too. The same counts for the underlying programming paradigms. We conjecture that s.Experts also loads on this factor, because it explains the confidence level with mainstream programming languages, because C and Java, as well the according paradigms, are taught as state of the art. Thus, if participants estimate to be more familiar with state-of-the-art techniques, they also estimate their experience compared to experts higher. We can name this factor *experience with common languages*.

The second factor contains the variables y.ProgProf, z.Size, s.NumLanguages, and s.ClassMates. These variables fit together well, because the longer participants are programming professionally, the more likely they have worked with large projects and the more languages they have encountered. Additionally, since it is not typical for second-year undergraduates to program professionally, participants who have programmed professionally estimate their experience higher compared to their class mates. We can name this factor *professional experience.*

Factor 3 and 5 group s.Functional/s.Haskell and s.Logical/s.Prolog in an intuitive way. Hence, we name these factors *functional experience* and *logical experience*.

The fourth factor summarizes the variables e.Courses, e.Years, and y.Prog, which are all related to the participant's education. We can name this factor *experience from education*.

Now, we have to take a look at the cross loadings. As an example, we look at e.Years, which also loads on *functional experience*. This means that part of this variable can also be explained by this factor. Unfortunately, we cannot unambiguously define to which factor this variable belongs best, we can only state e.Years has a higher loading on factor *experience from education*. This could also mean that we need two factors to explain this variable. However, with a factor analysis, we are looking for a parsimonious model without having more relationships than necessary. To have a deeper understanding of the relationship of our questions and factors, we need to conduct future experiments.

Variable	Factor 1	Factor 2	Factor 3	Factor 4	Factor 5
s.C	0.723				
s.ObjectOriented	0.700			0.403	
s.Imperative	0.673	0.333		0.303	
s.Experts	0.600	0.326			
s.Java	0.540		0.427		
y.ProgProf		0.859			
z.Size		0.764			
s.NumLanguages	0.335	0.489		0.403	
s.ClassMates		0.449	0.403	0.424	
s.Functional			0.880		
s.Haskell			0.879		
e.Courses				0.795	
e.Years			-0.460	0.573	
y.Prog		0.493		0.554	
s.Logical					0.905
s.Prolog					0.883

Gray cells denote main factor loadings.

Table 5.9: *Factor loadings of variables in questionnaire.*

To sum up the exploratory factor analysis, we extracted five factors: *experience with common languages*, *professional experience*, *functional experience*, *experience from education*, and *logical experience* that summarize the questions of our questionnaire in our sample.

The next step after an exploratory analysis is a *confirmatory analysis*. In a confirmatory analysis, we aim at confirming the model we have received, which has to be done with another data set. If we used the same data set, we could not show that our model is valid in general, but only for our specific data set. Currently, we are collecting data of additional students who answer our questionnaire. So far, we have more than 100 students of the universities of Magdeburg, Passau, Marburg, and Duisburg-Essen who completed the questionnaire. Since confirmatory analysis requires large sample sizes [Bentler and Chou, 1987], we need more participants before we can confirm our results.

5.6 Recommendations

The goal of this chapter is to develop a reliable and easy-to-apply questionnaire to measure programming experience. So far, we combined different questions from different categories found in literature into a single questionnaire. We conducted a controlled experiment with undergraduate students and explored our data for initial validation. Based on the results, we give the following recommendations for future research:

First, we showed that in literature, there are many different ways to measure and control programming experience. Furthermore, in many cases, the control techniques are not reported. We recommend to mix questions from different categories into a single questionnaire, of which we presented a draft. We recommend to report precisely which measure was used and how groups have been formed according to it. This helps to judge validity and compare and interpret multiple studies.

Second, we can recommend self-estimation questions to judge programming experience among undergraduate students. In our experiment, several self-estimation questions correlated with a strong to medium degree (s.PE: 0.539; s.ClassMates: 0.403; s.ObjectOriented: 0.354) with the number of correct answers—much more than questions regarding the categories education, size, and other. Among undergraduate students, answers to questions from the latter categories differ only slightly. The only medium correlation beyond self estimation is y.Prog (0.359), the number of years a participant is programming at all.

Third, if resource constraints allow it, researchers can combine multiple questions, of which some serve as control questions to see whether participants answer consistently, which is custom in designing questionnaires [Peterson, 2000]. For example, in our case, when using s.PE, questions s.ClassMates and s.ObjectOriented are suitable as control questions, since they both show a strong correlation with s.PE (s.ClassMates: 0.625; s.ObjectOriented: 0.696).

Fourth, since correlations between questions confound the strength of a question as indicator for programming experience (cf. Section 5.5.1), we applied stepwise regression and extracted two relevant questions, s.ClassMates and s.Logical, that together serve as best indicator to predict the number of correct answers in our experiment (each question can be supplied with control questions).

Fifth, our exploratory factor analysis indicates five factors for programming experience that can serve as starting point for developing a theory on programming experience. The results do not help building a survey right away, but with additional confirmation, such as confirmatory factor analysis on other data sets, they can help understanding how programming experience works and which kinds of questions query relevant parameters. However, to that end, there is still a long way.

Last, our proceeding can serve as recommendation for developing questionnaires. We constructed the questionnaire based on a literature survey, to include how other researchers understand programming experience. Another way is to consult experts on programming experience (e.g., programming experts, project managers) about their opinion on programming experience. We can also combine literature survey and expert consultancy. After we constructed questionnaire, we evaluated it with a controlled experiment.

Overall, note that while our literature survey and the construction of the questionnaire are intended for measuring programming experience in general, we only validated it for a specific setting: predicting programming experience among a homogeneous group of undergraduate students. This way, we achieve high internal validity, because our results are not confounded by different backgrounds of the participants. However, our recommendations remain limited to this setting. We conjecture that with experienced programmers, questions from the categories years and size have more predictive power.

Whether self estimation remains a good indicator in this setting remains an open question for future work. Thus, with our work, we completed only the first steps for developing a questionnaire—in future work, we plan to continue our endeavor.

5.7 Threats to Validity

Threats to internal validity are caused by the selection of tasks, by keeping wrong answers in our response-time analysis, and by the answers of participants. Threats to external validity are caused by neglecting categories of the measurement of programming experience.

5.7.1 Internal Validity

A first threat to validity is caused by the tasks. With other tasks, results may look different. However, we selected tasks representative for the experience level of undergraduate students and with varying difficulty. Thus, more experienced participants should perform better than less experienced participants. Hence, we argue that our task selection is appropriate for our purpose.

Next, we did not correct the response times for wrong answers when looking at the correlations with the answers in the questionnaire. Since we did not see any considerable deviations of response times toward zero and we do not analyze response time further, we sufficiently controlled this threat.

Last, we found that some questions of the questionnaire were difficult to answer, for example, for how many years participants were programming. Thus, the answers of participants might not be completely accurate. However, we believe that answers are a reliable approximation, because we observed that participants thoroughly searched for their answer. Nevertheless, in future work, we can analyze the predictive power of these questions.

5.7.2 External Validity

We did not compare self estimation with all identified ways to measure programming experience. We neglected the categories pretests and unspecified questionnaire, because we do not know what the tests and questionnaires looked like. Thus, it is impossible to test this category. Furthermore, we excluded supervisor assessment, because in our sample, we have no comparable supervisor. We could ask teachers of students to assess their programming experience, but teachers do not know their students nearly as good as supervisors know their professional programmers, so the reliability of teacher estimation is questionable. Despite those, we considered all other categories, so we controlled this threat sufficiently. Nevertheless, in future work, we can also work on comparing self estimation with the other techniques, for example, by asking authors of papers which pretest or questionnaire they used.

Last, the results of our exploratory analysis cannot be generalized without confirmatory analysis based on further experiments. To reduce this threat, we are currently collecting data, so that we can conduct a confirmatory analysis.

5.8 Related Work

In general, related work to ours evaluated possible criteria that can be used to categorize participants upfront. For example, Kleinschmager and Hanenberg analyzed the influence of self estimation, university grades, and pretests on historical data for programming experiments [Kleinschmager and Hanenberg, 2011]. To this end, they analyzed the data of two previously conducted programming experiments with students as participants. They compared self estimation, university grades, and pretests with the performance of participants in the experiments and found that self estimation was not worse than university grades or pretests to categorize participants. These results complement ours, as we did not look into pretests and grades.

Bornat and others compared the performance of students with the performance of professional software developers for non-trivial tasks regarding judgment about factors affecting the lead-time of software-development projects [Bornat et al., 2008]. They found no differences between groups. Thus, classification of participants had no effect on their performance.

Instead of studying single criteria to categorize participants, other lines of research use more complex criteria, such as the Myers-Briggs Type Indicator, an index for estimating the personality of humans [Myers and McCaulley, 1985]. For example, Sfetsos and others study the impact of certain criteria, such as the personality type of participants, on building groups doing pair programming [Sfetsos et al., 2009]. They found that pairs of different personality types are more effective than pairs of the same personality types.

5.9 Summary

To conduct reliable program-comprehension experiments, we need to control for programming experience as a major confounding parameter. Currently, researchers often do not specify their understanding of programming experience or do not consider it at all, which threatens the validity of experiments and makes interpretations across experiments difficult. Thus, we defined the goal to develop a questionnaire to reliably measure programming experience in an easy and cost-efficient way.

Based on a literature survey, we identified 7 categories how researchers measure programming experience: years, education, self estimation, unspecified questionnaire, size, unspecified pretest, and supervisor. Based on these categories, we designed a questionnaire that aims at measuring programming experience.

In a controlled experiment, we evaluated the relationship of each question with the performance of 128 students who solve program comprehension tasks. We found that self estimation is a good indicator for programming experience. Furthermore, results of stepwise regression suggest that programming experience compared to class mates (s.ClassMates) and experience with logical programming (s.Logical) are the most relevant

questions to measure programming experience. Additionally, with an exploratory factor analysis, we extracted a five-factor model describing programming experience.

Thus, with an overview of confounding parameters for program comprehension and a first version of a questionnaire to measure programming experience as the most important parameter, we reduced the effort for planning experiments. Next, we present our tool PROPHET, which supports researchers in conducting and replicating experiments.

Chapter 6

PROPHET: Program-Comprehension Experiment Tool

*This chapter shares content with the ICPC'12 tool demo "Supporting Comprehension Experiments with Human Participants" [Feigenspan and Siegmund, 2012] and the ESEM'11 poster "*PROPHET*: Tool Infrastructure to Support Program Comprehension Experiments" [Feigenspan et al., 2011d].*

The last part of our framework to support conducting comprehension experiments is the tool PROPHET (short for *pro*gram-com*preh*ension *e*xperiment *t*ool). During our work on program comprehension in feature-oriented software development, we found similar, but slightly different requirements to present source code, tasks, and questionnaires to participants. Furthermore, in the literature survey, we found that researchers often implemented tool infrastructures with similar requirements. Thus, instead of developing different tool infrastructures with similar requirements for each experiment, we set out to implement a customizable and extendable tool:

- A tool to fulfill common requirements of program-comprehension experiments that can be extended to address additional requirements.

To fulfill the requirements of a large set of experiments, we again consulted the paper of our literature survey, this time analyzing requirements to conduct comprehension experiments. Based on the identified requirements, we implemented our tool PROPHET. We developed it as stand-alone tool, not integrate it into existing integrated development environments, such as Eclipse, so that we can control the influence of familiarity with tools (cf. Section 4.5.1.2). PROPHET provides only basic functionality to browse through source code, so participants can easily learn how to use it. Furthermore, we focus on text-based material, so experiments on visual programming languages are currently not supported.

In Section 6.1, we describe the most common requirements of comprehension experiments based on a literature survey to give an overview. In Section 6.2, we present our tool

Component	ESE	TOSEM	TSE	ICPC	ICSE	FSE	ESEM	Total (%)
Source-code viewer	26	2	30	24	29	9	18	138 (81.7 %)
Measurement of time	24	2	32	29	28	9	18	142 (84.0 %)
Presenting tasks/ questionnaires	31	2	32	32	37	9	18	164 (97.0 %)
Logging	23	2	32	21	24	9	18	129 (76.3 %)
External tool	22	2	26	15	18	9	17	109 (64.5 %)

Table 6.1: *Requirements for comprehension experiments.*

PROPHET based on the identified requirements and discuss whether and how it fulfills these requirements. This way, we demonstrate the usefulness of our tool. To demonstrate its extensibility, we present and evaluate the extendable plug-in architecture in Section 6.3.

6.1 Requirements for Comprehension Experiments

In this section, we analyze how researchers conduct comprehension experiments. To this end, we again consulted the papers of our literature survey (cf. Chapter 4). We set the selection criteria broader, such that we also included experiments similar to program-comprehension experiments, such as experiments on software inspection (instead of source code, normal text documents are shown). Thus, for both International Conference on Software Engineering and Journal of Empirical Software Engineering, we considered 36 papers. For all other journals and conferences, the number of papers remains unchanged. Thus, we have 169 papers in total for this chapter. To identify requirements of comprehension experiments, we read all papers completely.[1] In Table 6.1, we summarize the identified requirements, including the number of papers per requirement.

In most experiments, authors implemented their own tool infrastructure or used existing tools, such as Eclipse, with or without extensions, or let participants complete the experiment by themselves (e.g., online, at home). Only rarely, authors reused self-made tool infrastructures. Some experiments were even paper based. However, using different tools makes the comparability of experiments difficult, because the underlying tools provide considerably different functionality. For example, Eclipse provides search functions, auto completion, compiling code, and running code. Experimental tools typically provide only limited subsets of these features, and paper-based experiments do not have any of the functionalities. Furthermore, familiarity with tool support can significantly influence the results. Additionally, replication of experiments is difficult, because providing all necessary details (e.g., what kind of search participants could use or in what size/ font/color source code is presented) is often impossible due to space restrictions. Thus, with PROPHET, we aim at having one tool that can be used in typical comprehension experiments and that stores relevant details about the experimental set up.

[1] For most of the papers, we extracted requirements at the same time as extracting confounding parameters. Thus, we rarely read a paper twice.

To have an understanding about the requirements that PROPHET should fulfill, we describe them in detail, including example studies.

6.1.1 Source-Code Viewer

In $82\,\%$ of the experiments, independent of the focus, source code was presented to participants, either just as fragments or as complete project. For example, in an experiment by Sfetsos and others [2009], participants should work in groups on maintenance tasks, and the effectiveness of groups was measured. Ceccato and others [2009] let participants implement source code based on decompiled obfuscated systems using Eclipse. In another experiment, source code was presented on paper [Bunse, 2006].

Thus, PROPHET should provide an according customizable source-code-viewing component, such that experimenters can selected whether source code is shown with or without syntax highlighting, with or without line numbers, can be editable or not, and provide a search or not.

6.1.2 Measurement of Time

In $84\,\%$ of the reviewed experiments, time was measured, either because of time constraints for a session, or to analyze how long participants needed to complete tasks or an experiment. For example, El-Attar and others [2009] measured the time participants needed to solve a task, and the time constraint was large enough to not let participants feel time pressure. Och Dag and others [2006] measured the time participants needed to consolidate requirements, so that the performance of participants could be analyzed in terms of requirements per minute. Experimenters instructed participants to stop when a defined amount of time had passed, and participants had to write down the time and number of analyzed requirements.

To support experimenters in measurement of time, PROPHET should automatically log the time participants need to solve a task or answer a question.

6.1.3 Presenting Tasks and Questionnaires

In most of the experiments ($97\,\%$), tasks were used to measure program comprehension or questionnaires were used to assess participants' opinion. In some experiments, participants were allowed to go back to previous tasks and questions. For example, Binkley and others [2009] let participants detect camelCase and under_score style identifiers and presented tasks subsequently on screen. Matthijssen and others [2010] used tasks and questionnaires to analyze how tool support helps to understand the code of Ajax applications.

Thus, PROPHET should show tasks and questionnaires to participants. Furthermore, PROPHET should allow the experimenter to decide whether participants are allowed to go back to previous tasks.

6.1.4 Logging

In 76 % of the experiments, the actions of participants were logged (including the answers) and used to analyze the behavior. For example, de Alwis and others [2007] logged the behavior of participants using an Eclipse plug in to evaluate how tools can help to complete software evolution tasks. Ko and Uttl [2003] video taped participants and captured their screen during completing tasks. Sharif and others logged the eye movements of participants with an eye tracker [Sharif and Maletic, 2010].

Thus, there are different ways to log the behavior of participants, of which PROPHET should at least support the basic data. We omit audio-, video-, and eye-tracking data in the first implementation, because they are not often logged and require additional hardware, which often is not available.

6.1.5 External Tools

About two third of the experiments (65 %) used external tools, which means that tools besides the actual experimentation environment are used. For example, LaToza and Myers [2010] allowed participants to use Windows Notepad to take notes. Dekel and Hersleb [2009] evaluated how programmers can be motivated to read the documentation of API functions more thoroughly. In addition to source code, participants viewed API specifications in a web browser.

Thus, PROPHET should support experimenters in specifying tools that participants are allowed to use and which should be called from within PROPHET.

6.2 PROPHET

Having identified typical requirements of comprehension experiments, we can present our tool PROPHET in this section, which we implemented in Java. Furthermore, we discuss how it supported us with our experiment regarding programming experience (cf. Chapter 5) to show its usefulness. PROPHET has two views: In the *experimenter view*, experimenters specify the settings, and in the *subject view*, participants see the material and tasks.

6.2.1 Experimenter View

In the *experimenter view*, we provide a user interface for experimenters that allows them to customize the experimental setting according to their needs. In Figure 6.1, we show a screenshot of the *experimenter view*. On the left side, we provide an overview of existing tasks. Categories group similar tasks. For convenience, tasks or categories can be copied and pasted, including all specified settings, so that similar tasks can be efficiently created.

For each category/task, there are four different tabs, which we discus from left to right. The content of the tab *Editor* allows experimenters to create descriptions of tasks and questions as HTML code. For convenience, we provide templates for commonly used HTML elements, such as font type (e.g., bold, italic) and forms (e.g., text field, radio

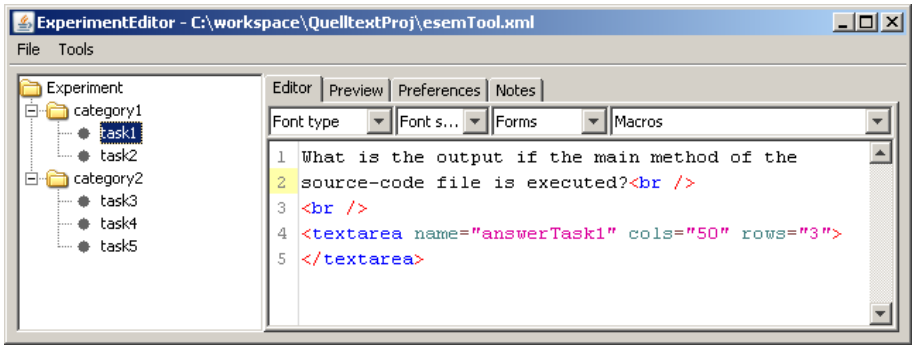


Figure 6.1: *Screenshot of* experimenter view *(*Editor *tab).*

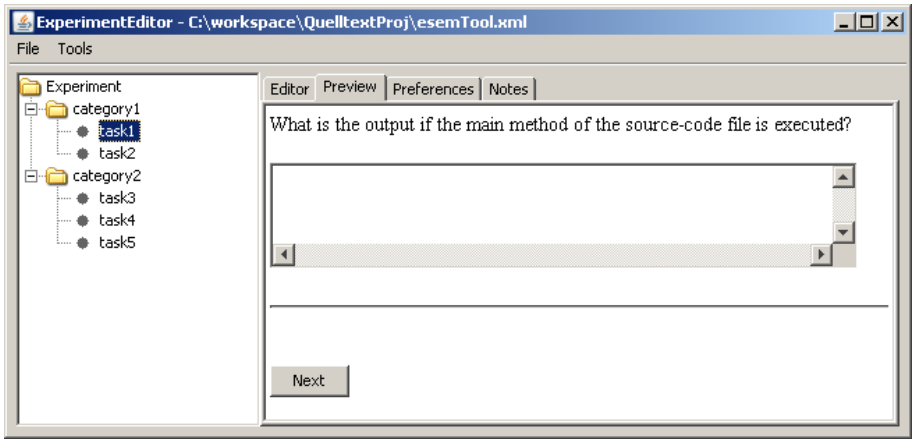


Figure 6.2: *Screenshot of the* Preview *tab.*

button). Furthermore, experimenters can define macros for often used combinations of HTML elements.

The tab *Preview* in Figure 6.2 shows a preview of how participants see the description of a task. In addition to the specified task or question, PROPHET automatically adds a horizontal line and a button "Next", with which participants start the next task or question.

The tab *Preferences* in Figure 6.3 shows numerous customizing options to present source code and tasks to participants. We describe the most interesting options in more detail. When the code viewer is activated (check box "Activate code viewer"), we can customize it, for example:

(1) Define a folder of which the contents are shown in the *package explorer* in the *subject view* (Section 6.2.2).
(2) Define a file that is displayed when a task begins.
(3) Choose whether source code is editable by participants.
(4) Choose what behavior of participants we log.
(5) Choose whether participants see line numbers.
(6) Decide whether participants can turn on and off line numbers.
(7) Choose whether participants can use the search (local, global, both, regex).

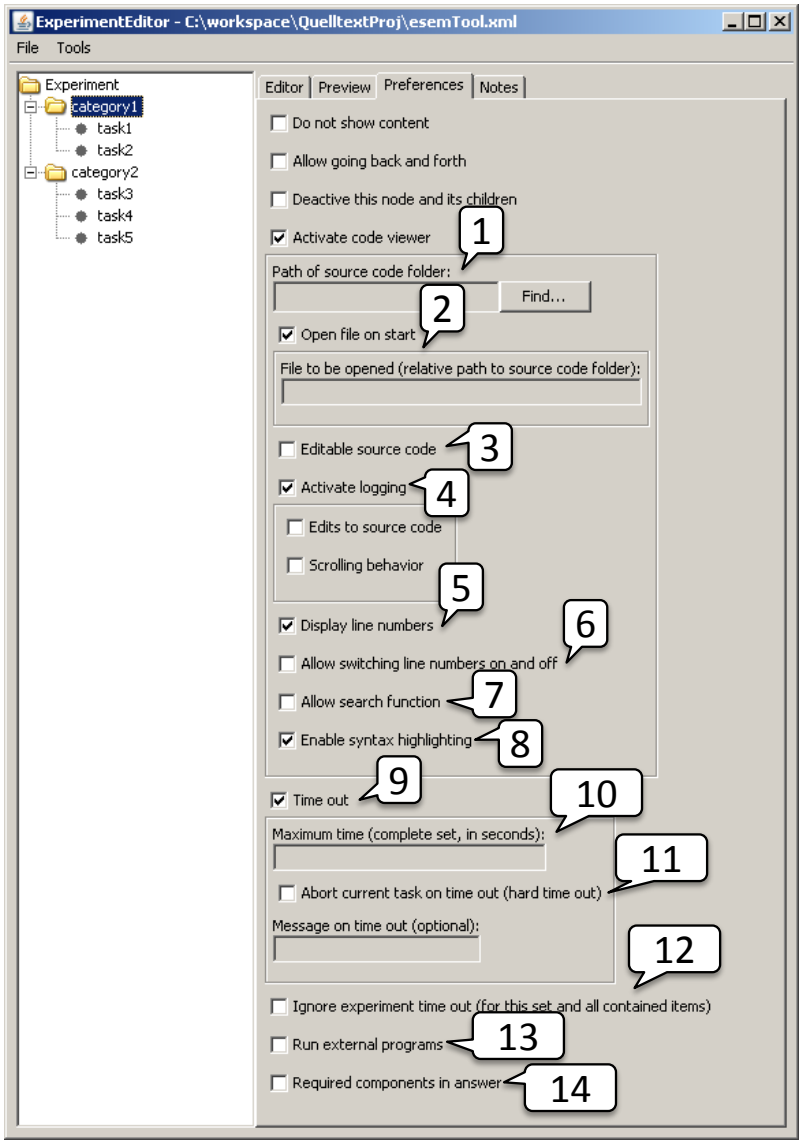


Figure 6.3: *Screenshot of the* Preferences *tab. The numbers refer to options we explain in detail.*

(8) Choose whether source code is displayed with syntax highlighting (for several languages).

Thus, the *source-code viewer* has several customizing options regarding how participants see source code. In our experiment, we presented source code with syntax highlighting and line numbers. Furthermore, participants should not modify source code, so it was not editable. We did not provide a search feature, because the source code was small enough to get an overview without search function.

When selecting to log data, PROPHET creates a folder for each participant during the experiment execution and stores all specified data as XML. In our experiment, we logged the answers of participants and how they opened and selected files.

Furthermore, we can define a time out at which the task is aborted (9). We can define a maximum time for a complete category (10) and additionally decide whether participants are allowed to finish the currently displayed task (soft time out) or not (hard time out, 11). Additionally, we can ignore a time out that may be set for a complete experiment (12). This is useful for debriefing questionnaires, which participants complete after they finished experimental tasks. In our experiment, we defined a soft time out, so that participants could finish a task. Furthermore, we showed a debriefing questionnaire after the time out, for which we selected the option to ignore the time out of the complete experiment.

Moreover, we can specify external programs that participants are allowed to use (13), which we did not need for our experiment.

In addition to defining settings for categories or tasks, we can define settings for a complete experiment. In Figure 6.4, we show a screenshot with the options. The most interesting option here is to send an e-mail. If this option is selected, the logged files will be automatically zipped and sent via e-mail at the end of the experiment, without requiring any interaction from participants. The sender and receiver mail addresses can be specified by the experimenter. We provide this option, because collecting log files per hand from each participant is tedious and error prone (e.g., we experienced in a previous experiment that we forgot the data of some participants). Another way would be to send the data to a database, but that requires additional software, such as database drivers. In our experiment, we used this option without any problems.

The last tab *Notes* contains an empty editor. It allows experimenters to take notes for a task, for example, deviations during the conduct of an experiment. This option proved useful for one experiment (described in Section 8.2), in which not the experiment designer, but a colleague at the according university conducted the experiment.

PROPHET stores the experiment, including all customizations and notes of experimenters, in an XML file. Researchers who want to replicate an experiment can just reuse the XML file. Thus, we improve the replicability of experiments with PROPHET.

PROPHET provides further options to increase usability, which we describe briefly:

- Export the experiment to PHP to conduct it via Internet in a browser; the hosting server needs to run Apache and MySQL
- Check for duplicate form entries to avoid errors caused by copying and pasting HTML elements
- Export answers and logged data to CSV, such that common statistic tools can read the data

6.2.2 Subject View

After the experiment is specified, it can be executed using the *subject view*. To this end, participants have to double click a jar file, so the working stations needs to run Java. Since nothing is installed, participants do not need administrator rights on the working station. To clearly separate tasks and according source code, the *subject view* consists of two separate windows: a *task viewer* and a *source-code viewer*. In the *task viewer*, the tasks

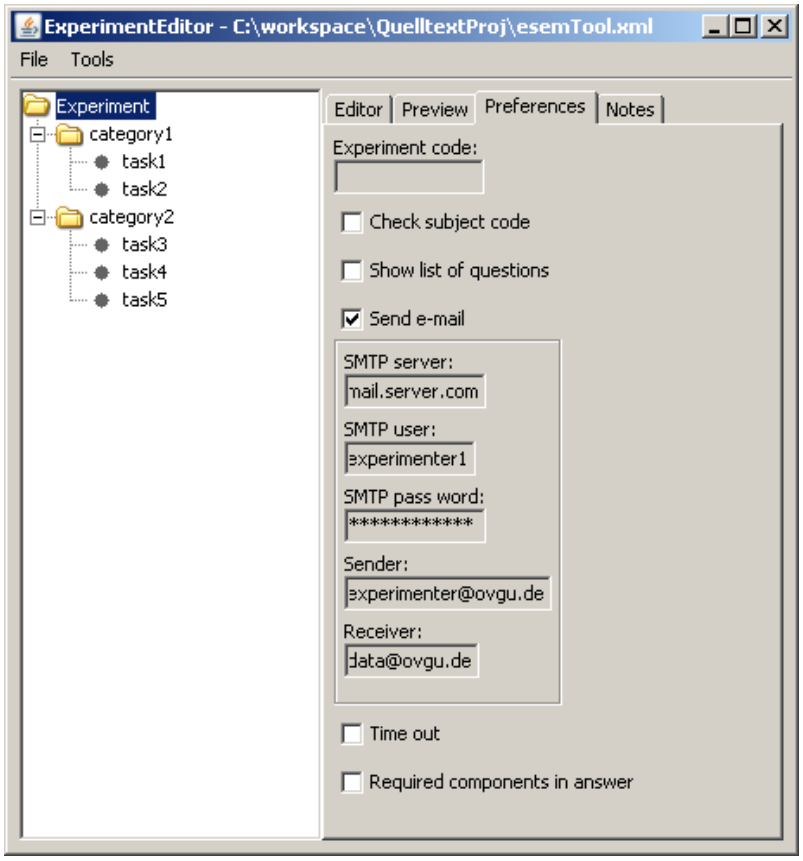


Figure 6.4: *Screenshot of* Preferences *tab for the complete Experiment.*

are displayed as specified in the *Editor* tab of the *experimenter view*. In Figure 6.5, we show a screen shot. In addition to the *Preview* tab of the *experimenter view*, the elapsed time for the current category and complete experiment is shown. Note that an overview of all tasks is not shown on the left side, because we did not select the option (cf. Figure 6.4, check box "Show list of questions").

In the *source-code viewer*, we present source code to participants with the specifications defined in the *Preferences* tab of the *experimenter view*. We show an example of the *source-code viewer* in Figure 6.6, in which line numbers are displayed and syntax highlighting is enabled.

6.2.3 PROPHET for Other Experiments

To evaluate whether PROPHET supports other experiments, we describe how it can help in two papers of the literature review. To have a fair evaluation, we selected one experiment that can be well supported, and another for which PROPHET needs to be adapted. In the first experiment, participants should rate how much they understand of source code [Lawrie et al., 2006]. The authors needed to present source code and provide means to rate it. Both can be done with PROPHET. A time limit was not mentioned, just

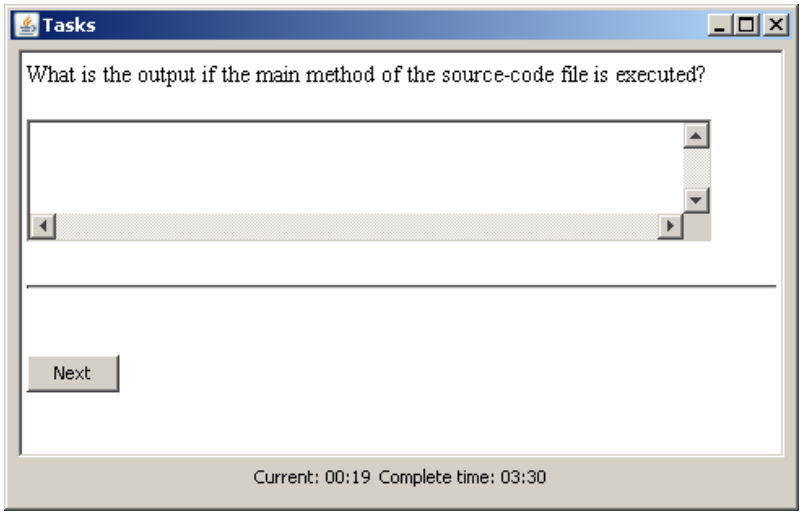


Figure 6.5: Task viewer *of the* subject view *of* PROPHET.

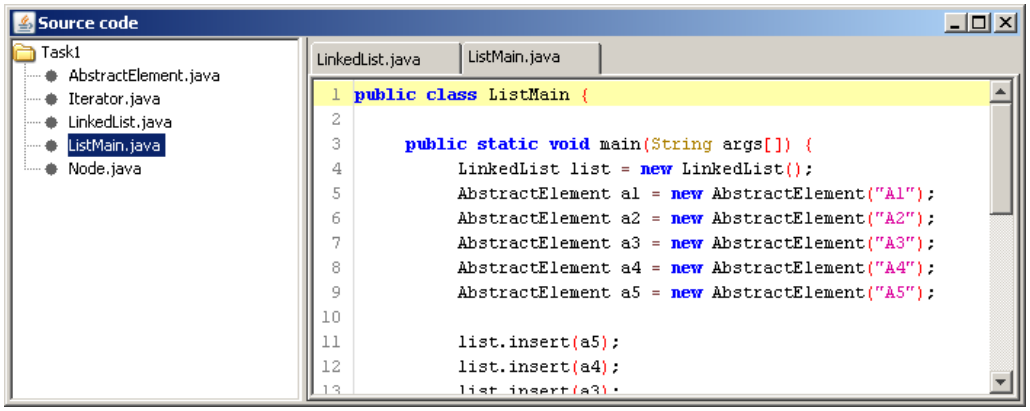


Figure 6.6: Source-code viewer *of* PROPHET.

that the experiment needed to be "completed in a timely manner"; it is possible to use the soft time out (i.e., a task can be finished after time has run out) to support this.

In the second experiment, participants worked with UML diagrams [Genero et al., 2007]. The UML diagrams were presented on paper, and participants should modify them. Currently, PROPHET does not support modifying images; we can only show them. There are two options to conduct this experiment with PROPHET: First, implementing a plug in that allows participants to modify graphics and log their changes. Second, specifying an external program from PROPHET. However, we give the control to the external program and cannot log participants' behavior. Thus, in its current version, PROPHET provides only limited support for experiments in which participants should modify graphics.

Thus, PROPHET fulfills most parts of the identified requirements. Current limitations are that source code cannot be compiled and executed, audio, video, or eye-tracking

```
1  public interface PluginInterface {
2
3      //Delivers settings components shown in the settings tab of the experiment editor.
4      public SettingsComponentDescription getSettingsComponentDescription(
5            QuestionTreeNode node);
6
7      //Called once when the experiment viewer is initialized.
8      public void experimentViewerRun(ExperimentViewer experimentViewer);
9
10     //If any plugin denies the currentNode to be entered
11     //(e.g., because of a timeout), it will be skipped.
12     public boolean denyEnterNode(QuestionTreeNode node);
13
14     //The node entered
15     public void enterNode(QuestionTreeNode node);
16
17     //A message shown to the participant to indicate what needs to be
18     //done to accept finishing this node (e.g. enter a needed answer)
19     public String denyNextNode(QuestionTreeNode currentNode);
20
21     //The node to be exited
22     public void exitNode(QuestionTreeNode node);
23
24     //A unique name for the plugin.
25     public String getKey();
26
27     //A message shown to the participant at experiment's end
28     public String finishExperiment();
29 }
```

Figure 6.7: *Source code of the interface to define new plug ins.*

data cannot be logged, and images cannot be modified, which is often necessary for UML experiments. Thus, experiments using such data (e.g., Sharif and Maletic [2010], Ko and Uttl [2003]) or require participants to modify images (e.g., Genero et al. [2007]) are currently not supported.

To address new requirements, we implemented PROPHET as plug-in architecture, which we discuss next.

6.3 Extensibility

To support extensibility, we use an approach based on plug ins [Clayberg and Rubel, 2006]. To implement additional functionality, experimenters need to implement a plug in. A plug in must implement PROPHET's plug-in interface, which we show in Figure 6.7. Available plug ins are displayed in the *experimenter view* (e.g., syntax highlighting, run external programs), and if an experimenter selects a plug in, it defines the customization options it provides.

In Line 4, we show the interface to the *experimenter view* of PROPHET. The type `SettingsComponentDescription` is used to describe possible settings of a plug in. To illustrate this mechanism, we show an excerpt of the plug in to hide a task (cf. Figure 6.3, check box "Deactivate this node and its children") in Figure 6.8. In Line 2, it implements

```
1  class InactivityPlugin implements PluginInterface{
2     public SettingsComponentDescription getSettingsComponentDescription(
3           QuestionTreeNode node) {
4        if (!node.isExperiment()) {
5           return new SettingsComponentDescription(SettingsCheckBox.class, KEY,
6              "Deactive this node and its children");
7        } else {
8           return null;
9        }
10    }
11
12 //...
13    public boolean denyEnterNode(QuestionTreeNode node) {
14       return Boolean.parseBoolean(node.getAttributeValue(KEY));
15    }
16 }
```

Figure 6.8: *Source code of a plug in to deactivate certain tasks.*

the method declared in PROPHET's plug-in interface (Line 4 in Figure 6.7) and introduces the according check box.

In addition to extending PROPHET with plug ins, we can also extend plug ins with plug ins. That is, plug ins can use or define other plug ins. For example, the *source-code-viewer* plug in uses sub plug ins, such as the search, which in turn define further plug ins, such as global search or regular-expression search. For the source-code viewer to use plug ins, it defines an own interface that has to be implemented by a sub plug in. One interface method is used to delegate customization options for experimenters to the plug-in structure of our tool, so that options of sub plug ins can also be visualized and manipulated in the *experimenter view*. Furthermore, we define interface options to modify contents of the *Preferences* tab (e.g., to show line numbers).

We experienced the extensibility of PROPHET during planning our experiment regarding programming experience, because we found that we needed requirements currently not implemented. Specifically, we implemented the PHP export, time limits, and mail delivery as plug ins. For none of the plug ins, we had to adapt the underlying implementation.

6.4 Related Work

SESE is a support environment for conducting experiments to evaluate software-engineering technologies [Arisholm et al., 2002]. It supports defining the experiment, recruiting participants, running the experiment, monitoring the experiment, and collecting the results. The focus of SESE is different from PROPHET, in that it supports the conduct of large-scale experiments with focus on external validity. This is possible with our tool using the PHP export, but we target laboratory experiments. Furthermore, our tool is designed to be extendable, such that new functionality can be added without adapting the code base.

EFS Survey[2] is a proprietary web-based survey system. It supports creating and conducting online questionnaires based on HTML pages. Like our tool, HTML forms are used to record answers of participants. Furthermore, stored data can be exported to several formats, including CSV and XLS. However, the focus of EFS Survey lies on conducting questionnaires and recruiting participants, not on assessing the performance of participants. Our tool supports questionnaires as well, but does not provide as much specialized functionality.

Do and others describe an infrastructure for evaluating software-testing mechanisms [Do et al., 2005]. The tool is based on a literature survey of papers describing testing procedures and aims at supporting experimentation on testing techniques. Similar to our tool, it is designed to be extendable. However, it does not focus on supporting experiments on comprehension, but on testing techniques.

6.5 Summary

In the context of comprehension experiments, similar, but slightly different requirements exist to present source code, tasks, and questionnaires to participants. In this chapter, we set the goal to develop a tool that meets these requirements and that can be extended to address additional requirements.

To fulfill our goal, we developed PROPHET, which addresses common requirements for comprehension experiments, which we identified based on a literature survey. Additionally, PROPHET has a plug-in architecture, so that currently not fulfilled requirements can be addressed.

We evaluated PROPHET by showing how the implemented components supported us during planning, conducting, and replicating experiments and by discussing how other experiments of the literature survey could be supported. We also encountered limitations, such that PROPHET does not support modifying images (e.g., UML models), compiling and executing source code, or logging video, audio, or eye-tracking data. However, we demonstrated PROPHET's extensibility, such that future plug ins can address current limitations.

Thus, with PROPHET, we provide a useful, extendable tool to support researchers in planning, conducting, and replicating experiments.

This concludes Part I of our thesis. We showed that software measures are not suitable for measuring program comprehension. Instead, we should conduct controlled experiments with human participants to observe this internal cognitive process. To reduce the effort for planning, conducting, and replicating experiments, we presented a list of confounding parameters for program comprehension, developed a preliminary version of a questionnaire to measure programming experience, the most important confounding parameters for program comprehension, and implemented our tool PROPHET.

In Part II, we present controlled experiments we conducted to evaluate program comprehension in feature-oriented software development. To show the applicability of our framework, we discuss how it supported us for each experiment. Since we developed the

[2] http://www.unipark.info/63-0-efs-survey.htm

framework in parallel to conducting the experiments, we could not use all components of the framework for all experiments. In such cases, we discuss how the framework could have helped.

Part II

Conducting Comprehension Experiments

Chapter 7

Using Background Colors to Escape the #ifdef Hell

This chapter shares content with ESE'12 paper "Do Background Colors Improve Program Comprehension in the #ifdef Hell?" [Feigenspan et al., 2012b], the EASE'11 "Using Background Colors to Support Program Comprehension in Software Product Lines [Feigenspan et al., 2011b]", the IET'12 paper "Supporting Program Comprehension in Large Preprocessor-Based Software Product Lines", the FOSD'09 paper "How to Compare Program Comprehension in FOSD Empirically–An Experience Report" [Feigenspan et al., 2009], and the ICPC'10 paper "Visual Support for Understanding Product Lines" [Feigenspan et al., 2010].

Having developed the framework in Part I, we can address our second goal to create a knowledge base regarding the effect of feature-oriented software development on program comprehension. To this end, we conducted a series of controlled experiments to address the goal of this chapter:

- Recommendation about the use of background colors to improve program comprehension in preprocessor-based software.

By improving how developers comprehend preprocessor-based code, we make an important contribution regarding time and cost of real-world software development, because preprocessors are often used in industry to implement variable code. Furthermore, they will still be widely used at least in the medium-term future, because introducing new techniques in industry is time-consuming, especially when large amount of legacy applications are involved.

In Section 7.1, we describe why background colors can improve program comprehension. This way, we enable others to understand our experimental set up. In Section 7.2, we discuss how the experiments build up on each other to increase external validity step by step. Then, we describe all three experiments in Sections 7.3 to 7.5 to enable researchers to understand our conclusions and replicate our experiments. In Section 7.6, we integrate the results of all three experiments to a knowledge base regarding the effect of background colors in preprocessor-based software on program

```
1  static int __rep_queue_filedone(dbenv, rep, rfp)
2     DB_ENV *dbenv;
3     REP *rep;
4     __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6     COMPQUIET(rep, NULL);
7     COMPQUIET(rfp, NULL);
8     return (__db_no_queue_am(dbenv));
9  #else
10    db_pgno_t first, last;
11    u_int32_t flags;
12    int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14    DB_MSGBUF mb;
15 #endif
16 // over 100 lines of additional code
17 #endif
18 }
```

Figure 7.1: *Code excerpt of Berkeley DB.*

comprehension. In Section 7.7, we apply this knowledge base to create *FeatureCommander*, a prototype of an integrated development environment, which consistently uses background colors. In Section 7.8, we discuss threats to validity for all three experiments, because similar threats exist. To show the applicability of our framework, we discuss how it could have supported us with these experiments in Section 7.9.

7.1 Background: Why Background Colors?

To understand why we use background colors to annotate feature code, we introduce typical problems that occur when using ifdef directives. Then, we explain why background colors can solve these problems.

7.1.1 Welcome to the #ifdef Hell

We illustrate the use of ifdef directives in Figure 7.1 with an excerpt of Berkeley DB[1]. We used a real-world example instead of the stack example of Section 2.3 to better demonstrate the problems when using ifdef directives. Identifying code fragments annotated with ifdef directives can be problematic, especially when

- ifdef directives are fine grained
- ifdef directives are scattered
- ifdef directives are nested
- long code fragments are annotated

These properties and combinations thereof often occur in preprocessor-based software [Liebig et al., 2010, 2011].

First, ifdef directives can be "hidden" somewhere within a single statement at a fine grain. For example, a programmer may annotate a variable or a bracket. Such anno-

[1] http://www.oracle.com/technetwork/database/berkeleydb

tations are difficult to locate, because they can hardly be distinguished from "normal" source code. Another problem is that fine-grained annotations can lead to syntactic errors after preprocessing, because a closing bracket may be annotated, but not the corresponding opening one. Tracking these errors at source-code level is difficult, because both brackets are visible in the source code.

Second, ifdef directives are typically scattered across the code base. In Figure 7.2, we illustrate this problem with a source-code excerpt from the Apache Tomcat web server, showing session management. Implementing an optional session-expiration mechanism involves the addition of code and ifdef directives in many locations. The red background color illustrates the scattering of feature *Session expiration* over the complete implementation of session management, which makes implementing and tracing this feature a tedious and error-prone task. A developer must take into account all affected modules when keeping track of the *Session-expiration* feature.

Third, ifdef directives can be nested. For example, in Figure 7.1, Lines 13 to 15 are defined within another ifdef directive, starting in Line 5. It might not be difficult to keep track of a nesting level of two (as in this case), which is typical for most projects. However, in practice, nesting levels of up to 24 may occur [Liebig et al., 2010].

Fourth, long code fragments can be annotated, as indicated in Figure 7.1: Line 16 states that over 100 additional lines of code occur, after which the according #endif of the #ifndef in Line 5 occurs. To keep track of this fragment of feature code, a developer typically has to scroll and, thus, keep in mind which code fragments belong to the according feature and which do not. A surrounding annotation might not be visible from the source-code excerpt shown in an editor.

7.1.2 Stairway to Heaven?

To escape the "#ifdef hell", several approaches were developed that aim at improving the readability of preprocessors, for example, by hiding selected feature code such as in the Version Editor [Atkins et al., 2002], CViMe [Singh et al., 2006], or C-CLR [Singh et al., 2007], or by annotating features with colors, such as in Spotlight [Coppit et al., 2007] (with vertical bars next to the code editor), NetBeans (one background color for all features), or CIDE [Kästner et al., 2008].

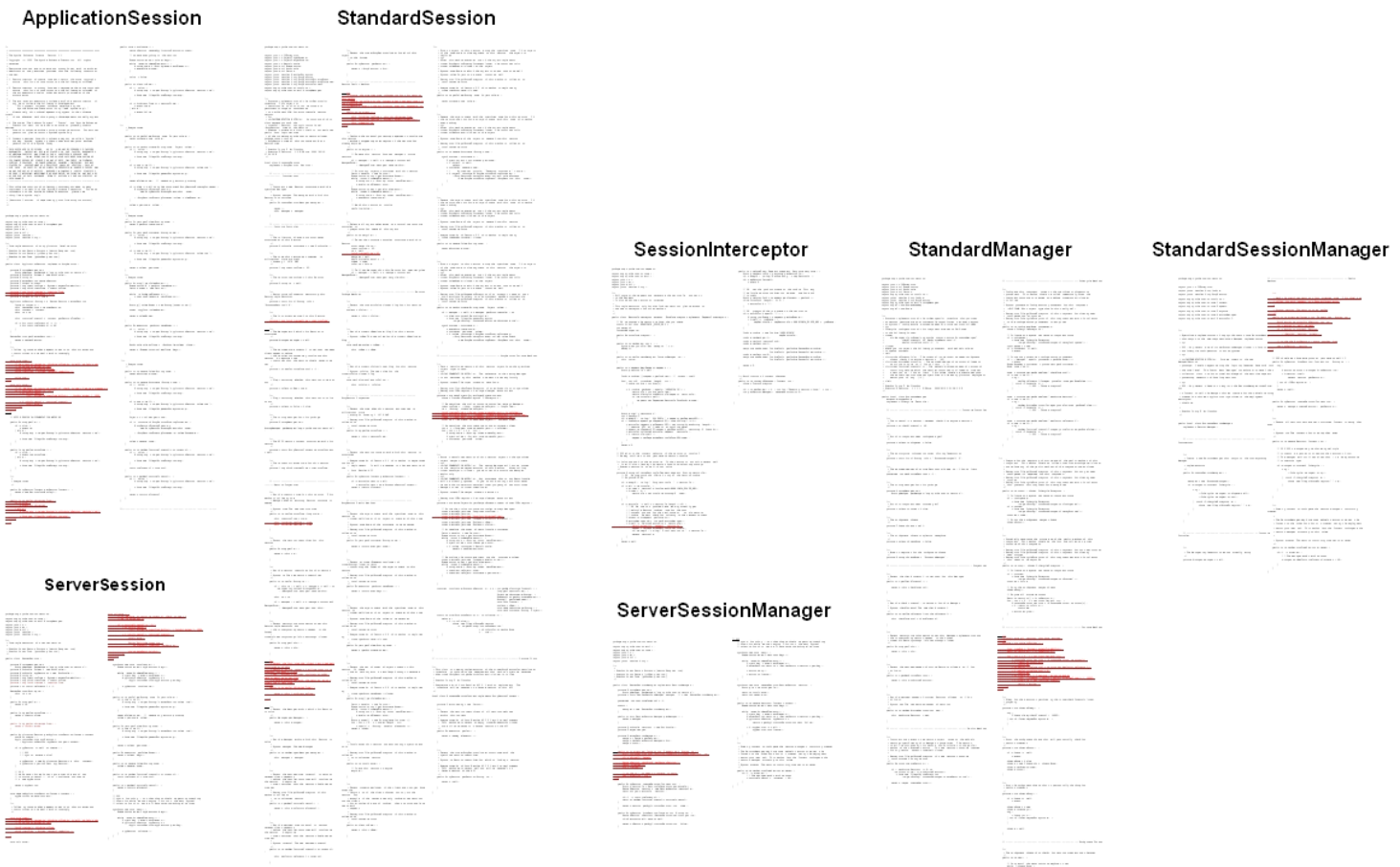


Figure 7.2: *Apache Tomcat source code illustrating scattering of session-expiration source code. This figure is from a tutorial on AspectJ: http://kerstens.org/mik/publications/aspectj-tutorial-oopsla2004.ppt*

```
1  static int __rep_queue_filedone(, dbenv, rep, rfp)
2     DB_ENV *dbenv;
3     REP *rep;
4     __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6     COMPQUIET(rep, NULL);
7     COMPQUIET(rfp, NULL);
8     return (__db_no_queue_am(dbenv));
9  #else
10    db_pgno_t first, last;
11    u_int32_t flags;
12    int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14    DB_MSGBUF mb;
15 #endif
16 // over 100 lines of additional code
17 #endif
18 }
```

Figure 7.3: *Excerpt of Berkeley DB with background colors to highlight feature code. Lines 5 to 16 are yellow, Lines 12 to 14 orange.*

In Figure 7.3, we illustrate how background colors can be used to annotate source code. All annotated source-code lines are displayed with a background color. Code of feature *HAVE_QUEUE* (Lines 5 to 16) is annotated with yellow. The according else directive (Line 8) has the same color, because the according annotated code is also relevant for this feature. Code of feature *DIAGNOSTIC* (Lines 12 to 14) is annotated with orange. In this example, we see how we deal with nested code: We display the background color of the inner feature *DIAGNOSTIC*, which is orange. In an early prototype, we blended the colors of all features that are nested. However, this way we introduced more colors than necessary and make distinguishing code of different features more difficult. Additionally, with a deeper nesting level it becomes difficult to recognize all involved features, because the blended colors would result in a shade of gray.

With background colors, we use a *highlighting* technique that supports users in finding relevant information [Fisher and Tan, 1989; Tamborello and Byrne, 2007]. Highlighting emphasizes objects that users might look for, such as menu entries or certain code fragments. It can be realized with different mechanisms, such as blinking or moving an object. In past work, colors have been shown to be effective for classifying objects into separate categories and to increase the accuracy in comprehension tasks [Chevalier et al., 2010; Fisher and Tan, 1989; Ware, 2000].

The benefit of colors compared to text-based annotations is twofold: First, background colors clearly differ from source code, which helps distinguish feature code from base code. Second, humans process colors preattentively[2] and, thus, considerably faster than text [Goldstein, 2002]. This allows a programmer to identify feature code at first sight and distinguish code of different features. As a consequence, a programmer should be able to get an overview of a software system considerably faster.

Based on the comparison of the code fragments in Figures 7.1 and 7.3, we could intuitively argue that one approach is better than the other or that both should be combined.

[2]Preattentive perception is the fast recognition of a limited set of visual properties [Goldstein, 2002].

GQM	Experiment 1	Experiment 2	Experiment 3
Analyze	Background colors	Background colors	Background colors
Purpose	Evaluation	Evaluation	Evaluation
With respect to	Program comprehension	*Use of opportunity to switch*	Program comprehension
Point of view	Developer	Developer	Developer
Context	Medium preprocessor-based system	Medium preprocessor-based system	*Large preprocessor-based system*

Table 7.1: *Description of all three experiments using the goal-question-metric approach. We emphasized differences of experiments.*

For example, one could argue that colors are distracting [Fisher and Tan, 1989] or do not scale for large software systems, or colors do improve program comprehension due to preattentive perception [Goldstein, 2002]. However, since program comprehension is an internal cognitive process, we can only assess it empirically. In the remainder of this chapter, we describe our experiments to evaluate whether background colors have a benefit for program comprehension.

7.2 Family of Experiments

Each experiment focuses on a different facet of background-color use. In short, they address the following questions:

- Can colors improve program comprehension? (Section 7.3)
- Do participants use colors? (Section 7.4)
- Do colors scale to large systems? (Section 7.5)

By combining the results of all three experiments, we aim at providing a deeper understanding of the influence of background colors on program comprehension in preprocessor-based software. For a better overview, we describe each experiment using the goal-question-metric approach in Table 7.1 [Basili, 1992].

The focus of the first and third experiment lies on program comprehension, whereas the focus of the second experiment lies on the behavior of participants, that is, how participants use the opportunity to switch between background colors and preprocessor directives. The context of the first and second experiment is medium-sized systems, whereas the last experiment uses a large system. In all other criteria of the goal-question-metric approach, the experiments are the same. Due to this small delta between the experiments and the combination of their results, we ensured high internal and external validity.

Furthermore, we avoided threats to validity caused by learning or maturation effects by recruiting different participants for the first two experiments. In the third experiment,

Context	Description
Objective	Evaluate whether background colors improve program comprehension in preprocessor-based software
Material	5th release of MobileMedia in two versions: Java ME with Antenna, Java ME with background-color annotation
Participants	43 undergraduate students from the University of Passau
Tasks	Static tasks (locating feature code); maintenance tasks (locating bugs)
Execution	One computer lab; 17" TFT; browser w/o search to present source code
Analysis	Correctness of answers and response time
Result	Colors speed up program comprehension in static tasks; no effect or slow down for maintenance tasks

Table 7.2: *Experiment 1 in a nutshell.*

one participant of the second experiment also took part. However, since we had different research hypotheses and different material, no learning or maturation effects occurred.

7.3 Experiment 1:Can Colors Improve Program Comprehension?

In this section, we present the design of our first experiment. Since we described it in detail in our master's thesis, we present only a summary of this experiment [Feigenspan, 2009]. We start with an overview of the most important information in Table 7.2. This was the first experiment regarding colors, so we started with a medium-sized software system with only four features. The rationale was that we assumed a benefit to occur in small settings with only a few features, because preattentive perception is limited to only few items. Thus, if there is a benefit, we most likely find it here.

As material, we used the fifth release of MobilbeMedia (cf. Section 2.4). From the original source code annotated with ifdef directives (referred to as *ifdef version*), we created a version that uses background colors (referred to as *color version*) instead of ifdef directives. We did not combine background colors and ifdef directives, because there is no prior empirical work regarding the effect of colors on program comprehension in the context of preprocessor-based software on which we can base our experiment. Thus, to not confound the effect of text and background colors, we explicitly compare the two extremes of pure textual annotations versus pure graphical annotations with background colors in this first experiment.

For code fragments that were shared by the features *SMSFeature* and *CopyPhoto* (see Figure 7.3 for an example of shared/nested code), we selected a separate color. We selected the following bright and clearly distinguishable colors as background colors:

- *SMSFeature*: red (rgb: 255 − 127 − 127)
- *CopyPhoto*: blue (rgb: 127 − 127 − 255)
- *Favourites*: yellow (rgb: 255 − 255 − 127)
- *CountViews*: orange (rgb: 255 − 191 − 127)
- *SMSFeature* & *CopyPhoto*: violet (rgb: 170 − 85 − 170)

Task	Bug description	Feature
S1	Which features implement code in which classes?	—
S2	In which classes do features CopyPhoto and SMSFeature share code (i.e., interact)?	*CopyPhoto*, *SMSFeature*
M1	If pictures should be sorted by views, they are displayed unsorted anyway.	*CountViews*
M2	When a picture is displayed, the variable that counts the views is not updated.	*CountViews*
M3	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.	*Favourites*
M4	If during sending a picture the according picture is not found, a NullPointerException is thrown.	*SMSFeature*

Table 7.3: *Overview of Tasks.*

To measure program comprehension, we designed six tasks (plus a warming up task to let participants familiarize with the setting). We had two static tasks, in which participants should locate features and locate all occurrences of two interacting features. The remaining four tasks were maintenance tasks. For a better overview, we present all tasks in Table 7.3. For each task, we provided the according feature to participants, so that they could concentrate on feature code (which is highlighted with background colors). We expect a speed up in the comprehension process especially in the first two tasks, because participants only need to look for the presence of a color or recognize a certain color. Thus, due to preattentive perception, participants should be faster than without colors. For maintenance tasks, we do not expect a benefit, because we believe that locating feature code is only a minor part of the comprehension process. Furthermore, we do not expect a difference in correctness of solutions, because both versions provide the same amount of information.

We divided our sample of 43 students into two homogenous groups regarding programming experience (based on a preliminary version of the questionnaire in Chapter 5). One group worked with the color version, the other group with the ifdef version of MobileMedia. We conducted the experiment in a computer lab with 50 seats, 17" TFT monitors, Linux as operating system, and Mozilla as browser to present the source code, tasks, and forms to enter the solutions. Few deviations occurred, such that participants arrived late and were seated in another room to not disturb the others. To not jeopardize the anonymity of participants, we did not trace the deviations to the participants. Our sample is large enough to compensate for these deviations.

We found that, for static tasks, participants of the color group were significantly faster (on average by $34\,\%$). Thus, colors provide a benefit as expected. For maintenance tasks, we found a significant difference for the last task, such that participants of the color group were significantly slower (by $37\,\%$) than participants of the ifdef group. This is in contrast to our expectation that there should be no difference. We believe this is caused by the annotation: In this task, relevant source code was annotated entirely with red, which

may have caused visual fatigue, such that participants had to rest their eyes more often. This explanation is supported by comments of participants, stating that the red color for this task was annoying. For correctness, we found no significant differences as expected.

Furthermore, we analyzed the opinion of participants regarding background colors. Almost all participants who worked with the ifdef version estimated that they would have performed better with the color version, whereas participants who worked with the color version thought they would have performed worse with the ifdef version. This holds even in the last task, in which participants of the color group were significantly slower than participants of the ifdef group. Hence, we found a strong effect regarding participants' estimation that is in contrast to participants' actual performance. Some participants of the color group noted that they were happy to get to work with it, whereas some participants of the ifdef group wished they had worked with the color version. This could explain the difference in estimating the performance, because some participants liked the color version better, which they reflected to their performance.

7.4 Experiment 2: Do Participants Use Colors?

The results of our first experiment indicate that participants like the color idea, but that carelessly chosen colors are disturbing (as some participants noted) and can slow them down. This indicates that different kinds of annotations might be suitable for different tasks, and we should offer developers the opportunity to switch between them as needed for the task at hand. Hence, in a follow-up experiment, we evaluated whether developers would use the option to switch between background colors and ifdef directives. Our results indicate that participants prefer background colors, even if they slow them down. We had the chance to perform this experiment twice, first in 2010, then we replicated it with different participants with similar background in 2011. Hence, we have two instances of our second experiment. Since both instances differ only in few details, we describe them together, and present information about the replication in angle brackets, ⟨like this⟩.

For better overview, we present important information of this experiment in Table 7.4.

7.4.1 Objective and Material

The goals of the follow-up experiment differ from the first experiment: Rather than examining the effect of background colors on program comprehension, we evaluate whether and how participants use the chance to switch between ifdef directives and colors as annotations. Based on the insights from the first experiment, we state the following hypothesis:

> **RH1**: For locating feature code, participants use colors, while for closely examining feature code, participants use ifdef directives.

We used the same source code and background colors as for our first experiment. To present the source code, we implemented a tool similar to the browser setting. In

Context	Description	Section
Objective	Evaluate whether participants use background colors	7.4.1
Material	5th release of MobileMedia in two versions: Java ME with Antenna, Java ME with background-color annotation	
Participants	10 ⟨10⟩ graduate students from the University of Magdeburg	7.4.2
Tasks	Static tasks (locating feature code); maintenance tasks (locating bugs)	7.4.3
Execution	One computer lab; 17" TFT; tool w/o search with two buttons to switch between annotations to present source code	7.4.4
Analysis	Behavior of participants regarding use of background colors	7.4.5
Result	Participants use colors in the beginning, but reduce their switching behavior	7.4.6

Table 7.4: *Experiment 2 in a nutshell.*

addition, we provided two buttons to enable participants to switch easily between color version and ifdef version. Our tool logged each button click with a time stamp, such that we can analyze the behavior of participants.

7.4.2 Participants

We asked students of the 2009 ⟨2010⟩ course *Contemporary Programming Paradigms* at the University of Magdeburg, Germany to participate, which was one of multiple alternative prerequisites to pass the course. The course was very similar to that of our first experiment, so the background of students was comparable. Additionally, two graduate students who attended that course in the fall term 2008 volunteered to participate, as well. Altogether, our sample consisted of 10 ⟨10⟩ participants. One week before the experiment, we administered the same preliminary version of the programming-experience questionnaire as in the first experiment. None of the participants was color blind, and 1 ⟨0⟩ was female.

7.4.3 Tasks

We used the same tasks as for our first experiment (cf. Table 7.3, including a warming-up task (W0)). However, we changed the order of the tasks to M1, M3, S1, M4, M2, S2. We alternated static and maintenance tasks, such that we could observe whether participants actually switch between both representations in line with our hypothesis.

7.4.4 Experiment Execution

We booked a room with 16 seats. All computers had Windows XP as operating system and 17" TFT screens. The experiment took place in January 2010 ⟨January 2011⟩ in Magdeburg instead of a regular lecture session. We gave the same introduction as for the first experiment, with the addition that we showed how participants could switch

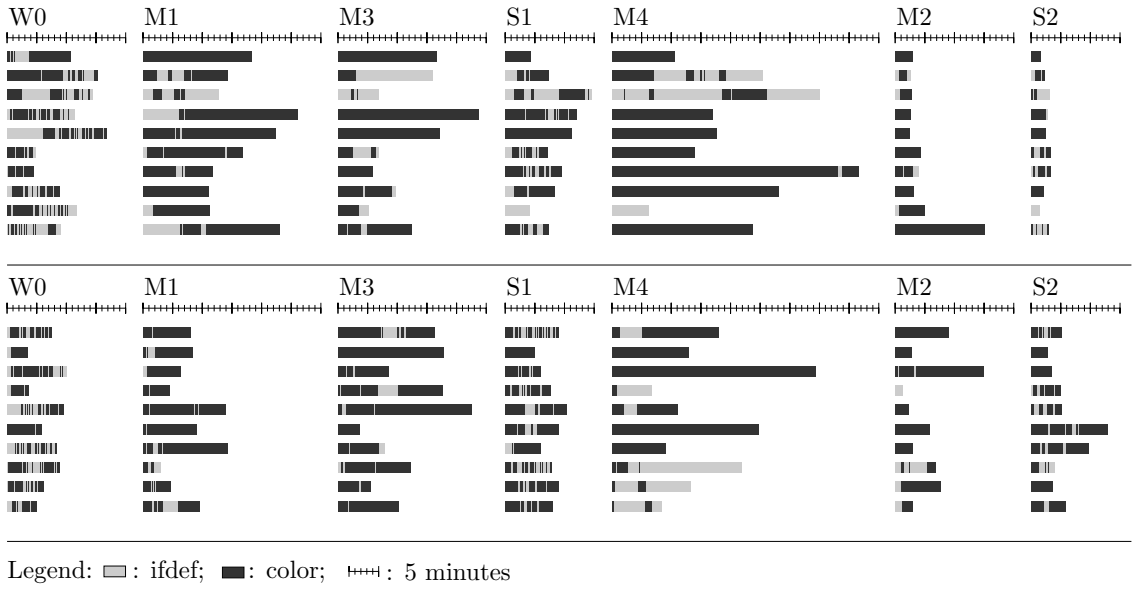


Figure 7.4: *Experiment 2: Timeline how participants switched between textual and colored annotations. Top: first instance 2010; bottom: second instance 2011.*

between ifdef directives and background colors. We did not provide any information on which annotation style is most suitable for which task, so that we could observe the behavior of participants unbiased. Since we had a smaller sample, two experimenters ⟨one experimenter⟩ sufficed to conduct the experiment. No deviations occurred.

7.4.5 Analysis

We discuss only the information necessary to evaluate our hypothesis. In Figure 7.4, we show how participants switched between the annotation styles in each task (light gray: ifdefs; dark gray: colors). Each row denotes the performance of a participant. For example, if we look at the first row, for W0 (warming-up task), the according participant switched between annotation styles (light and dark gray alternate). For all remaining tasks, the participant used only background colors.

The lengths of the bars indicate the time participants spend with a task. For example, the first participant needed considerable more time to solve M1 than to solve M2.

An interesting result can be seen in M4, the task, in which the target code was annotated with a red background color and participants of the color group performed worse in the first experiment. Although participants of our first experiment complained about the background color, most participants of our follow-up experiment used mainly the color version; only 3 of 10 ⟨4 of 10⟩ participants spent more time with the ifdef version.

We included the warming-up task W0 (counting the number of features), because it allows an interesting observation: Al participants switched between the annotation styles in this task. As the experiment went on, participants tend to stick with the color version. Hence, we reject our research hypothesis.

7.4.6 Interpretation

The results contradict our hypothesis. Based on the result of the first experiment and on the comments of some participants that the background color in M4 was disturbing, we assumed that participants would switch to ifdef directives when working on maintenance tasks, especially in M4, in which the entire class was annotated with red. However, most participants used the color version.

We believe that most participants did not even notice the disturbing background color. When we observed participants during the experiment, we found that some of them, currently working with the color version, moved close to the screen and stared at source code with red background color. Hence, we could observe that participants behaved like the background color was disturbing, but did not notice this consciously—they did not think of switching to ifdefs. We could have made our participants aware of the unpleasant background color. However, this would have biased our results, because our objective was to evaluate whether and how participants used the opportunity to switch between ifdef directives and colors.

Thus, participants did not necessarily recognize the disturbing effect of a background color. As a consequence, they were slowed down, such that they were as fast as the participants of our first experiment who also had the color version (a Mann-Whitney-U test revealed no differences between participants of this experiment and the color group of the first experiment). This illustrates the importance of choosing suitable background colors, because developers may not always be aware that their screen arrangement is unsuitable. Furthermore, since we did not tell our participants when to use ifdef directives and when to use background colors (we only showed them *how* they could switch), our result indicates that developers need to be trained in using a tool that uses background colors to highlight source code. We come back to the discussion of how to design proper tool support in Section 7.7.

7.5 Experiment 3: Do Colors Scale?

A question that immediately arose, even before the first experiment, is whether background-color use scales to large software systems. Obvious objections are that in real-world systems with several hundred of features, there would be considerably more colors than a developer can distinguish and that the nesting depth of ifdef directives would be too high to be visualized by blending colors. Hence, in a third experiment, we concentrate on scalability. In a nutshell, we could confirm the results of our first experiment for a large software system with over $99\,000$ lines of code and 346 features implemented in C, in that we could show an improvement of program comprehension for locating feature code when using background colors. In this section, we present the details of this experiment.

For better overview, we present the most important information in Table 7.5.

7.5.1 Objective

In this experiment, we evaluate whether background colors improve program comprehension in large software systems. To understand our setting, we have to understand

Context	Description	Section
Objective	Evaluate whether background-color usage scales to large systems	7.5.1
Material	Xenomai in two versions: C preprocessor, C preprocessor + background colors	7.5.2
Participants	14 graduate students from the University of Magdeburg	7.5.3
Tasks	Static tasks (locating features, interacting features, and files that contain feature code); maintenance tasks (locating bugs)	7.5.4
Execution	One computer lab; 17" TFT; source code presented in predecessor of FeatureCommander	7.5.5
Analysis	Correctness of answers and response time	7.5.6
Result	Colors speed up program comprehension in two kinds of static tasks; no effect for maintenance tasks	7.5.7

Table 7.5: *Experiment 3 in a nutshell.*

human limitations on perception. First, preattentive perception is limited to only few items (e.g., few different colors [Goldstein, 2002]). When there are too many distinctive items, the perception process is slowed down considerably, because more cognitive resources are required (e.g., to count the number of items). Second, human working memory capacity is limited to about 7 ± 2 items [Miller, 1956]. When there are more items to be kept in mind, they have to be memorized otherwise (e.g., by writing them down). Third, human ability to distinguish colors without direct comparison (i.e., when they are not shown directly next to each other) is limited to only few colors [Rice, 1991].

These limitations make a one-to-one mapping of colors to features infeasible in large software systems. Instead, we suggest an as-needed mapping, such that only a limited subset of colors is used at any time, which facilitates human perception. Our as-needed mapping is based on previous investigations of occurrences of ifdef directives in source code. First, for most parts of the source code, only two to three features appear on one screen [Kästner, 2010]. Second, most bugs can be narrowed down to certain features or feature combinations [Kästner, 2010]. Hence, developers can focus on few features most of the time, and avoid limitations to perception.

Thus, we propose a customizable mapping, which we show in Figure 7.5 (we present an extension of this tool in Figure 7.9). We provide a default setting, in which two shades of gray are assigned to features. Code of features located nearby in the source-code file has a different shade of gray, such that developers can distinguish them, but not recognize the features. Additionally, developers can assign colors to features they are currently working with. Since they are working only with a few features at a time, perception limits are not exceeded. Hence, our research hypothesis is:

RH2: Background colors improve program comprehension in large software systems.

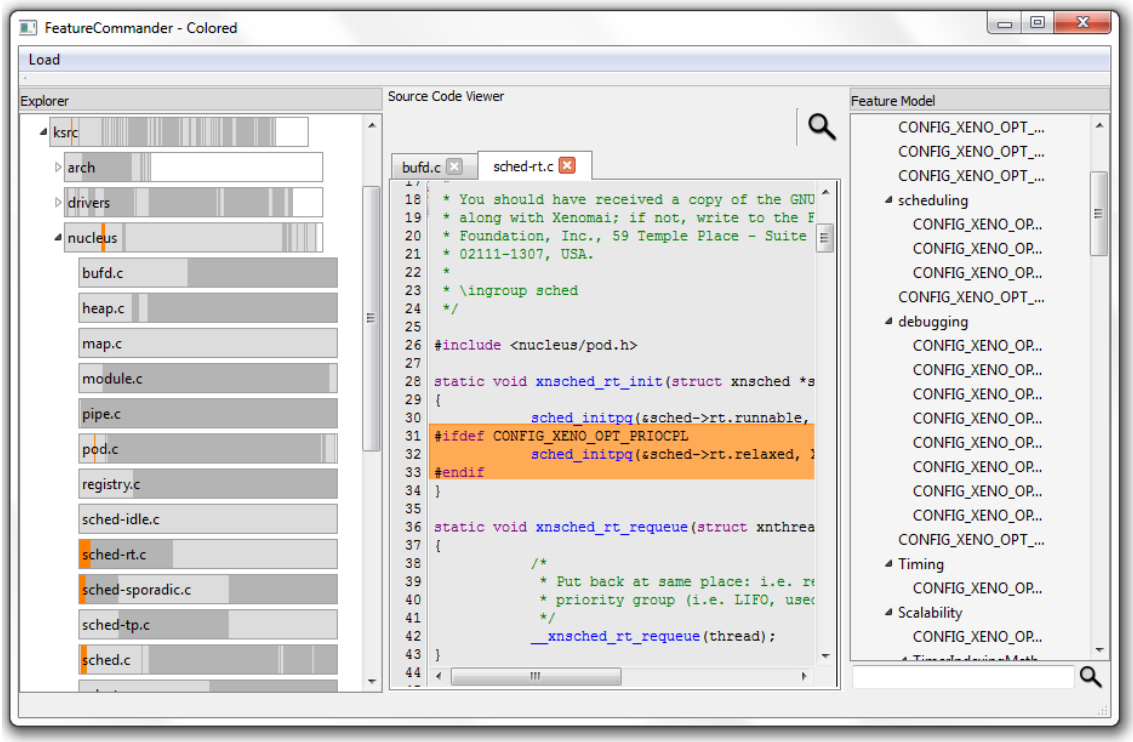


Figure 7.5: *Experiment 3: Screenshot of tool infrastructure of the color version.*

Large means that the source code consists of at least 40 000 lines of code [von Mayrhauser and Vans, 1995] and considerably more than 7 ± 2 features, such that humans cannot distinguish colors without direct comparison, if we used a one-to-one mapping of colors to features.

Regarding the opinion of our participants, we assume that they like background colors also in large software projects. Hence, our last research hypothesis is:

RH3: Participants prefer background colors over ifdef directives in large software systems.

7.5.2 Material

To evaluate our hypotheses, we replace MobileMedia (5 000 lines of code, 4 features) with Xenomai[3], a large real-time extension for Linux implemented in C. It consists of 99 010 lines of code including 24 709 lines of feature code and 346 different features. Xenomai can be configured for different platforms and provides numerous features, such as real-time communication and scheduling. There are a number of projects using Xenomai for

[3] `http://www.xenomai.org`

System	LOC	NOFC	LOF (%)	AND	SD	TD	ND
Apache	212 159	1113	44 426 (20.9)	1.17	5.57	1.74	5
FreeBSD	5 902 461	16 135	841 360 (14.3)	1.13	10.48	2.51	24
Linux	5 985 066	9 093	642 304 (10.7)	1.09	4.66	1.68	6
Solaris	8 238 178	10 290	1 630 809 (19.8)	1.12	16.17	2.72	8
SQLite	94 463	273	48 845 (51.7)	1.29	7.59	1.67	5
Sylpheed	99 786	150	13 607 (13.6)	1.06	6.31	1.38	6
Xenomai	99 010	346	24 709 (25.0)	1.21	6.07	1.44	5

LOC: Lines of code; NOFC; Number of features; LOF: Lines of feature code; AND: Average nesting depth; ND: maximum nesting depth; SD: Occurrences of features in different ifdef expressions; TD: tangling degree of expressions in ifdef directive.

Table 7.6: *Comparison of complexity of different systems.*

real-time behavior, for example, *RT-FireWire*[4], *USB for Real-Time*[5], and *SCALE-RT Real-time Simulation Software*[6].

To ensure the comparability of Xenomai with other real-world systems, we compared it with Apache, FreeBSD, Linux, Solaris, SQLite, and Sylpheed. To this end, we used cppstats[7], which computes several metrics to analyze the complexity of ifdef directives. In Table 7.6, we summarize the metrics [Liebig et al., 2010]. The systems differ in lines of code (LOC) and number of features (NOFC), some in the same range (e.g., SQLite), some larger (e.g., Linux) than Xenomai. Regarding the use of ifdef directives, Xenomai has the second highest percentage of annotated code (LOF) and the second highest average nesting depth (AND). The scattering degree (SD) indicates how often a feature occurs in different ifdef expressions, whereas the tangling degree (TD) indicates the number of different features in an ifdef expression. In both metrics, Xenomai shows similar values as Apache, Linux, SQLite, and Sylpheed. The same counts for the maximum nesting depth (ND).

We did not base this experiment on Java as the other experiments, because it was rather difficult to find a large-scale preprocessor-based software system implemented in Java. The largest we are aware of is ArgoUML, which consists of more than $100\,000$ lines of code, but has only 8 features [Couto et al., 2011]. We could have developed our own system in Java, but this would have been very time consuming and could have easily lead to a biased program (in that we design the system such that it confirms our hypotheses). Since there are numerous systems implemented in C [Liebig et al., 2010], we used an existing large-scale system, even though it was in a different language.

To present the source code to our participants, we implemented our own tool infrastructure including a source-code viewer using standard syntax highlighting and background colors (cf. Figure 7.5). We provided a file-browsing component, a list of all features as tree structure derived from Xenomai's build system, and a menu to load predefined color assignments. The file-browsing component had horizontal bars for each

[4] `http://rtfirewire.dynamized.com`
[5] `http://developer.berlios.de/projects/usb4rt`
[6] `http://www.linux-real-time.com`
[7] `http://fosd.de/cppstats`

folder and file, which indicates whether and how much feature code a folder or file contains.

In Xenomai, else and elif directives occurred.[8] We decided to assign the same color for each else and elif directive as to the according ifdef directive for two reasons. First, the code is still relevant for the same feature, because the selection of a feature has an effect on all accordingly annotated code fragments. This way, we can visualize that the same feature influences the annotated code fragments. Second, we did not want to introduce more colors than necessary because of the limits of human perception. Annotating each else and elif directive in a different color would exceed the limit of human perception faster. In Section 7.7, we present additional concepts to visualize nested ifdef directives as well as else and elif directives, which we did not evaluate in this experiment.

To ensure an optimal color selection for each task and prevent participants from having to search their own preferred color assignment, we defined a set of colors for each task. We ensured an optimal color selection by having consistent color assignments across tasks (i.e., a feature that occurred in more tasks has the same or similar color in all tasks) and by having colors that participants can distinguish within a task without direct comparison [Rice, 1991]. We chose more transparent colors than in the first two experiments and additionally allowed participants to adjust the intensity of background colors with a slider. In this experiment, we displayed the ifdef directives in the color version (instead of removing them as in the first experiment), because in the previous experiments, we showed a benefit of pure background colors. Furthermore, to scale background-color use to large systems, we do not have a one-to-one mapping of colors to features, so we need the textual information to tell to which feature a colored code fragment belongs. Additionally, we do not blend colors of nested ifdef directives, because we did not want to introduce more colors than necessary. Instead, we always display the color of the innermost feature and use vertical bars next to the source-code editor to visualize nested ifdef directives.

In addition to the color version, we designed another version, in which we removed everything associated to colors (ifdef version). Since the source code was large, we provided search functionalities for both versions.

In a second window, we presented the tasks to participants and provided text fields for their answers. Furthermore, to support participants in keeping track of time and preventing them from getting stuck on a task, a pop up appeared every 15 minutes to notify participants about the time that had passed.

As in the previous experiments, we gave participants paper-based questionnaires to collect their opinion (i.e., estimation of difficulty, motivation, and performance with the other version).

7.5.3 Participants

Our sample consisted of 9 master's and 5 PhD students from the University of Magdeburg, Germany. The master's students were participants of the 2010 course *Embedded Networks*, in which they completed several assignments regarding operating systems and networks, such as the implementation of clock synchronization of different computers.

[8] Code of an else directive is selected when code of an according ifdef directive is not selected.

They were offered to omit one implementation assignment as reward for participating in the experiment. The PhD students were experienced in the operating and embedded-systems domain and invited via e-mail. They participated without reward.

We measured programming experience with a preliminary version of the questionnaire described in Chapter 5. All participants were male; none was color blind. As in the first experiment, we created two comparable groups regarding programming experience according to the value of the questionnaire. Additionally, we matched both groups according to the familiarity with Xenomai, because some participants had some experience with the source code of Xenomai.

7.5.4 Tasks

In this experiment, we focused on static tasks, because we found in our first experiment a benefit of background colors for static tasks, but not for maintenance tasks. However, we included two maintenance tasks to control whether our findings still hold.

We had 10 tasks: 2 warming-up tasks (W1, W2; not included in the analysis), 6 static tasks (S1–S6), and 2 maintenance tasks (M1, M2). We had three different types of static tasks, two tasks per type:

For each type, we prepared two tasks. As example, we present one task for each type:

> *S1:* In which files does feature *CONFIG_XENO_OPT_TATS* occur?

The feature occurred in 9 files that were distributed in two folders: One include folder that contained the header files, and another folder that contained the c-files. In this task, the feature was annotated with a yellow background color. Hence, participants with the color version only had to look for a yellow background color in the software project.

> *S2:* Do features *CONFIG_XENO_OPT_PRIOCPL* and *CONFIG_XENO_OPT_SCHED_SPORADIC* occur together (i.e., nested) somewhere? If yes, in which files? At which lines does the inner feature start and end?

Both features were nested in the file *xenomai/ksrc/nucleus/sched-sporadic.c*, once in the middle, once near the end of the file. Nested ifdef statements are especially difficult to get right, which is why we included this type of task. We assigned yellow and blue to both features, because these colors are clearly distinguishable. To solve this task, participants with the color version had to look for a joint occurrence of yellow and blue, and make sure that the according ifdef statements are nested, not just used subsequently. In the second task of this type, the nesting occurred only at one position.

> *S3:* Which features occur in file *xenomai/ksrc/nucleus/sched.h*?

In this task, participants had to identify twelve different features. Hence, participants with the color version had to look for twelve different colors. This is an important task to get an overview of a file.

For maintenance tasks, we proceeded as for the first experiment. That is, we introduced bugs into the source code and gave participants a typical bug description that

included the feature selections in which the bug occurred. We consulted an expert in C and Xenomai to make sure that the bugs were typical for C programs. As example, we present the first maintenance task:

M1: If the PEAK parallel port dongle driver (*XENO_DRIVERS_CAN_SJA1000_PEAK_DNG*) should be unloaded, a segmentation fault is thrown.
The problem occurs when we select the following features:
CONFIG_XENO_DRIVERS_CAN
CONFIG_XENO_DRIVERS_CAN_SJA1000
CONFIG_XENO_DRIVERS_CAN_SJA1000_PEAK_DNG

We omitted the check whether a variable was null. Instead of `if (ckfn && (err = ckfn(block)) != 0)`, the code said `if ((err = ckfn(block)) != 0)`. If `ckfn` is accessed when it is null, a segmentation fault is thrown.

We presented the tasks in two phases. In the first phase, participants completed the tasks W1, S1, S2, S3, M1, and in the second phase, the tasks W2, S4, S5, S6, M2. We used a within-subjects design with two groups, such that each group works with both versions of the source code. Group A worked with the color version in the first phase and switched to the ifdef version in the second phase, whereas Group B worked with the ifdef version in the first phase and switched to the color version in the second phase. In each phase, both groups worked with the same tasks in the same order. Hence, Group A solved tasks W1, S1, S2, S3, and M1 with the color version, and W2, S4, S5, S6, and M2 with the ifdef version (vice versa for Group B). Corresponding tasks of both phases (i.e., W1/W2, S1/S4, S2/S5, S3/S6, M1/M2) are comparable (e.g., the same number of features had to be entered as solution). This allows us to compare the results within phases (between groups), and between phases (within groups).

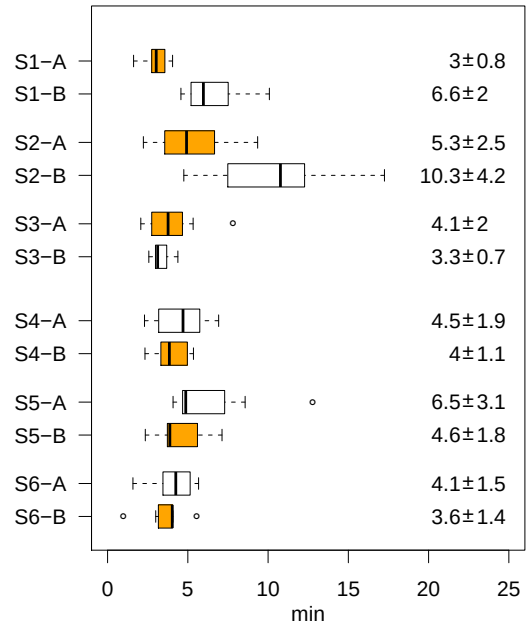
7.5.5 Experiment Execution

The experiment took place in June 2010 instead of a regular lecture session in a room with sufficient working stations (Windows XP) with 17" TFT displays. We gave an introduction, in which we explained the procedure of the experiment and how to use the tool. After the introduction, participants started to work on their own. When participants finished the last task of a phase, we gave them the usual paper-based questionnaire. Three experimenters checked that participants worked as planned. No deviations occurred.

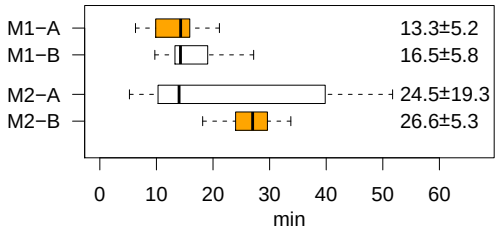
7.5.6 Analysis

Descriptive Statistics

As we did in the first experiment, we examined response times and correctness of tasks. In Figure 7.6, we show the response times of our participants. For the first two static tasks (S1 and S2), Group A (color version) is faster than Group B: In S1, Group A needed only 3 minutes, compared to 6.6 minutes of Group B (a difference of 55 %). In S2, Group A needed 5.3 minutes, and Group B 10.3 minutes (speed up by 49 %). Furthermore, maintenance tasks needed considerable more time (note the different scale in Figure 7.6b).



(a) Static tasks.



(b) Maintenance tasks.

Figure 7.6: *Experiment 3: Response time of participants in minutes. Highlighted boxes show groups who worked with the color version.*

In Figure 7.7, we show the correctness of answers. We omitted maintenance tasks in Figure 7.7, because we could not regard any of the answers as correct, although most participants narrowed the problem down to the correct file and method. We discuss this issue in Section 7.8. In S1, the difference is the largest, such that participants of Group B (without colors) performed better than participants of Group A.

In Figure 7.8, we present the opinion of participants, which we assessed after each phase. In the first phase, participants of Group A thought they would have performed worse with the ifdef version (medians for each task range from 2 to 3), whereas participants of Group B thought they would have performed better with the color version (medians for each task vary from 3 to 5). In the second phase, this estimation was reversed, such that participants of Group A thought they would have performed better with the color version (medians of 4 in each task), and vice versa for Group B (medians of 2 in each task). For difficulty, in four static tasks (S1: locating files of a feature; S2, S5: locating nested ifdefs; S3: locating all features in a file) and one maintenance task, the median is the same. For the remaining tasks, the median differs by 1. Regarding motivation, participants rated their motivation more heterogeneously. The median shows at least a medium level of motivation. For the first maintenance task (M1), the motivation for Group A (with colors) was very high, compared to Group B with a medium motivation.

In addition, we asked what version participants prefer: 12 participants liked the color version better and 13 said the color version is more suitable when working with preprocessor-based software. One participant did not answer any of both questions.

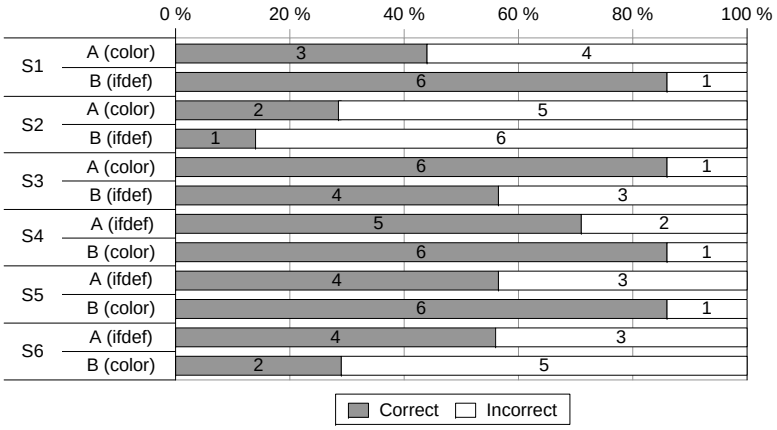


Figure 7.7: *Experiment 3: Number of correct answers per group and task.*

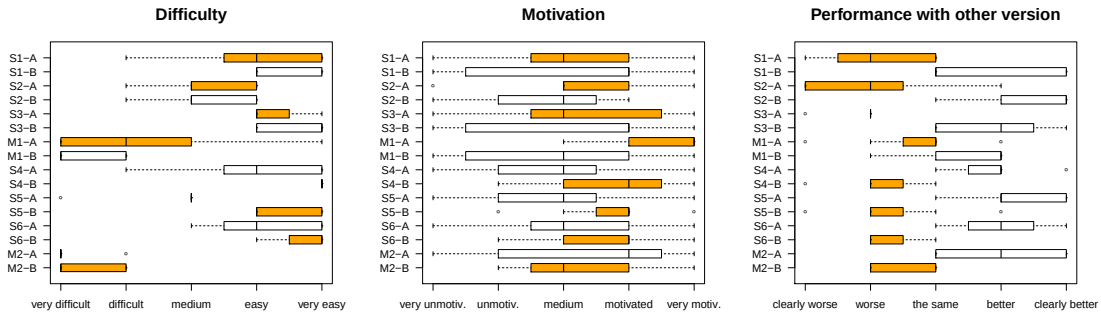


Figure 7.8: *Experiment 3: Box plots of participants' opinion.*

7.5.6.1 Hypotheses Testing

To evaluate our research hypotheses, we analyze correctness and response time. We start with evaluating the response times of participants in static tasks (**RH2**), for which we make several comparisons: between groups (Group A vs. Group B) and within groups (Group A (first phase) vs. Group A (second phase)) and (Group B (first phase) vs. Group B (second phase)). Since we make 3 comparisons on the same data, we need to adjust the significance level, for example, with a Bonferoni correction [Anderson and Finn, 1996]. In our case, we have to divide the significance level by three (because of 3 comparisons), which leads to a significance level of 0.017 to observe a significant difference (instead of 0.05).

We start with Group A vs. Group B. We applied t tests for independent samples, since the response times are normally distributed. In this experiment, we included incorrect answers, because our sample was too small to delete them. We discuss this in Section 7.8. We only observed significant differences for tasks S1 (p value: 0.001) and S2 (p value: 0.017). Hence, only for the first two tasks, participants who worked with the color version (Group A) were faster. In the second phase, we did not observe a benefit of background colors for program comprehension.

Next, we compare the response times of corresponding tasks between both phases (within groups), that is, S1 vs. S4, S2 vs. S5, S3 vs. S6, and M1 vs. M2. For Group A, we did not find any significant differences. However, for Group B, we observed a significant speed up for S4 (compared to S1; p value: 0.007) and S5 (compared to S2; p value: 0.011). Hence, when adding background colors, the performance of according participants increased for two tasks. On the other hand, removing background colors does not seem to affect performance, because participants of Group A were not significantly slower in the second phase. Hence, the results regarding response time speak both in favor of and against our research hypothesis.

Regarding correctness of answers, we conducted Fisher's exact test, because expected frequencies are smaller than 5. We did not find any significant difference (p values range from 0.133 to 0.500)

Finally, we compare the opinion of participants (**RH3**). A Mann-Whitney-U test reveals that the difference regarding estimation of performance with the other version is significant for all tasks except M1, the first maintenance task. To compensate for our small sample, we used an adapted table for the U distribution to decide whether a difference was significant [Nachar, 2008]. For difficulty, participants of Group B rated S4 and S5 significantly easier than participants of Group A. This is also reflected in the performance, such that participants of Group B are faster in these tasks (S4 vs. S1, S5 vs. S2). For motivation, we observe a significant difference only for the first maintenance tasks, such that participants of Group A were more motivated to solve this task compared to participants of Group B.

7.5.7 Interpretation

RH2 ***Background colors improve program comprehension in large software systems*** Our data can be interpreted both in favor of and against this hypothesis. However, since we did not specify that colors help only for certain kinds of tasks, we cannot accept this hypothesis. When comparing the response times between groups, we observed significant differences only in the first phase for two static tasks, such that participants working with the color version were up to $55\,\%$ faster. In the second phase, we did not observe any significant differences between groups. However, we observed that when we add colors in the second phase, the comprehension process of according participants (Group B) got faster by up to $55\,\%$. For maintenance tasks, we did not observe a significant difference in response time. Hence, we found that background colors improve program comprehension in preprocessor-based software systems in two static tasks.

For the third kind of static tasks (i.e., locating all features in a file), we did not observe significant differences. A possible reason is that in these tasks, the number of relevant features was 12, which means that participants had to work with 12 different colors. Although we selected colors to be clearly distinguishable without direct comparison, 12 might be too much and exceed the limits of human perception (cf. Section 7.5.1). Additionally, the working memory capacity of 7 ± 2 is exceeded with 12 features. For the other tasks, only 1 (S1, S4) or 2 (S2, S5) features had to be kept in mind. However, since we only combined 12 features with the third kind of static tasks, we can only theorize why this result occurred.

Furthermore, none of our participants solved a maintenance task correctly. The most likely explanation is that these tasks were too difficult given the time limit of the experiment. We discuss this problem in more detail in Section 7.8.

To sum up, background colors can help to familiarize with a large software system, especially to get an overview of the files of a feature or of nested ifdef directives. When we add background colors in the second phase, the performance of according participants increases. When we remove colors, it has no effect on the performance of according participants. Our observations align with the results of the first experiment that background colors can improve program comprehension in static tasks.

RH3 ***Participants prefer background colors over ifdef directives in large software systems*** We can accept this research hypothesis, because we found a preference for background colors. Participants who worked with the color version estimate they would perform worse without colors, even when we observed no difference in performance. We found the same effect in our first experiment. Additionally, all participants rate colors as more suitable when working with preprocessor-based software systems, and all but one participant preferred colors over no colors (except one participant who answered neither of both questions). This is also in line with the first two experiments, in which background colors were rated positively.

Hence, in large preprocessor-based systems, background colors have a potentially positive impact on program comprehension in terms of locating feature code. This means that we can circumvent human limitations regarding (preattentive) color perception and working memory capacity. Instead of a one-to-one mapping, we used an as-needed mapping based on observations about the occurrences of ifdef directives in source code, which scales to large software systems with over 300 features.

7.6 Summary of the Experiments

All three experiments analyzed how background colors affect program comprehension in preprocessor-based software. The focus of the first experiment was on medium-sized preprocessor-based systems, the focus of the second experiment on the behavior of participants using medium-sized systems, and the focus of the third experiment was on program comprehension in large systems. In Table 7.7, we summarize the results of all three experiments to give a better overview.

Interpreting the results of all three experiments together yields the following conclusions:

1. Carefully chosen background colors improve program comprehension in preprocessor-based software systems in terms of locating feature code, independently of size and programming language of the underlying system
2. Colors with a high saturation can slow down the comprehension process in terms of bug fixing
3. Participants like and prefer the color idea

Exp.	Source code	LOC	Features	Result
1	MobileMedia	5 000	4	Colors speed up static tasks; no effect or slow down for maintenance tasks
2	MobileMedia	5 000	4	Participants are unaware of the potentially negative effect of colors
3	Xenomai	99 010	346	The positive effects found in Experiment 1 scale for large systems

LOC: Lines of code.

Table 7.7: *Summary of main findings for all three experiments.*

First, we could show that carefully chosen background colors lead to a performance increase of participants for static tasks. This generalizes to medium-sized and large software systems. Additionally, we observed a performance speed up with both Java and C. Although we showed the positive effect only for two different sizes and two different programming languages, we expect similar positive effects also with small software systems (because limits to human perception are less stressed) and other programming languages that are similar to Java and C.

Second, we found that highly saturated background colors can slow down the comprehension process during bug fixing. We believe that visual fatigue causes this slow down. However, when given the choice, participants do not seem to be aware that a background color is disturbing and slowing them down. Nevertheless, for locating feature code, we did only find positive (or no) effects of background colors. Hence, depending on the task, the saturation of colors may play an important role. Thus, we suggest that source-code editors using background colors provide the option to adjust the saturation of background colors.

Third, the majority of our participants favored background colors. This is encouraging, because a new concept that is not accepted by the ones who are supposed to use it will hardly have a chance in practice. Hence, the acceptance of background colors is an important positive result.

However, we have to be careful with our conclusions. We cannot state that background colors are always helpful in every situation in which preprocessors are used to implement variability. Instead, we have to keep in mind the context of our experiments. We recruited mostly students for our experiments with considerably less experience than experts, who spent years on developing and maintaining preprocessor-based software systems. Furthermore, we only used two different software systems. Our results only apply to similar software systems, although we have evidence that many open-source systems, such as FreeBSD, Linux, Solaris, SQLite, and Sylpheed have similar characteristics. If the nature of a software system is different, we can only theorize how background colors affect the comprehension of preprocessor directives. Hence, future experiments with different experimental contexts are necessary to build a more complete understanding of the effect of background colors on program comprehension in preprocessor-based software systems.

To sum up, all results encourage us to use background colors more often in source-code editors. Hence, we developed a prototype FeatureCommander, which we present next.

7.7 Toward Better Tool Support

Our experiments were based on a relatively simple concept of background colors. Specifically, we based our work on CIDE [Kästner et al., 2008], a tool that uses background colors to visualize feature code. With our experiments, we gained useful insights into tool requirements for preprocessor-based software development. Based on these insights, including comments and suggestions from participants about functionalities, and by consulting similar tools (e.g., Spotlight) and literature on software visualization (e.g., Diehl [2007]), we implemented the tool *FeatureCommander*[9].

FeatureCommander is a prototype of an integrated development for the development of preprocessor-based software systems. It offers multiple visualizations to support program comprehension. The basic characteristic of FeatureCommander is the consistent use of colors throughout all visualizations. In Figure 7.9, we show a screenshot of FeatureCommander displaying source code from our third experiment. We refer to the numbers in Figure 7.9 when explaining the according concepts in the next paragraphs.

To assign colors to feature, we provide two different options: First, users can assign colors to features by dragging a color from the color palette (1) and dropping it on a feature in any of the views.[10] For efficiency, users can also automatically assign a palette of colors to multiple features (2). The automatic color assignment chooses colors such that they are as different as possible in the hue value of the HSV color model [Smith, 1978]. Furthermore, color assignments can be saved (3) and loaded (4), so that developers can easily resume their work. This way, we support an as-needed mapping of colors to features. When no color is assigned to a feature, it is represented by a shade of gray in all visualizations.

Using the color concept, we address the restricted human working memory capacity and the restricted human ability to distinguish colors without direct comparison (cf. Section 7.5.1). First, with the customizable color assignment to features and the default setting (shades of gray), we support the limited working memory capacity. Developers can select the features that are relevant for their task at hand, which is typically in the range of 7 ± 2. Hence, developers can immediately recognize whether they are looking at a relevant feature, because it is colored, whereas the non-relevant features do not stand out; they are gray. Second, developers have to tell only a few different colors apart, which is well within the human range to distinguish colors without direct comparison. Furthermore, we support developers in switching between tasks with different features, because color assignments can be easily saved and loaded.

[9] `http://fosd.de/fc`. On the website, there is also a video demonstrating the use of FeatureCommander. This video shows *all* functionality of FeatureCommander, not the reduced set we used in the third experiment. FeatureCommander with the reduced set is also available at the website.

[10] To recognize feature code, FeatureCommander uses a file that describes where an ifdef directive starts and where it ends.

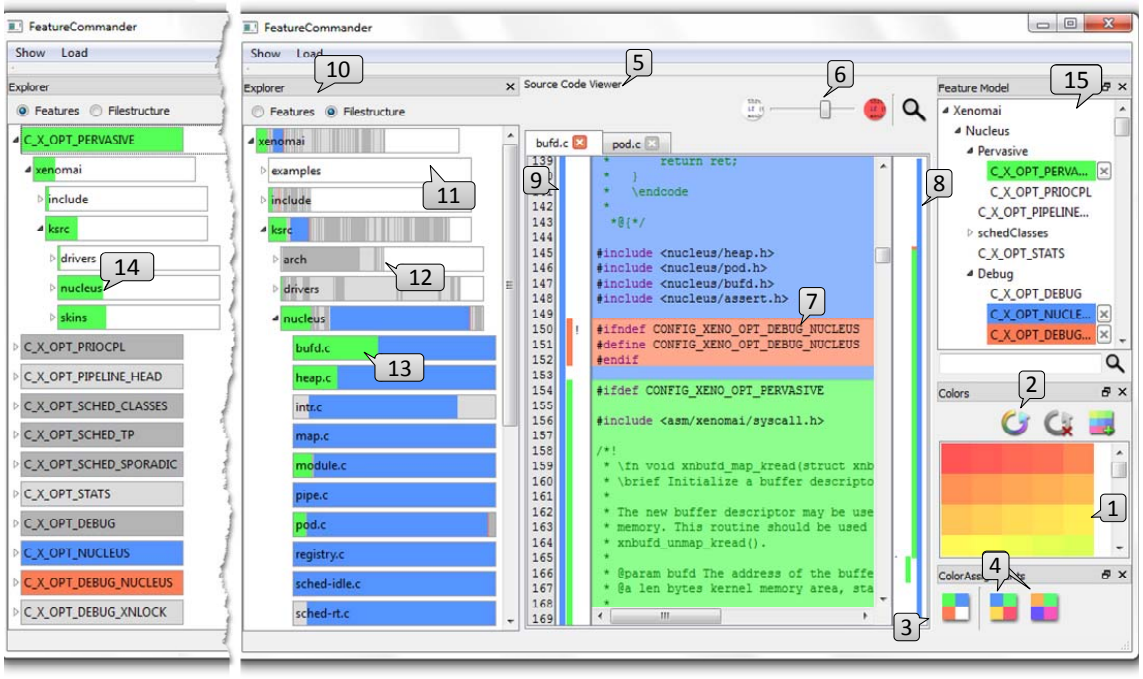


Figure 7.9: *Screenshot of FeatureCommander. The numbers designate concepts we explain in detail in the text.*

Similar to other integrated development environments [Kästner et al., 2009b; Stengel et al., 2011], we provide different views: *source-code view*, *explorer view*, and *feature-model view*. In the *source-code view* (5), the background color of source-code fragments indicates to which features fragments are related; according ifdef directives are also shown. To compromise between code readability and feature recognition, users can adjust the opacity of the background color (6). This way, we address that too highly saturated colors negatively affect program comprehension.

If a code fragment is assigned to multiple features (i.e., nested ifdef directives), we show only the background color of the innermost feature (7). The other features are visualized in the sidebars on either side of the source-code view, which visualize features as bars, ordered by the nesting hierarchy (8, 9). The right sidebar gives an overview of the whole document (8), the left sidebar shows the hierarchy of features of the currently displayed source code (9). Both sidebars are interactive, such that clicking them shows the according code fragment immediately. We implemented both sidebars, because it further supports users in locating a code fragment (although we did not evaluate the impact on program comprehension).

With our concept to deal with nested ifdef directives, we address both limitations of human perception: limited working memory capacity and limited ability to distinguish colors without direct comparison. First, when developers are working with a set of fea-

tures, they do not have to memorize additional colors, which would occur if we blended the colors. Second, the limited capability to distinguish colors without direct comparison is not exceeded, since the number of colors is not larger than the number of currently relevant features. In addition, the sidebars allow developers to navigate efficiently to feature code.

In the *explorer view* (10), users can navigate the file structure and open files. Files and folders are represented by their name and horizontal boxes, in which we visualize whether a file/folder contains feature code or not: If a file/folder does not contain any feature code, we leave the horizontal box empty (11). If a file/folder does contain feature code, we display vertical bars of different colors. When a feature has no color assigned, we use a shade of gray to indicate the occurrence of feature code (12). To allow developers to distinguish subsequent features without an assigned color, we use alternating shades of gray. When a feature has a color assigned, we show the according color in the explorer view (13). Furthermore, the amount of feature code in a file/folder is indicated by the length of each vertical bar. For example, if half a file contains feature code, then the horizontal box is filled half with vertical bars.

By using a visual representation to highlight files/folders, we allow developers to efficiently get an overview of a software system. They immediately recognize whether a file/folder contains feature code and whether the feature code is relevant for their current task. By using alternating shades of gray in the default setting, we allow developers to recognize the presence of different features in a file/folder, including the amount of feature code, without opening it.

To further support the developer in navigating in a large software system, we provide two tree representations of the project. One ordered according to the file structure, as displayed in Figure 7.9 (10). The other representation is ordered by features (14). For each feature, the files and folder hierarchies are displayed, including the horizontal boxes and vertical bars indicating the amount of feature code in a file/folder. This way, if developers need to get an overview of all files of a feature, they just activate the feature representation of the explorer view and can see the according files at one glance. In both representations, tool tips show the features of a file/folder.

With the representation ordered by features, we support developers in getting an overview of a software system. This way, they immediately recognize the files/folders, in which a feature is defined, without having to open any of them.

Finally, in the *feature-model view* (15), the feature model is shown in a simple tree layout. Features that are currently not of interest to developers can be collapsed. Colors can be dragged and dropped on features, as well as deleted. This helps developers to quickly locate a feature relevant for the current task and assign a color. After color assignment, all other views of FeatureCommander are updated with the assigned colors, so developers can efficiently locate feature code in files and folders of all other views.

FeatureCommander addresses the restricted human working memory capacity and the restricted human ability to distinguish colors without direct comparison. Both human limitations pose problems to a scalable use of background colors in large software systems. Developers can assign colors to features as needed. Since they typically work with only few features at the same time, we do not exceed working memory capacity or ability to distinguish colors. Furthermore, tool tips in the explorer view and source-code

view as well as assigned colors in the feature-model view support developers: When they forgot or cannot tell to which feature a color belongs, they can easily look it up.

In addition to the human-related problem, we address the problems of preprocessor statements: Long annotated code fragments, nested statements, and similarity to non-preprocessor code. First, since we highlight code fragments with background colors, ifdef and according endif statements can be easily spotted. Furthermore, we display vertical bars left and right of the source-code editor, which visualize the features of the currently displayed code fragment (left) or the features scaled to the complete file (right). Hence, the beginning and ending of each feature can be spotted easily. Second, we visualize nested statements. We always show the color of the innermost feature and the nesting hierarchy with the vertical bars, which allow users to easily identify the location of nested features in a file. Third, since background colors clearly distinguish from source code and colors are processed preattentively, FeatureCommander helps to locate feature code at first sight.

To sum up, with FeatureCommander, we provide a tool that support developers and researchers in working with preprocessor-based software systems. So far, we found that one other researcher group uses FeatureCommander [Zhang, 2012].

7.8 Threats to Validity

In this section, we discuss threats to validity for all three experiments. We summarize the threats, because they are similar for all experiments, and because we minimized some threats to external validity by conducting three similar experiments. Internal validity is threatened by the deviations, by the programming-experience questionnaire, and by keeping wrong answers in the response-time analysis. Furthermore, for the third experiment we could not rate any of the answers for the maintenance tasks as correct. External validity is threatened by maximizing internal validity in the first experiments.

7.8.1 Internal Validity

Some threats to internal validity are caused by deviations that occurred (described more detailed in Feigenspan [2009]). For example, a few participants arrived late, were seated in another room, received a personal introduction to the experiment, and could not be observed during the complete experiment. However, to assure anonymity of our participants, we did not retrace the deviations to the participants. Our sample is large enough to compensate for the deviations. They may have intensified or weakened the differences we observed, but they were too small compared to our large sample to significantly bias our results.

Another threat to internal validity is caused by the programming-experience questionnaire we used. Since these experiments were conducted before we developed the questionnaire presented in Chapter 5, we cannot be sure how well we measured programming experience. However, we constructed the questionnaire with the help of programming experts and a literature survey. Furthermore, the questionnaire was similar to the one we developed, so we measured programming experience well enough.

Additionally, we did not correct the response times for wrong answers for the second and third experiment. For the second experiment, this does not pose a threat, because we were not interested in response times, but in the behavior of participants. For the third experiment, our sample was too small to omit response times for wrong answers. To evaluate whether wrong answers might threaten the validity of our results, we checked the log data and response times for indications of biased results. We found no indication of bias in our data (i.e., wrong answers often missed only one or two features and the response times did not deviate considerably toward zero), so the threat caused by response times for wrong answers is negligible.

One problem only of the third experiment is that we could not rate any solution for maintenance task as correct. However, participants often named the correct file and method, which indicates that if participants had more time, they might have succeeded eventually. We believe that the realistic nature of the maintenance task (ensured by an expert on C and Xenomai) was too difficult for the time constraint and participants' expertise, despite pretests. Furthermore, our primary focus was static tasks. Thus, this threat does not bias our results.

7.8.2 External Validity

In the first experiment, we maximized internal validity to feasibly and soundly measure the effect of different annotations on program comprehension in pre-processor-based software systems. Thus, we deliberately accepted reduced external validity as tradeoff for increased internal validity. In the two follow-up experiments, we increased external validity by using different color settings, material, and more experienced participants.

In the first two experiments, we selected colors that are clearly distinguishable for participants. If we chose other colors (e.g., less saturated), we could have received different results (e.g., no significant differences for the last maintenance task). However, we wanted to make sure that colors are easily perceived and distinguished by participants. In our third experiment, we used different color settings to generalize our results regarding the use of colors and find optimal colors for highlighting feature code. Thus, we minimized the threat caused by color selection.

Another important facet is the scalability of colors to a large number of features. For the first two experiments, we had only four features, which is few enough that for a one-to-one color assignment, participants can discriminate different colors without direct comparison. To evaluate whether background-color use scales, we conducted the third experiment. Since we could confirm the benefit of background colors, we minimized this threat.

7.9 Applying our Framework

How could our framework helped us for our experiments? First, the overview of confounding parameters would have supported us in the planning phase of the experiment, such that we could have consulted the list and then decided for each parameter whether it is important and how we could have controlled it. To demonstrate the feasibility of our

Parameter	Control technique		Measured/Ensured	
	How?	Why?	How?	Why?
Personal parameters				
Color blindness	Constant	Large bias on result	Asked participants	Reliable
Domain knowledge	Constant	Ensure same comprehension process	Asked participants	Reliable
Programming experience	Matching	Major confound	Questionnaire	Reliable
Experimental parameters				
Attitude toward study object	Analyzed afterwards	Changes during experiment	Questionnaire	Reliable
Familiarity with study object	Constant	Reliable	Participants of same course	Have same familiarity
Process conformance	Observed participants	Reliable	Reminded participants to follow instructions	Reliable

Table 7.8: *Selection of confounding parameters for presented experiments.*

suggestion to present confounding parameters, we show the most important confounding parameters of all three experiments in Table 7.8. For example, we controlled for color blindness by keeping it constant. We selected this control technique, because colors blindness is rare in the population, so our results are applicable to a large part of the sample; thus, we do not limit external validity to much. To measure it, we asked participants whether they are color blind, because it is reliable and easy to apply. For programming experience, we applied matching, because it is the most important confounding parameter. We could not use it as independent variable, because we do not have the resources to recruit expert programmers. To measure it, we applied a questionnaire (i.e., a preliminary version of our questionnaire), because it is a reliable technique. Of course, it is subjective what parameters are most important; we believe that if we ignored any of these parameters, our results would have been significantly biased. In alignment with our recommendation, we present all confounding parameters in Appendix 10.2.

Second, we developed a preliminary version of the programming-experience questionnaire, for which we had to consult programming experts and literature to create the questionnaire, which took us some time. If we already had the questionnaire, we could have completed the planning phase of the experiment faster.

Third, PROPHET would have been very valuable for us. For the first experiment, we used a browser to present source code and created web pages for each source-code file, ensuring syntax highlighting and an equivalent to a package explorer. Furthermore, since participants were not allowed to use the search, we had to observe and remind participants to follow these instructions. For the second experiment, we implemented our own tool infrastructure to allow participants to switch between annotations. For the third

experiment, we implemented another tool infrastructure for the scalable color concepts. For both tools, we again had to implement several functionality, such as syntax highlighting and a package-explorer equivalent. If we could have extended PROPHET instead, such that it supports highlighting source code with background colors, we would have saved a considerable amount of time.

7.10 Related Work

In literature, the C preprocessor is often heavily criticized. Numerous studies discuss the negative effect of preprocessor use on code quality and maintainability (e.g., Adams et al. [2008]; Ernst et al. [2002]; Favre [1995, 1997]; Krone and Snelting [1994]; Spencer and Collyer [1992]). However, researchers have also explored different strategies to deal with these problems.

One group of approaches extracts structures from the source code (e.g., nesting, dependencies, and include hierarchies) and visualizes them in a separate view [Krone and Snelting, 1994; Pearse and Oman, 1997; Spencer and Collyer, 1992]. We follow this line of work and use similarly simple structures, but we focus on supporting developers directly in working with the annotated source code and integrate a visual representation of annotations with the underlying source code.

The model editors *fmp2rsm* [Czarnecki and Antkiewicz, 2005] and *FeatureMapper* [Heidenreich et al., 2008] allow users to annotate model elements to generate different model variants. Both tools can represent annotations with colors. The tool *Spotlight* [Coppit et al., 2007] uses vertical bars in the source-code editor to represent annotations, which are more subtle than background colors. Spotlight aims at improving the traceability of scattered concerns, which are represented by different colors. *SeeSoft* [Eick et al., 1992] represents files as rectangles and source-code lines as colored pixel lines. The color is an indicator for the age of the according source-code line. In contrast to our work, the influence of visualizations of these tools on program comprehension has not been assessed empirically.

In addition to visualizations, also views on configurations have been explored, which show only part of the feature code and, hence, reduce complexity [Atkins et al., 2002; Chu-Carroll et al., 2003; Hofer et al., 2010; Kästner et al., 2008; Singh et al., 2007]. A view on a variant or a view on a feature displays only feature code of selected features and hides all remaining code. Some tools even hide annotations completely, such that developers work on only one variant and may not even be aware of other variants or features [Atkins et al., 2002]. In an analysis of the change history of a large telephone switching software system, Atkins and others showed a productivity increase of $40\,\%$, when developers work with views provided by the Version Editor. However, hiding feature code may not always be feasible: For example, when code of a hidden feature shares code with a feature in which developers fix a bug, they might introduce bugs into the hidden feature code without knowing it [Ribeiro et al., 2010]. In this case, developers need the context of the complete software system to fix a feature-specific bug. Hence, views on source code and background colors complement each other for different tasks.

Furthermore, a severe problem for many approaches is precise fact extraction from unpreprocessed C code, especially if we want to reason not only about the preprocessor directives, but also about their combination of C code. Many researchers attempted analysis and rewrite systems for unpreprocessed C code [Aversano et al., 2002; Baxter and Mehlich, 2001; Garrido and Johnson, 2005; Hu et al., 2000; Livadas and Small, 1994; Overbey and Johnson, 2009; Tartler et al., 2011; Vidács et al., 2004]. For example, Ernst and others identify problematic patterns and quantify them in a large code base [Ernst et al., 2002], Tartler and others search for code blocks that are dead in all feature configurations [Tartler et al., 2011], and Hu and others use control-flow graphs to analyze the inclusion structure of files [Hu et al., 2000]. However, all these approaches aim not directly at improving program comprehension of developers, but form underlying mechanisms that can be used to build tools.

Another way to overcome understanding problems caused by preprocessors is to abandon them in favor of more disciplined approaches, such as feature-oriented programming [Prehofer, 1997] and aspect-oriented programming [Kiczales et al., 1997], or syntactic preprocessors such as ASTEC [McCloskey and Brewer, 2005]. Several researchers investigated automated refactorings [Adams et al., 2009; Kästner et al., 2009a]. However, preprocessors are still common in practice and the vast amount of legacy code will not disappear soon. Hence, there is still significant need for tools like ours that support developers when forced to deal with legacy code.

Finally, the idea of using colors to support a developer is not new. Early empirical work was published in 1986 [Rambally, 1986]. In this experiment, Rambally found that annotating source-code fragments with colors according to their functionality improves program comprehension, compared to a control-structure color-coding scheme (e.g., loops, if-then-else statements), and no colors at all. Furthermore, color use for various tasks is evaluated by several research groups, for example, highlighting source code for error reporting [Oberg and Notkin, 1992] or merging [Yang, 1994]. In 1988, the ANSI/HFS 100-1988 standard[11] was published, which included recommendations about the contrast of background colors and foreground colors. Today, syntax highlighting is an integral part of most integrated development environments. However, to the best of our knowledge, the use of background colors in preprocessor-based software has not been evaluated empirically.

7.11 Summary

Preprocessors are frequently used in practice to implement variable software. However, they introduce threats to program comprehension and are even referred to as "#ifdef hell". In this chapter, our goal was to give recommendations about the use of background colors to improve program comprehension in preprocessor-based software.

To fulfill our goal, we conducted a family of three controlled experiments, in which we revealed both benefits and drawbacks of background-color use. The results clearly showed that background colors have the potential to improve program comprehension in preprocessor-based software systems. Specifically, background colors helped par-

[11] `http://www.hfes.org/web/Standards/standards.html`.

ticipants to locate feature code, independently of size and language of the underlying project. Additionally, we found in all experiments that participants favor background colors. This is an important result, because the attitude of developers toward the tool they are working with can significantly affect their performance [Mook, 1996], for example, because they may stick longer with a task and not get frustrated by the tool. This effect is exploited in many tools, which typically provide numerous customizing options, so that users can adjust the tool according to their preferences.

However, we also found that colors have to be chosen with great care. Otherwise, they can slow down developers. Our results indicate that bright, saturated colors, such as we used in the first two experiments, are distracting and cause visual fatigue. Consequently, developers need more time when working with colors, for example, because of a need to rest their eyes. Hence, developers should be able to customize color settings according to their needs. For example, when developers located a code fragment that they suspect to cause a problem, they can turn off colors or adjust the saturation to a low degree.

Based on the results of our experiments, we implemented the prototype FeatureCommander, in which we realized scalable background-color use. Developers can efficiently adjust color settings to their needs, for example, by adjusting opacity. Thus, customizable background-color concepts as implemented in FeatureCommander can increase the efficiency of maintenance developers and reduce the cost of software development.

Chapter 8

Current Projects

The current projects are collaborations with André Brechmann (Leibniz Institute for Neurobiology), Christian Kästner (Philipps University Marburg), Sven Apel (University of Passau), Thomas Leich (Metop Research Institute), Don Batory (University of Texas at Austin), and Taylor Riché (National Instruments).

8.1 Using Functional Magnetic Resonance Imaging to Measure Program Comprehension

This section shares content with FSE-NIER'12 paper "Toward Measuring Program Comprehension with Functional Magnetic Resonance Imaging" [Siegmund et al., 2012a]

In Chapter 3, we described how we can measure program comprehension. We found that software measures should not be used as indicator for program comprehension and that controlled experiments are necessary. However, even when letting participants complete maintenance tasks or think aloud during program comprehension, we only have an indirect measure. To the best of our knowledge, researchers have not yet explored a direct way to observe what is happening inside the brain during program comprehension.

In neuroscience, researchers use *functional Magnetic Resonance Imaging (fMRI)* to observe cognitive processes since 1991 [Belliveau et al., 1991].

Background: Functional Magnetic Resonance Imaging FMRI is based on measuring differences in oxygen levels of blood flow in the brain. If a region in a brain becomes active, its oxygen need increases. To fulfill that increased need, the amount of oxygenated blood increases, and the amount of deoxygenated blood decreases. Both have different magnetic properties, which are used by fMRI to identify active brain areas.

Today, numerous studies analyze the functional organization of brain areas by varying cognitive tasks and task demands. The results can be used to interpret which brain areas contribute to which cognitive processes. For example, Cabeza and Nyberg describe that the prefrontal cortex is activated in almost all tasks that require high-level cognitive

functions, such as memory retrieval [Cabeza and Nyberg, 2000]. Thus, for a number of cognitive processes, we know which brain regions are responsible. This enables us to draw conclusions about how many resources different cognitive processes require and how different cognitive processes might be related.

In this section, we present our work regarding whether and how fMRI can be used to identify brain areas that are activated during the cognitive processes needed for program comprehension. We do not expect to find one area that is activated during program comprehension, but several areas that reflect the different facets of programming, such as reading words and working with numbers. This way, we hope to relate program comprehension to other cognitive processes (e.g., reading comprehension) and get a deeper understanding of how developers comprehend source code. In the long run, we might be able to answer questions like "What distinguishes good programmers from bad programmers?" "What distinguishes program comprehension from reading comprehension?" "How can we design better tools and programming languages?". Here, we present a concept of how such questions can be addressed with fMRI and propose a first experimental setup that is able to identify brain areas that are required during program comprehension.

8.1.1 Requirements for fMRI Studies

The most difficult issue in fMRI studies and most other studies that evaluate cognitive processes is to select suitable material and tasks (that is, source code in our case) and devise control tasks that control for brain activation elicited by processes that are needed for programming, but are not specific for it, such as reading itself. It is imperative that source code and tasks without a doubt lead participants to use the cognitive process that is the target of the evaluation, because otherwise, we cannot be sure what we measure. Furthermore, there are requirements specific to fMRI studies:

1. Source code should be short enough
2. Source code should have appropriate difficulty
3. There must be a control task
4. The complete experiment should not last too long

First, source code needs to be short enough to fit on a screen that is typically used within a magnetic-resonance scanner (from here on referred to as scanner). If source code is too long, participants would have to scroll. However, scrolling would also activate brain regions responsible for motor areas, so the activation caused by understanding would be confounded with the activation caused by scrolling.

Second, source code should be neither too difficult nor too easy to solve. If programs are too difficult, participants might not be able to determine the output correctly. In this case, we cannot be sure whether the understanding process took place or whether participants did something else. Furthermore, participants might require too much or too little time to solve a task. Typically, cognitive fMRI experiments require several repetitions

of tasks to ensure sufficient statistical power for data analysis. Between these blocks of programming activity, which should be of comparable length of up to 1–2 minutes, periods of rest or other control conditions are required to allow the amount of oxygenated blood elicited by the programming activity to return to a resting baseline. If a program is too easy to understand (i.e., within a few seconds), we might not be able to observe the activation elicited by the comprehension process, because of insufficient demand on the neural processes. In our studies, we have the opportunity to work with undergraduate students; thus, we adapt the difficulty to their ability.

Third, we need control tasks. Imagine we have suitable programs and tasks, what kind of brain activation would we see? Of course, the activation that is necessary for understanding. However, there is additional activation. First, participants see the source code, so there is an activation in the visual cortex (i.e., the part in the brain responsible for perceiving visual information). Second, participants move their eyes to see the source code, so we have to expect activation in the responsible motor cortex. To deal with these additional activations, we need control tasks that ideally only differ to the processes needed for program comprehension, nothing else. This way, we can compute the *difference* of activation between the control task and understanding task and see activation caused only by understanding.

Fourth, one complete session in a scanner should not last longer than one hour. Inside the scanner, participants have to lie as motionless as possible to avoid motion artifacts. However, after an hour, participants start getting restless and lose attention. To avoid bias, the session duration should not be too long.

Having presented requirements for fMRI studies, we describe how we selected according material and tasks in the next section.

8.1.2 Pilot Studies

To select suitable source code, experimental tasks, and control tasks, we conducted two pilot studies, which we summarize in Table 8.1. In this stage, it is not necessary to observe participants inside an fMRI scanner, because we can use response time and correctness to evaluate the suitability of source code and tasks. Thus, we conducted both pilot studies without a scanner. In Section 8.1.3, we describe the current setting of our study *with* fMRI.

8.1.2.1 Finding Suitable Understanding Tasks

As tasks, participants should manually compute the output of source-code snippets. To compute the output, they must understand the snippets. Thus, if participants determined an output correctly, a comprehension process must have occurred. As source code, we selected 23 different snippets from typical algorithms taught in first-year courses at German universities. Algorithms of first-year courses have a suitable difficulty, because we have the opportunity to work with second-year students and our participants should be able to solve as many tasks as possible correctly (so that we have as many times as possible a successful understanding process). To illustrate the nature of the programs, we show an example in Figure 8.1, which reverses a String, so the correct output is “olleH”. We present the selected source codes in Appendix 10.4.

Context	Description
Objective	Find suitable understanding tasks
	Find suitable control tasks
Material	23 small Java programs
	Subset (16) of previous programs
Participants	41 undergraduate students from the University of Passau
	8 students from the universities of Marburg and Magdeburg,
	1 professional Java developer
Tasks	Determining the output of a method
Execution	One computer lab; 17" TFT; PROPHET
Analysis	Correctness of answers and response time for both experiments
Result	12 understanding and control tasks

Table 8.1: *FMRI: Pilot studies in a nutshell.*

```
1 public static void main(String[] args) {
2    String word = "Hello";
3    String result = new String();
4
5     for ( int j = word.length() - 1; j >= 0; j--)
6       result = result + word.charAt(j);
7
8    System.out.println(result);
9 }
```

Figure 8.1: *Source code for one task.*

Note that, in all programs, we obfuscated identifier names to avoid giving participants hints about a program's purpose. For example, the variable `result`, which contains the result, is not named after what it contains (i.e., the reversed string), but the purpose of the variable (i.e., to hold the result of the program). This way, we force participants to use bottom-up comprehension. If participants would use top-down comprehension, we would also observe activation caused by memory retrieval, because information in source code is compared with domain knowledge, which is stored in memory. Thus, we focus on bottom-up comprehension, because brain activation is not confounded with memory activation. In the long run, if fMRI proves useful to measure program comprehension, we shall also consider more sophisticated settings measuring top-down comprehension.

In our first pilot study, we evaluated the suitability of the selected source-code snippets. To this end, we analyzed the time participants needed to compute the output of a source-code snippet and the correctness of the determined output.

Our participants were 41 second-year students from a software-engineering course at the University of Passau. They completed a basic programming course, so they were familiar with this kind of programs. The experiment was conducted on a computer in a lab at the University of Passau. To present source code and tasks to participants, we used our tool PROPHET (cf. Chapter 6).

To evaluate the difficulty of programs, we looked at response time and correctness. First, the mean response times for the tasks are between 15 and 316 seconds. Based on

```
1 public static void main(String[] ) {
2    String word = "Hello';
3    String result = new String();
4
5    for (int j = word.length() - 1; j >= 0; j--)
6       result = result + word.charAt(j);
7
8    System.out.println{result);
9 }
```

Figure 8.2: *Syntax errors for one task.*

these results, we excluded six tasks from further studies with response times above 120 seconds, because participants would require too much time to solve them while in the scanner. Furthermore, we excluded one task with a mean response time below 30 seconds to further reduce the variability of fMRI activation due to differences in task duration. Consequently, we have 16 tasks with mean response times between 37 and 104 seconds. When looking at correctness, the 16 tasks were solved correctly by 29 to 41 participants. On average, the number of correct solutions for a task was 37 (90 %) participants. Thus, the tasks did not seem too difficult to solve, so we did not exclude any task based on correctness.

8.1.2.2 Finding Suitable Control Tasks

Suitable control tasks should ideally only differ to the understanding task in that comprehension did not take place. Everything else should be the same. Thus, we use the same programs as for the understanding task. As tasks, we ask participants to identify syntax errors that we introduced to the source code. As an example, we show the source code of the first pilot study (cf. Figure 8.1) with syntax errors in Figure 8.2. The errors are in Line 1 (parameter name is missing), Line 2 (wrong character to terminate String "Hello"), and Line 8 (curly bracket to pass result). For the remaining 15 tasks, we included similar errors (always 3).

To evaluate the suitability of our control tasks, we conducted a second pilot study. As participants, we recruited students from the Philipps University Marburg (4), students from the University of Magdeburg (4), as well as one professional Java programmer. All participants were familiar with Java at least at the level of second-year students. Again, we used PROPHET to show source code to participants and collect response times and answers of participants.

To select suitable control tasks, we looked at the response times of participants, which are between 20 and 120 seconds. Regarding correctness, we found that most participants found at least two syntax errors. Thus, we can be sure that participants worked on the tasks and that the tasks are neither too easy nor too difficult.

Based on the pilot studies, we found suitable understanding and control tasks. The next step is to set up the experiment for the fMRI scanner. In the next section, we describe what a session inside a scanner can look like as a first approach to understand the neural correlates of program comprehension.

8.1.3 Program Comprehension Based on fMRI

To better understand our setting, we introduce the typical setting of fMRI experiments. First, there is a measuring stage of about 10 to 20 minutes, in which the brain of participants is measured regarding size and form. This is necessary to map the measured changes in blood flow to the brain region. Then, the actual experiment starts. One typical trial is structured as follows:

- Experiment condition
- Rest condition
- Control condition
- Rest condition

The experiment and control conditions are the ones we described in the previous section (i.e., understanding, syntax error). In the rest condition, participants relax or do nothing. This is necessary to let the level of oxygenated and deoxygenated blood return to a baseline level, and because working inside the scanner is exhausting.

Taking into account the initial measuring and trial length, we have to make sure that a session inside the scanner does not last too long (cf. Section 8.1.1). However, with our 16 tasks, the experiment would be too long. Hence, we excluded another four tasks. We excluded one task with the shortest and one with the longest response time in the second pilot study. Furthermore, we excluded two tasks that are similar to other tasks (e.g., we excluded a program that computes the sum from 1 to n, which is similar to compute the product from 1 to n). Thus, we have 12 tasks in our final experiment.

While lying inside the scanner, participants are instructed to determine the output of a method (experiment condition) or find three syntax errors (control condition) and press a button when they are finished. They do not say or enter their solution to avoid motion artifacts, activation in the brain region responsible for producing speech, and to minimize the time inside the scanner. To evaluate whether participants solved a task correctly, we ask them after the scanner session to look again at the source code and state the answer, which is a typical setting for fMRI studies. In the rest condition, participants are instructed to relax. During all conditions, participants are told to move as little as possible to avoid motion artifacts. Furthermore, we use an eye tracker to track eye movement during tasks. To show what it is like for participants to lie inside the scanner, we show a photo of one of our participants in Figure 8.3. To reduce motion artifacts, the head of participants is fixated.

We arranged the order of code of the experiment and control condition such that most of the code in the experiment condition is presented before code in the control condition. This way, participants do not recognize source code from the control condition, but understand a program bottom up.

Currently, we are running the experiment described in the previous section inside the scanner. Conducting and running such experiments is a long process (e.g., getting a time slot to use the scanner, recruiting participants, analyzing the data), so we have only preliminary results to report. So far, we can confirm that the tasks are actually suitable

Figure 8.3: *Photo of participant inside the scanner.*

for our purpose, and that a session is not too long for participants. Furthermore, the experiment is interesting for participants.

To give an impression of the results we might get from our study, we show a typical image of brain activation in Figure 8.4 from a different study of ours. Highlighted regions indicate an activation, in this case mostly the prefrontal and visual cortex. When our studies are completed, we might obtain similar images.

8.1.4 Vision

When finished with the measurements, we hope to have a first impression about which brain regions are activated during program comprehension. Specifically, we expect to see activation in the prefrontal cortex, which is active during higher cognitive tasks (what we believe program comprehension is). Furthermore, we believe that programs containing loops require more cognitive resources than programs without any loops. Thus, we may observe a stronger activation during understanding programs with loops. We also believe that we observe more eye movements in source code containing loops (which we observe with an eye tracker). Additionally, we may see activation in regions related to reading comprehension, because participants have to read and understand words. Moreover, we believe that verbal working memory capacity is needed to comprehend source code, because words have to be processed. Thus, we expect activation in the related brain areas (left parietal lobe). To sum up, we believe to observe several activations in different brain regions that are related to activities involved in program comprehension.

In the long run, we might be able to find out what distinguishes good programmers from bad programmers. Good programmers may have a certain activation pattern that completely differs from bad programmers. For example, fMRI studies of expert and novice golfers showed completely different activation patterns when they were thinking about hitting a golf ball. Expert golfers showed activation in one small, distinguished brain region, whereas novices showed activation in several different brain regions. The reason is that expert golfers have abstracted the knowledge of hitting a ball. With pro-

Figure 8.4: *Example for activation pattern.*

gram comprehension, it might be similar, such that experts somewhat abstracted the comprehension process.

Having a deeper understanding of program comprehension, we might be able to better teach programming to students and develop tools and languages that support the human way of program comprehension.

8.1.5 Related Work

The single most related paper to our work describes the information programmers need to continue their tasks after interruption [Parnin and Rugaber, 2012]. To this end, Parnin and Rugaber describe how different types of memory located in different brain areas affect different programming activities. It is similar to our work, in that it maps brain regions to programming tasks. However, the authors do not use fMRI, but base their work on previous studies that map brain regions to different types of memory. We are not aware of any results regarding program comprehension based on fMRI.

Furthermore, there is work in the domain of neuroscience to analyze cognitive processes. Most related to ours is work based on reading comprehension (i.e., how participants understand written text). For example, Moss and others analyzed brain regions activated during strategic reading comprehension [Moss et al., 2011]. To this end, participants were given different tasks, such as paraphrasing of different texts. Depending on the tasks, different brain regions were activated.

In the domain of software engineering, there is work aiming at measuring and improving program comprehension without fMRI. For example, Robillard and others conducted a controlled experiment to analyze how developers understand source code and how effective they are [Robillard et al., 2004]. Jeanmart and others analyze how different programming styles affect program comprehension [Jeanmart et al., 2009].

8.1.6 Conclusion

To conclude, we are exploring whether we can use fMRI to better understand program comprehension in an ongoing endeavor. To the best of our knowledge, there is no prior empirical work to measure program comprehension using fMRI. So far, we designed and tested the material we are using. Furthermore, we conducted first sessions inside a scanner that show that our experimental setup is feasible.

As a next step, we will continue our measurements. We hope to find a mapping of brain regions to other, already evaluated cognitive processes, such as reading comprehension, which might enable us to develop a theory of program-comprehension processes based on neuroscience.

8.2 Comparing Comprehension of Physically and Virtually Separated Concerns

This section shares content with FOSD'12 paper "Comparing Program Comprehension of Physically and Virtually Separated Concerns" [Siegmund et al., 2012b]

Separation of concerns has been recognized as an essential strategy to implement understandable and maintainable software systems [Parnas, 1972]. It is common to believe that separating code along features improves program comprehension. In this section, we set out to evaluate whether separating features physically (i.e., into separate feature modules) improves program comprehension compared to separating code virtually (i.e., annotating feature code). In particular, we compare the mechanism of feature-oriented programming as implemented in the tool FeatureHouse [Apel et al., 2009] to preprocessors, in which feature code is annotated with ifdef directives.

8.2.1 Experimental Design

To evaluate whether physical separation of concerns à la FeatureHouse has a benefit on program comprehension, we designed a controlled experiment. So far, we have only preliminary data to report, because we could recruit only eight participants. Hence, we regard the first conduct as pilot study to evaluate the feasibility of our experimental setting. Thus, we focus on the design of the experiment. Our data analysis and interpretation are a suggestion on how the research questions can be answered; we do not have sufficient data to reliably answer any of the research questions. Hence, our analysis and interpretation do not aim at answering the research questions, but serve as example how the research questions can be answered. To give an overview of our experiment, we summarize the most important information in Table 8.2.

8.2.1.1 Objective

With our experiment, we target the question whether participants understand physically separated source code (feature modules) different than virtually separated source code

Context	Description	Section
Objective	Evaluate feasibility of experimental design; Compare effect of physical and virtual separation of concerns on program comprehension	8.2.1.1
Material	MobileMedia in two versions: Java ME with Antenna, FeatureHouse	8.2.1.2
Participants	8 graduate students from the University of Passau	8.2.2
Tasks	Maintenance tasks (locating bugs)	8.2.1.3
Execution	One computer lab; 17" TFT; PROPHET	8.2.2
Analysis	Correctness of answers, response time, search behavior, first action to solve task	8.2.2.1
Result	Two tasks are too easy, FeatureHouse group unhappy to work with that version	8.2.2.3

Table 8.2: *Separation of concerns: Pilot study in a nutshell.*

(preprocessor directives). This question arises when we look at human information processing. To process information from the outside world, we use our working memory, which holds information we perceive and makes it available for further processing [Baddeley, 2001]. However, working memory capacity is limited to only few items [Miller, 1956]. By structuring information, we can store more information. For example, we can group information of a shopping list into groceries and clothing and then memorize few items of the grocery and few items of the clothing category. In physically separated source code, the amount of information presented in one place is smaller and more clearly structured, so the working memory of participants might not be stressed too much.

However, when the present information is not enough to understand code, participants need to search for relevant information. Hence, they might need more time, and during their search, they have to keep in mind where their search started. For that, they need more working memory capacity. Thus, our first research question is the following:

RQ1: Does physical separation of concerns improve program comprehension?

Additionally, we are interested in the search behavior of participants. In virtually separated code, files are larger, because they typically contain code of several features. Thus, participants may use the search function more often to find information. In physically separated code, one file contains information of only one feature; hence, relevant code may be easier to find without using the search or using it less frequently. However, the information presented in one file might not be enough to understand the code, so participants might use a global search (i.e., across modules) more often. Thus, we state a second research question:

RQ2: Is there a difference in the search behavior between physically and virtually separated concerns?

Furthermore, there might be a difference in the strategy that participants use to find a bug. In the FeatureHouse version, participants might start by opening a file in the relevant feature module, because according code is located only in that module and because files are short compared to the ifdef version. In the ifdef version, participants might start by using the global search function to locate code of the relevant feature, because according code is scattered across the project. Thus, we state a third research question:

RQ3: Is there a difference in the first action to find a bug?

8.2.1.2 Material

As material, we used the last release of MobileMedia. From the ifdef version, we created another version based on FeatureHouse.[1] To ensure that both versions differ only in the underlying programming technique, two reviewers realized the refactorings. They evaluated the work of the other reviewer on few code fragments. We explicitly encourage other experimenters to evaluate the comparability of both versions and give us feedback.

To illustrate what physical and virtual separation looks like in MobileMedia, we show an example in Figure 8.5. The top excerpts (8.5a) show virtual separation implemented with ifdef directives; the bottom excerpts (8.5b to d) an implementation of the same code with FeatureHouse.

An important difference between both versions is caused by the technique, such that in the FeatureHouse version, there are more folders, because for every feature or feature combination, a new folder is created, in which files are stored according to the declared packages. In the ifdef version, there are no folders for features or feature combinations, but only those folders defined by the package declarations (which are also present in the FeatureHouse version). To illustrate this difference, we present screen shots of the file structure of both versions in Figure 8.6.

To evaluate our research questions, we use a between-subjects design. This way, we can compare the performance of participants of both groups. For the first research question, we analyze response time and correctness for maintenance tasks. Response time was logged automatically with PROPHET, and correctness determined manually by an expert.

For the second research question (regarding the search behavior), we log how participants use the search function during solving maintenance tasks. Participants can either use a local search, that is, within a file, or a global search, that is, in all files and folders of the complete project. Both searches use strings (no pattern matching or syntactical search).

For the third research question, we log the behavior of participants, that is, opening and closing files, switching between files, and using local or global search, including the search term.

[1] There is also an AspectJ version of MobileMedia, which uses physical separation of concerns. However, AspectJ syntax requires considerable training, so we use FeatureHouse instead, and leave evaluation of physical separation of concerns á la AspectJ for future work.

```
1  // #if includeMusic || includeVideo
2  //...
3  public class MusicMediaUtil extends MediaUtil {
4     public byte[] getBytesFromMediaInfo(MediaData ii)
5           throws InvalidImageDataException {
6        byte[] mediadata = super.getBytesFromMediaInfo(ii);
7        if (ii.getTypeMedia() != null) {
8           //#if (includeMusic && includeVideo)
9           if ((ii.getTypeMedia().equals(MediaData.MUSIC)) ||
10              (ii.getTypeMedia().equals(MediaData.VIDEO)))
11          //#elif includeMusic
12          if (ii.getTypeMedia().equals(MediaData.MUSIC))
13             //#elif includeVideo
14             if (ii.getTypeMedia().equals(MediaData.VIDEO))
15             //#endif
16          //#endif
17       //#endif
18       //...
19    }
20    //...
21 }
22 //#endif
```

(a) Virtual Separation

```
1  public class MusicMediaUtil extends}}@* MediaUtil {
2    private boolean isSupportedMediaType(MediaData ii) {
3      return false;
4    }
5
6    public byte[] getBytesFromMediaInfo(MediaData ii)
7       throws InvalidImageDataException {
8       byte[] mediadata = super}}@*.getBytesFromMediaInfo(ii);
9       if (ii.getTypeMedia() != null) {
10        if (isSupportedMediaType(ii))
11        { ... }
12      }
13   }
14   //...
15 }
```

(b) FeatureHouse–*Music_OR_Video*

```
16 class MusicMediaUtil {
17    private boolean isSupportedMediaType(MediaData ii) {
18       return original(ii) || ii.getTypeMedia().equals(MediaData.MUSIC);
19    }
20 }
```

(c) FeatureHouse–*Music*

```
21 class MusicMediaUtil {
22    private boolean isSupportedMediaType(MediaData ii) {
23       return original(ii) || ii.getTypeMedia().equals(MediaData.VIDEO);
24    }
25 }
```

(d) FeatureHouse–*Video*

Figure 8.5: *Virtual and physical separation of concerns using the preprocessor Antenna (a) and FeatureHouse (b-d).*

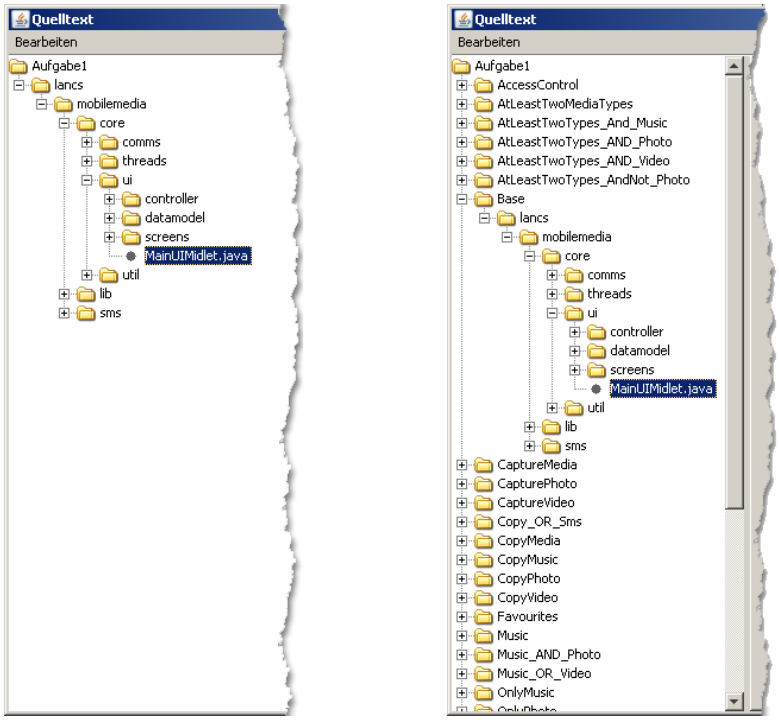


Figure 8.6: *File structure of ifdef version (left) and FeatureHouse version (right).*

To present source code, tasks, and the questionnaire to participants, we used our tool PROPHET (cf. Chapter 6). To control for programming experience, we used the questionnaire presented in Chapter 5. In addition to measuring program comprehension, the search behavior, and first action for a task, we used a questionnaire to assess the opinion of participants regarding difficulty of tasks and motivation to solve a task (both on a five-point Likert scale). This way, we get more information to interpret our data.

8.2.1.3 Tasks

We developed five bug-fixing tasks, such that we can evaluate the claimed benefit of physical separation of concerns. Hence, classes in the FeatureHouse version that contain the bug are small compared to the ifdef version. To get an impression of how short source code has to be to provide a benefit (if any), we introduced the bugs in classes of different size. All tasks were designed to have comparable difficulty, so that it does not confound the results. We encourage other researchers to evaluate the comparability of tasks. Additionally, we evaluate whether comparing similar statements of different features helps to find a bug (Task 2). Furthermore, we analyze how the need to consider two classes of different features affects program comprehension (Task 5). We designed only 5 tasks to avoid a too long duration.

To present the tasks, we gave participants a bug description as a user might provide it. Additionally, we provided the feature that is selected when the bug occurs, so that participants can focus on feature code. This way, we can evaluate our research ques-

Task	Bug Description	Feature
1	When converting media, the counter that describes how often a medium was looked at is always set to 0 instead of the actual value.	*Sorting*
2	When a video should be played, the according screen ("Play Video") is not shown. Nothing happens.	*Video*
3	When clicking on "View Favorites" in the menu, no favorites are shown, although there are favorites and the according functionality is implemented.	*Favourites*
4	When pictures should be shown sorted by number of views, they appear unsorted anyway.	*Sorting*
5	Although a user has no rights to delete a picture, she can delete it anyway.	*AccessControl*

Table 8.3: *Overview of maintenance tasks.*

tion, because code is separated along features. In Table 8.3, we provide an overview of all tasks. To complete a task, participants are instructed to determine the class and line number of the bug, describe why the problem occurs, and suggest a solution as verbal description. We use all information to determine whether a task was solved correctly. Next, we describe each task in detail, show relevant code fragments with bugs highlighted, and discuss whether the FeatureHouse or ifdef version might provide benefits for program comprehension.

Task 1 In this task, the bug is located in feature *Sorting*, which provides functionality to sort media data, for example, according to how often a photo was viewed. As bug, we set the according counter to 0 instead of the actual value. To illustrate this bug, we show relevant source code in Fig. 8.7. The class that contains the bug is considerably smaller in the FeatureHouse version, such that the complete class fits on one screen. However, the original method definition in the base feature might be relevant to understand the bug. Thus, participants of the FeatureHouse group might be faster, if they do not look at the base code.

Task 2 In Task 2, a false identifier is used (`SHOWPHOTO` instead of `PLAYVIDEO`). We show an excerpt in Figure 8.8. Like in Task 1, the FeatureHouse version is considerably shorter. However, in the ifdef version, source code for other features (e.g., *Photo*) is visible, which participants might compare with feature *Video* and, thus, might use to recognize that `SHOWPHOTO` is the wrong identifier to play a video. Another difference is the location at which the command is defined. In the FeatureHouse version, command definition and use appears on the same screen, but not in the ifdef version. Thus, we can argue both in favor of and against a benefit for program comprehension in the FeatureHouse version.

Task 3 In Task 3, `false` instead of `true` is passed to a method showing a list of media, so no favorites are shown (Figure 8.9). In the ifdef version, the bug is located in class

```
1  public class MediaUtil {
2  // 73 lines of additional code
3     public MediaData getMediaInfoFromBytes(byte[] bytes)
4           throws InvalidArrayFormatException {
5  // 64 lines of additional code
6        MediaData ii = new MediaData(x.intValue(),
7           albumLabel, imageLabel);
8  // 5 lines of additional code
9
10       // #ifdef includeSorting
11       ii.setNumberOfViews(0);
12       // #endif
13 // 62 lines of additional code
```

(a) Ifdef

```
1  class MediaUtil{
2     private MediaData createMediaData(String iiString, String fidString,
3           String albumLabel, String imageLabel) {
4
5  // 16 Lines of additional code
6        MediaData ii = original(iiString, fidString,
7              albumLabel, imageLabel);
8
9        ii.setNumberOfViews(0);
10       return ii;
11    }
12
13 // 10 lines of additional code
```

(b) FeatureHouse–*Sorting*

```
14 public class MediaUtil {
15 // 121 additional lines of code
16    private MediaData createMediaData(String iiString,
17       String fidString, String albumLabel, String imageLabel) {
18
19       Integer x = Integer.valueOf(fidString);
20       MediaData ii = new MediaData(x.intValue(), albumLabel, imageLabel);
21
22       return ii;
23    }
24 // 47 additional lines of code
```

(c) FeatureHouse–*Base*

Figure 8.7: *Bug location for Task 1 (bug highlighted).*

`MediaController` in Line 6; in the FeatureHouse version, the bug is located in class `MediaController` in feature *Favourites* in Line 5. Like for the other tasks, the ifdef version is longer; however, the code of the FeatureHouse version does not fit on one screen. Thus, we might observe no or a weaker benefit for program comprehension for this task.

Task 4 In this task, the sorting algorithm is not implemented, so calling it is futile (Figure 8.10). In the ifdef version, the bug is located in class `MediaListController` in Line 5; in the FeatureHouse version in feature *Sorting*, Line 6 of class `MediaListController`. Again, the ifdef version is considerably longer, whereas in the FeatureHouse version, the

```
1  public class MediaListScreen extends List {
2      //#ifdef includePhoto
3      public static final int SHOWPHOTO = 1;
4      //#endif
5       //#ifdef includeVideo
6      public static final int PLAYVIDEO = 3;
7       //#endif
8  // 64 additional lines of code
9      public void initMenu() {
10         //#ifdef includePhoto
11         if(typeOfScreen == SHOWPHOTO)
12            this.addCommand(viewCommand);
13         //#endif
14 // 7 additional lines of code
15         //#ifdef includeVideo
16         //[NC] Added in the scenario 08
17         if(typeOfScreen == SHOWPHOTO)
18            this.addCommand(playVideoCommand);
19         //#endif
20 // 32 additional lines of code
```

(a) Ifdef

```
1  class MediaListScreen {
2     public static final Command playVideoCommand =
3           new Command("Play Video", Command.ITEM, 1);
4     public static final int PLAYVIDEO = 3;
5
6     public void initMenu() {
7        original();
8
9        if(typeOfScreen == SHOWPHOTO)
10          this.addCommand(playVideoCommand);
11    }
12 }
```

(b) FeatureHouse–*Video*

Figure 8.8: *Bug location for Task 2 (bug highlighted).*

according class fits on one screen. However, this is a rather obvious bug, so it is not clear whether the shorter class helps finding this bug.

Task 5 In Task 5, we implemented the additional feature *AccessControl* to observe how participants of the FeatureHouse group can trace source code across different feature modules. The feature introduces rights to manage pictures, so if users have no rights to delete a picture, they cannot delete it. As bug, we use a wrong label for deleting a picture, such that the check for according rights is never executed and a user can delete a picture without according rights (Figure 8.11). The definition of the correct label is in another class, so participants have to look at two classes to locate the bug. In the FeatureHouse version, the two classes are located in different feature modules, which might slow down participants.

In addition to the five maintenance tasks, we designed a warming up task to let participants familiarize with the experimental setting. In this task, participants should count

```
1 public class MediaController extends MediaListController {
2 // 10 additional lines of code
3    public boolean handleCommand(Command command) {
4 // 215 additional lines of code
5       } else if (label.equals("View Favorites")) {
6          showMediaList(getCurrentStoreName(), false, false);
7          ScreenSingleton.getInstance()
8          .setCurrentScreenName(Constants.IMAGELIST_SCREEN);
9 // 257 additional lines of code
```

(b) Ifdef

```
1 class MediaController {
2    public boolean handleCommand(Command command) {
3 // 34 additional lines of code
4       if (label.equals("View Favorites")) {
5          showMediaList(getCurrentStoreName(), false, false);
6          ScreenSingleton.getInstance().
7          setCurrentScreenName(Constants.IMAGELIST_SCREEN);
8 // 7 additional lines of code
```

(b) FeatureHouse–*Favourites*

Figure 8.9: *Bug location for Task 3 (bug highlighted).*

```
1 public class MediaListController extends AbstractController {
2 // 124 additional lines of code
3    public void bubbleSort(MediaData[] medias) {
4       System.out.print("Sorting by BubbleSort...");
5       // TODO implement bubbleSort
6    }
```

(a) Ifdef

```
1 class MediaListController {
2 // 27 additional lines of code
3    public void bubbleSort(MediaData[] medias) {
4       System.out.print("Sorting by BubbleSort...");
5       // TODO implement bubbleSort
6    }
```

(b) FeatureHouse–*Sorting*

Figure 8.10: *Bug location for Task 4 (bug highlighted).*

the occurrence of a feature (ifdef version) or how often a class is refined (FeatureHouse version). The result of this task is not analyzed.

8.2.1.4 Analysis Methods

To analyze the data, we use descriptive statistics (mean, standard deviation, frequencies, and box plots) to describe response time, correctness, search behavior, and first action for a task. This way, we get an overview of how data are distributed. To evaluate the first research question, we analyze whether there is a difference in correctness (χ^2 test) and response time (either t test or Mann-Whitney-U test).

```
1  public class MediaController extends MediaListController {
2  // 14 additional lines of code
3     public boolean handleCommand(Command command) {
4        // #ifdef includeAccessControl
5        if (label.equals("Delete Label"))
6           if (!AccessController.hasDeleteRights()) {
7              gotoAccessDeniedScreen();
8              return true;
9  // 467 additional lines of code
```

(a) Ifdef

```
1  class MediaController {
2     public boolean handleCommand(Command command) {
3        if (label.equals("Delete Label"))
4           if (!AccessController.hasDeleteRights()) {
5              gotoAccessDeniedScreen();
6              return true;
7  // 16 additional lines of code
```

(b) FeatureHouse–*AccessControl*

```
8  public class MediaController extends MediaListController {
9  // 8 additional lines of code
10    public boolean handleCommand(Command command) {
11 // 43 additional lines of code
12       /* Case: Delete selected Photo from recordstore */
13       } else if (label.equals("Delete")) {
14          String selectedMediaName = getSelectedMediaName();
15 // 169 additional lines of code
```

(c) FeatureHouse–*Base*

Figure 8.11: *Bug location for Task 5 (bug highlighted).*

For the second research question, we compare the frequencies of local and global search within groups and between groups with a χ^2 test. For the third research question, we can either use a qualitative analysis, or compare frequencies of different actions with a χ^2 test or Fisher's exact test.

8.2.2 Pilot Study

To evaluate the feasibility of our design and give suggestions how to analyze and interpret the data to evaluate our research questions, we conducted a pilot study. Our participants were 8 undergraduate and graduate students from the University of Passau, who were enrolled in the course *Contemporary Programming Paradigms*, in which modern programming techniques, such as preprocessors and FeatureHouse, were taught. Thus, participants have the necessary knowledge to complete the tasks. All were aware that they took part in an experiment and that their performance does not affect their grade for the course. Participants volunteered and did not receive compensation for their participation.

To create two comparable groups, we applied our programming-experience questionnaire before the experiment. Unfortunately, not all participants who completed the questionnaire showed up for the experiment. Thus, both groups differ in size and their

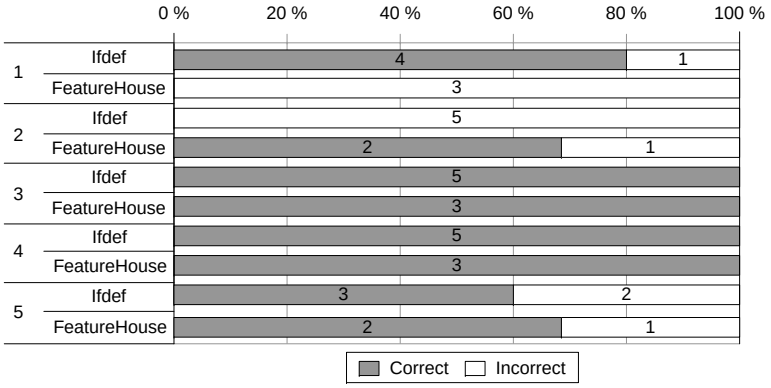


Figure 8.12: *Number of correct answers per group and task.*

programming experience. The ifdef group consists of 5 participants and is more experienced than the FeatureHouse group, which consists of 3 participants.

We conducted the experiment at the University of Passau in one computer lab instead of a lecture session. Before the experiment, we gave participants an introduction about what to expect. After all questions were answered, participants started to work on the tasks on their own. One to two experimenters checked that participants worked as planned. On two occasions, participants talked to each other, until the experimenter reminded them to work for themselves.

8.2.2.1 Experiment Results

In this section, we demonstrate how the results can be analyzed. The purpose of the analysis is not to evaluate the research questions, for which our sample is to small and groups are too heterogeneous.

First, we evaluate program comprehension by analyzing correctness, response time, search behavior, and first action for each task. To separate reporting data from interpreting them, we only report the data here and discuss them in Section 8.2.2.2, in which we also discuss the feasibility of our design.

Correctness First, we look at correctness. In Figure 8.12, we give an overview of the number of correct solutions. Task 3 and 4 appear to be easy, because all participants found the correct solution. The first task appears to be too difficult for the FeatureHouse group, because no participant found the correct solution. The same counts for the second task for participants of the ifdef group.

Response Time Second, we look at the response times. In Table 8.4, we show how long participants needed to solve each task and all tasks together (in minutes). For most of the tasks, the ifdef group was faster. Only for the second task, the FeatureHouse group was faster—one participant of the ifdef group needed 48 minutes for this task. The difficulty seems to vary, because the response times differ between tasks.

Task	Group	RT	Std	Min	Max
1	Ifdef	12.41	4.99	3.86	16.17
	FeatureHouse	14.03	5.37	7.84	17.42
2	Ifdef	22.79	15.34	9.53	48.14
	FeatureHouse	13.06	1.92	10.86	14.41
3	Ifdef	8.2	1.05	7.29	9.49
	FeatureHouse	12.77	3.78	8.98	16.53
4	Ifdef	4.16	2.34	2.14	7.86
	FeatureHouse	9.53	2.68	6.47	11.42
5	Ifdef	7.27	3.35	2.95	12.27
	FeatureHouse	12.38	6.08	6.08	18.08
All	Ifdef	54.83	9.97	42.17	66.58
	FeatureHouse	61.77	7.81	53.99	69.60

RT: response time in minutes, Std: standard deviation, Min: fastest response time, Max: slowest response time, All (last row): response time for all task combined.

Table 8.4: *Response times of participants per task.*

Search Behavior In Table 8.5, we show how often participants used the search feature (local, global, and combined). Participants of the ifdef group used the search considerably more often than participants of the FeatureHouse group. For the local search, participants always used it more often than the global search.

First Action In Table 8.6, we summarize how participants started to solve a task. Participants of the ifdef group most often used a global search to find code fragments of the relevant feature, whereas participants of the FeatureHouse group most often opened a file in the relevant feature. Additionally, in tasks where a label of a button is mentioned in the bug description, some participants searched for that label. However, they did not start to search for the label in the first task where it is mentioned (Task 2), but only for the subsequent tasks. Furthermore, two participants of the FeatureHouse group started in a wrong feature (*SortPhoto*), most likely, because *SortPhoto* sounds relevant for the task (feature *Sorting* is the correct one).

Opinion of Participants Regarding the opinion of participants, we find a tendency that the ifdef group found the tasks easier to solve, except for Task 2. For motivation, there is a tendency that participants of the ifdef group are more motivated to solve a task, which aligns with the comments of two participants of the FeatureHouse group, who were unhappy to be in that group. Thus, the FeatureHouse version appears more difficult to participants and they did not like it. This can affect their performance, such that they work slower [Mook, 1996].

Task	Group	Local	Global	Combined
1	Ifdef	166	21	187
	FeatureHouse	32	13	45
2	Ifdef	152	25	177
	FeatureHouse	28	13	41
3	Ifdef	106	11	117
	FeatureHouse	39	19	58
4	Ifdef	21	5	34
	FeatureHouse	16	7	23
5	Ifdef	73	8	91
	FeatureHouse	25	12	37

Table 8.5: *Search behavior of participants per task.*

Task	Group	Open file in:			Global search for:	
		base	relevant feature	wrong feature	relevant feature	label
1	Ifdef	-	-	-	5	-
	FeatureHouse	1	1	-	1	-
2	Ifdef	1	-	-	4	-
	FeatureHouse	-	2	-	1	-
3	Ifdef	-	-	-	3	2
	FeatureHouse	-	1	-	-	2
4	Ifdef	-	-	-	2	3
	FeatureHouse	-	-	2	-	1
5	Ifdef	-	-	-	5	-
	FeatureHouse	-	2	-	1	-

Table 8.6: *First action participants used to solve each task.*

8.2.2.2 Interpretation

Since our sample is too small and the ifdef group is more experienced, we cannot meaningfully interpret the effect of physically and virtually separated concerns. Except for Task 2, the faster response time of the ifdef group could be caused by the higher experience. However, in Task 2, the FeatureHouse group was faster. Thus, in this specific case (i.e., the class containing the bug fits on one screen, less experienced participants), physically separated code appears to be beneficial for program comprehension; the opportunity to compare similar code fragments does not seem to be a benefit for program comprehension.

Regarding the search behavior, we found that participants of the ifdef group used the search function considerably more often than participants of the FeatureHouse group. Additionally, all participants used the local search more often than the global search. There are two interesting facets regarding the search behavior of the FeatureHouse group.

First, for the second task, in which the class containing the bug consists of only few lines, participants used the global search more often. Second, for the last task, in which two classes in two different folders needed to be located to find the bug, the global search is used only half as much as the local search (similar to the search behavior for the other tasks). Thus, this tracing task seems to have comparable effort as the other tasks.

In Table 8.6, we show the first action of each participant for solving a task. Participants of the ifdef group most often searched for feature code with a global search, whereas participants of the FeatureHouse group opened a file in the relevant feature (or features that appear relevant). Thus, it appears as if participants use different strategies to solve a task.

The interpretations are a suggestion how to draw conclusions from data rather than actual conclusions for our research hypotheses. We need to replicate the experiment with more participants to confirm the results and conclusions.

8.2.2.3 Feasibility

With our pilot study, we found evidence about the feasibility of our design. Participants always understood the tasks and questionnaire and knew what they had to do. Only on two occasions, participants talked to each other, but the experimenter reminded them to work for themselves. Furthermore, two participants mentioned being unhappy to be in the FeatureHouse group. Thus, when conducting the experiment, participants of the FeatureHouse group should be sufficiently motivated about the benefits of FeatureHouse. Besides that, no problems occurred. Thus, the task descriptions and questionnaires seem to be clear to participants.

However, we found that two tasks (3 and 4) appear to be too easy, because all participants solved it correctly. Hence, when replicating the experiments, we should increase the difficulty of these tasks. For example, for Task 3, providing the label might have made the task too easy, because it occurs only 2 times in the complete project. For Task 4, we can provide an erroneous implementation of bubble sort, instead of a `TODO` in the empty method body.

Additionally, one participant of the ifdef group needed 48 minutes to complete this task, which is most of the time for the complete experiment (66 minutes). Thus, he might be fatigued or unmotivated to solve the remaining tasks. Hence, in future experiments, we can set a time limit for each task or give recommendations how long each task should last.

8.2.3 Threats to Validity

One threat is how we obtained the physically separated version of MobileMedia. It was derived from the ifdef version by refactoring. Although the refactorings and the resulting code have been reviewed carefully, it is unclear whether designing and implementing a system like MobileMedia from scratch in a feature-oriented way would have led to a different, more favorable decomposition, possibly making use of more effective modularization patterns. Exploring such patterns and related anti-patterns empirically is an avenue of further work.

A second threat is caused by the sample. When comparing techniques, we have to ensure that participants have comparable familiarity. Otherwise, we would measure differences in familiarity, not in the comprehensibility of both techniques. To control this threat, we recruited students from a course in which FeatureHouse and preprocessors were taught. Thus, we can assume that all participants have comparable knowledge of the evaluated techniques.

Last, our sample is too small to draw sound conclusions regarding our research questions. Thus, we used the data as evidence for the feasibility of our design and for suggestions regarding analysis and interpretation, instead of evaluating our research questions. Furthermore, the FeatureHouse group is less experienced than the ifdef group and mostly unhappy to work with the FeatureHouse version. Thus, worse program comprehension of the FeatureHouse group may be caused by lower experience or happiness, not the underlying technique. In future experiments, both groups should have comparable programming experience and comparable preferences of FeatureHouse and preprocessors.

8.2.4 Related Work

When object-oriented programming emerged, researchers conducted studies to evaluate its effect on comprehension and maintainability. For example, Daly and others analyzed how the depth of inheritance affects development time of maintenance tasks [Daly et al., 1995]. They found that participants maintaining a system with a depth of three were faster than participants maintaining a system without inheritance. Henry and others compared the performance during maintenance tasks of a procedural implementation and an object-oriented version [Henry et al., 1990]. Participants were faster and made fewer errors with the object-oriented version. Both experiments evaluate a facet of a programming paradigm that was new and was supposed to improve comprehension and maintenance, as in our work. The difference is that we focus on feature-oriented programming, not object-oriented programming.

Besides our own work and tools (cf. Chapter 7), there is work by Robillard and Murphy on concern graphs [Robillard and Murphy, 2002, 2003]. Their tool FEAT allows developers to annotate classes, methods, or fields that belong to one concern. The authors evaluated the usefulness of FEAT and found that developers could trace source code and that their concept scales to industrial-sized programs.

8.2.5 Conclusion

Separation of concerns is supposed to improve program comprehension. However, there are no empirical studies that evaluate comprehensibility of physically and virtually separated code. To close this gap, we presented an experimental design to compare program comprehension of physical and virtual separation of concerns. We refactored the ifdef version of MobileMedia (virtually separated) to a FeatureHouse version (physically separated). In a pilot study with eight students, we showed the feasibility of our design. Our next step is to replicate the experiment with a larger sample. Furthermore, we encourage other researchers to replicate our experiment.

8.3 Is the Derivation of a Model Easier to Understand Than the Model Itself?

This section shares content with ICPC'12 paper "Is the Derivation of a Model Easier to Understand than the Model Itself?" [Feigenspan et al., 2012a]

Our last ongoing project deals with comprehension of models. Specifically, we are targeting the question whether presenting a derivation of a complex model improves comprehensibility, compared to presenting it all at once. Complex architectural models (typically represented as graphs, in which nodes are components and edges are connectors) encode a substantial amount of expert domain knowledge [Riché et al., 2010]. Without such knowledge, it is difficult to understand a model.

A host of researchers in the past 15 years have suggested another way to explain software architectures: Instead of presenting an architectural model as a fully-completed diagram, they start from an elementary model that is easily understood, and apply a series of semantics-preserving refinements and optimizations to transform the simple model into the complex model that represents the complete architecture [Riché et al., 2010]. Each transformation, in isolation, can be easily grasped and represents a fundamental mapping that arises in architectural designs in that domain. These mappings, which represent a computational abstraction with one of its implementations, correspond to a law in that domain.

To evaluate whether deriving a model improves its comprehensibility, compared to presenting it at once, we conducted a series of controlled experiments. For better overview, we present a summary of all three experimental runs in Table 8.7.

8.3.1 Objective

We believe that the derivation of a model is easier to understand than the model itself. The line of argumentation is the same as for the benefit of physically separated code on program comprehension: We need working memory to perceive information, which has limited capacity. If we have a complex model with many items, we cannot understand it all at once, but look only at parts of the model to comprehend it. Thus, when we present an architectural model piece by piece, we do not have to absorb more information at a given time than we can. Hence, we defined the following research hypotheses:

RH1: The derivation of model is easier to memorize than the model itself.

RH2: The derivation of model is easier to comprehend than the model itself.

RH3: The derivation of model is easier to modify than the model itself.

To evaluate our research hypotheses, we conducted a pilot study (to test our material) and two controlled experiments.

What?	Run		Section
Objective	Pilot	Test material and setting	8.3.2
	Exp1	Evaluate whether derivation of a model is easier	8.3.3
	Exp2	to manage	8.3.4
Material	Pilot	Gamma (parallelized database machine), Synchronous Upright (crash-fault-tolerance server without recovery)	8.3.2.1
	Exp1	Synchronous Upright	8.3.3.1
	Exp2	Asynchronous Upright (with recovery)	8.3.4.1
Participants	Pilot	4 graduate students (either software-engineering or database domain)	8.3.2.3
	Exp1	12 undergraduate students of software-engineering and database course	8.3.3.1
	Exp2	10 undergraduate students of software-engineering course All participants from the University of Texas at Austin	8.3.4.1
Tasks	Pilot	Redraw and modify models, 5 multiple-choice comprehension questions	8.3.2.2
	Exp1	Redraw and modify model, 5 multiple-choice comprehension questions	8.3.3.1
	Exp2	Redraw and modify model, 7 multiple-choice comprehension questions	8.3.4.1
Execution	Pilot	One room, all material on paper, participants looked at models themselves	8.3.2.4
	Exp1	One room, all material on paper, 2 subsequent sessions with different experiment conditions, expert presented models	8.3.3.1
	Exp2	One room, all material on paper, 2 parallel sessions with different experiment conditions, experts presented models	8.3.4.1
Analysis	Pilot	Correctness of answers, perceived difficulty and motivation	8.3.2.5
	Exp1		8.3.3.2
	Exp2		8.3.4.2
Result	Pilot	Two models too much, going through models alone too tedious	8.3.2.5
	Exp1	No difference between presenting derivation vs.	8.3.3.2
	Exp2	complete model	8.3.4.2

Table 8.7: *Model comprehension: Experiments in a nutshell.*

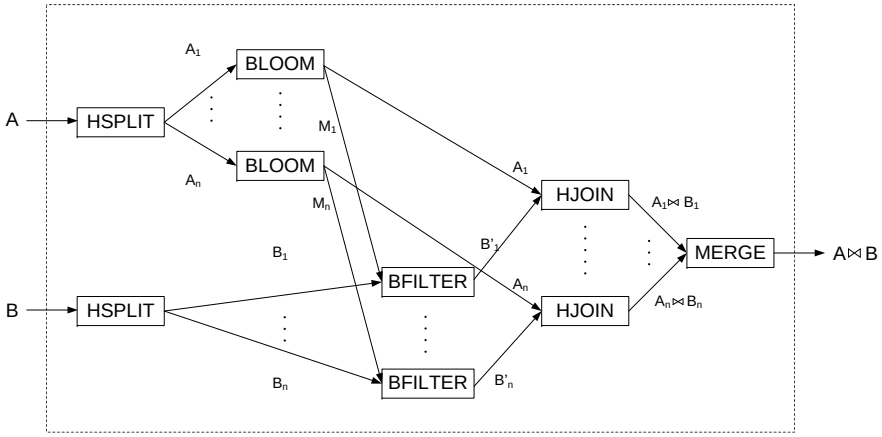


Figure 8.13: *Parallel hash join in Gamma.*

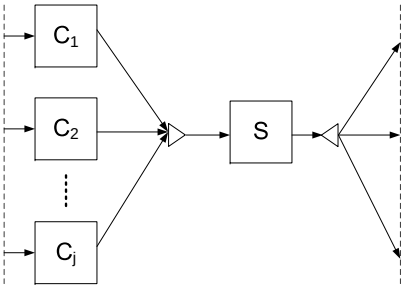
8.3.2 Pilot Study

The pilot study tested our experimental material and setting (e.g., to avoid ambiguous formulations of the tasks).

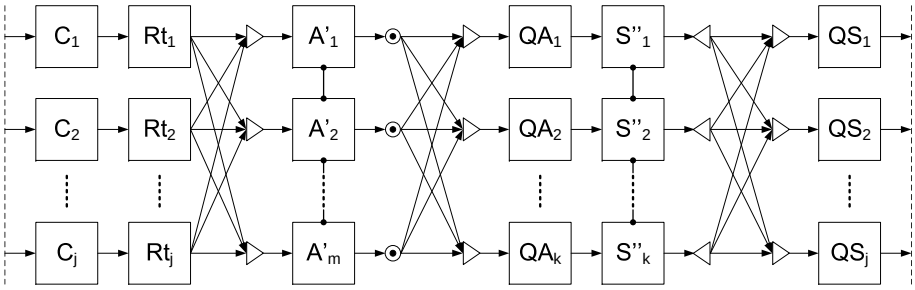
8.3.2.1 Material

We used two systems: Gamma (a relational database machine) and Upright (a synchronous crash-fault-tolerance server). First, Gamma is a database machine, which is known for its innovative parallelization of hash joins [Dewitt et al., 1990]. In Figure 8.13, we show how Gamma implements parallel hash joins. The relations to be joined are first split into substreams using a hash function (HSPLIT). For each tuple of the first relation, the join key is hashed and stored in a bitmap M (BLOOM). Then, for each tuple of the second relation, the join key is hashed and compared with the join-key hashes stored in M. All tuples that have no entry of the hashed join key in M cannot be joined and are deleted from the second relation (BFILTER). Then, the remaining tuples of the second relation are joined with the tuples of the first relation (HJOIN). Finally, the joined substreams are merged and the result is returned. We used the hash join in Gamma, because it is simple enough to understand in a limited amount of time, but not too simple to understand at first sight.

Second, we used Upright, a synchronous crash-fault-tolerance server [Riché et al., 2010]. Client messages are processed sequentially by a server (Figure 8.14a). Through server and client-message replication, a certain number of server crashes can be tolerated and still provide the image of a single server processing client messages (Figure 8.14b). A client C_j sends a message to a routing box Rt_j, which routes a copy of the message to all agreement nodes (A_1 to A_n). As part of the agreement protocol, each agreement node votes by broadcasting the message that it believes should be processed next to all quorum nodes (QA_1 to QA_k). When the quorum nodes have received a sufficient number of identical messages, that message is sent to each server replica (S_1 to S_k). Each server processes the same message, and sends its response to a quorum node (for a message



(a) Upright before transformations.



(b) Upright after transformations.

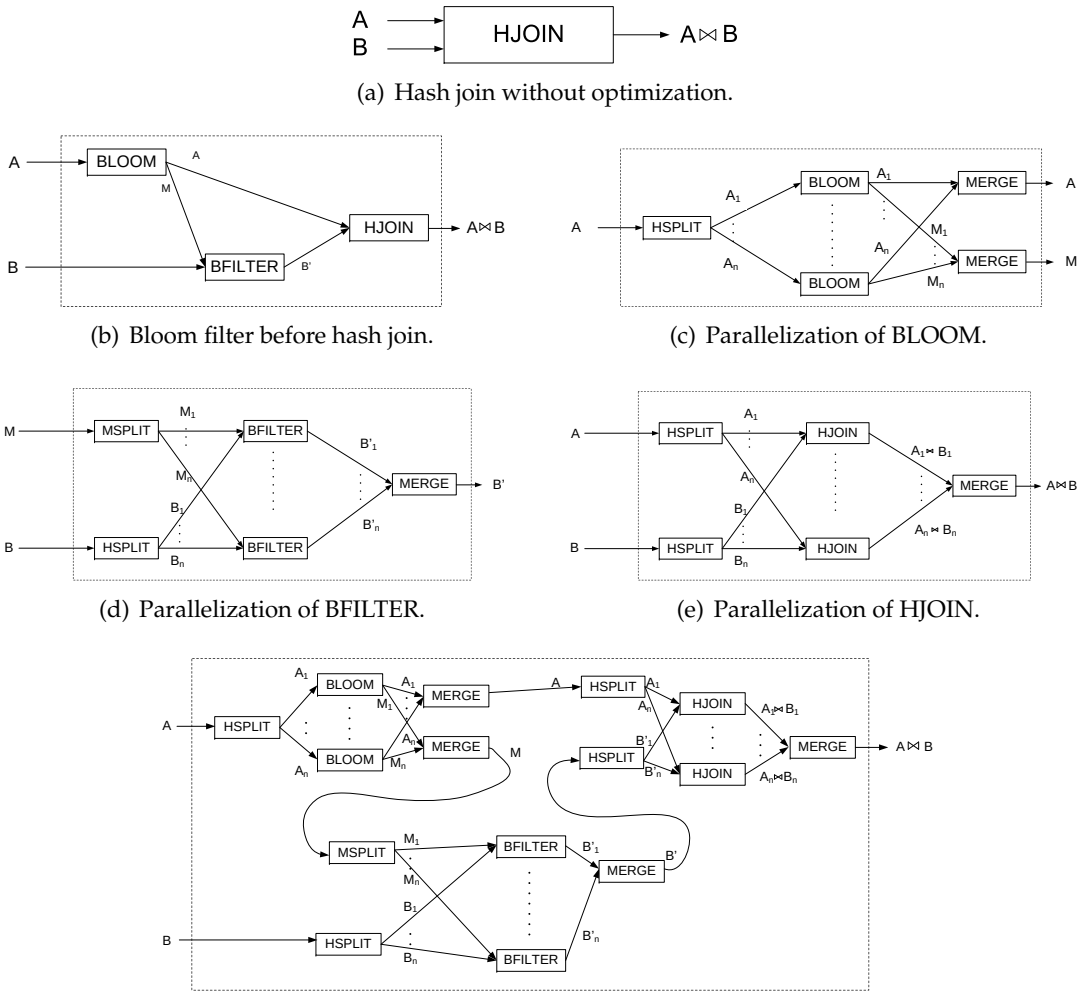
Figure 8.14: *Synchronous crash-fault-tolerance server before and after transformations.*

from C_j, the quorum node would be QS_j) that is located in front of the receiving client. A quorum is taken, and a single response is returned to the client.

For both Gamma and Upright, we found a derivation of their models. For demonstration, we illustrate the derivation of Gamma in Figure 8.15. We use Gamma, because it is easier to understand than Upright and requires fewer steps. We start with a hash join without optimization (Figure 8.15a). Then, we introduce a bloom filter before the hash join, in which we delete tuples of B that cannot be joined with A (Figure 8.15b). In the next steps, we parallelize each box by splitting the stream of tuples into substreams, processing each substream, and merging each substream (Figure 8.15c to 8.15e). We put the parallelized boxes together (Figure 8.15f) and, by deleting unnecessary merge and split operations, we obtain the final architecture of Gamma (cf. Figure 8.13).

For Upright, we started with a simple client-server abstraction of j clients $C_1 ... C_j$ sending messages to a single, state-free server, shown in Figure 8.14a. By applying a sequence of semantics-preserving transformations, we derive the architecture of Upright as shown in Figure 8.14b. In summary, the transformations contain adding new nodes step by step and then replicating the nodes to tolerate crashes. For more information on the transformations, we recommend the article by Riché and others [Riché et al., 2010].

For each model, we created two sets of slides—one with the derivation, one with the complete model—with explanations of the boxes. One group of participants received the set with the derivation, the other group the set with the complete model, both on paper. We did not use PROPHET, because it does not support the tasks, which we present next.



(a) Hash join without optimization.

(b) Bloom filter before hash join.

(c) Parallelization of BLOOM.

(d) Parallelization of BFILTER.

(e) Parallelization of HJOIN.

(f) Putting parallelized boxes together.

Figure 8.15: *Derivation of Gamma.*

8.3.2.2 Tasks

We created three tasks to test each hypothesis. First, participants should redraw the models of Gamma and Upright. Second, participants should answer multiple-choice comprehension questions (five questions per model), for example:

> Why are A nodes required to communicate amongst themselves?
>
> a) To combine messages from the clients to reduce server load
> b) The A nodes split the load coming from the clients
> c) The A nodes run a decision making protocol that can tolerate if some of them crash

Third, participants should modify the models of Gamma and Upright. For Gamma, they should delete the BLOOM box (requiring to also delete the BFILTER box). For Up-

right, they should add a server replica, such that the system can tolerate one more server crash. The tasks were identical for the derivation and the complete model.

Finally, we asked our participants to estimate the difficulty of the tasks and their motivation to solve them on a five-point Likert scale.

Both, the first and third task, require participants to draw models, which PROPHET does not support currently. We could have presented the model and second task with PROPHET; however, it is unclear whether using two different media (computer, paper) affects cognitive resources and bias results. Thus, we conducted the complete experiment on paper.

8.3.2.3 Participants

As participants, we recruited four male PhD students from the University of Texas at Austin. Two were working in the empirical software-engineering domain, two in the database domain. All volunteered and did not receive any compensation for their participation. All participants had comparable familiarity with databases, crash-fault-tolerance servers, pipe-and-filter architectures (a class of architectures to which Gamma and Upright belong), and modeling, which we assessed with a questionnaire. We did not assess programming experience, because none of the tasks required programming. However, in future work, it is interesting to develop a questionnaire that assesses experience with modeling, since it is also an often evaluated facet in empirical software engineering (e.g., UML modeling).

8.3.2.4 Experiment Execution

We conducted the pilot study in November 2011. Participants completed the familiarity questionnaire first. After an introduction, we handed out the slides for Gamma. When participants finished reviewing the slides, we distributed the tasks one at a time. We recorded the time participants needed to finish a task manually. Then, we gave participants the questionnaire regarding motivation and difficulty. After a few minutes break, we repeated the same procedure with Upright. There are no deviations to report.

8.3.2.5 Results and Consequences

The pilot study yielded two important results. First, two models were too much. Although we included a short break, participants were fatigued when working with the second model.

Second, we noticed (and participants told us) that going through the models on their own is tedious. There were many slides, especially for the derivation. Furthermore, participants with the derivations felt rushed, since the other participants had to wait for them before they could start the tasks.

As consequences, we henceforth used only one model. We selected Upright, because it is more complex and has many elements, such that working memory capacity is exceeded and we could we assume that the benefit of the derivation would be more evident (cf. Section 8.3.1). Additionally, we changed the presentation of the model and the

derivation, respectively, such that an expert on crash-fault-tolerance servers presents the slides in the subsequent experiments.

8.3.3 Experimental Run 1

With our first experimental run, we evaluated our research hypotheses. The material was the same as for the pilot study, except for the two changes we described.

8.3.3.1 Experiment Design and Execution

Our participants were undergraduate students from the University of Texas at Austin who were enrolled either in a database or software-engineering course. The mean age of participants who worked with the complete model was 23, the mean age of participants who worked with the derivation was 25.7. One female participant worked with the derivation. Both groups estimated their experience with crash-fault-tolerance servers as low (2 for the derivation, 1.5 for the complete model; both on a five-point scale). The same counts for their experience with modeling (3 for the derivation, 2.5 for the complete model). Participants volunteered and were rewarded with food and beverages. Participants were aware that their performance in the experiment did not affect their grade and that they could leave any time.

We had two appointments for the experiment. In the first, we presented the derivation, in the second, the complete model. Participants could choose the appointment to their convenience, which lead to different group sizes: Eight participants worked with the derivation, four with the complete model. We prepared a booklet with all tasks and questionnaires, which we distributed at the beginning of the experiment. After an introduction, an expert on Upright presented either the derivation or the complete model. Then, participants solved the tasks. For each task, participants had a time limit (based on the pilot study), after which they had to turn to the next task. The time limit was large enough so that no participant experienced time pressure.

One experimenter checked that participants worked as planned. There was one deviation: For one participant in the derivation group, the first and third tasks were in reverse order. Since we cannot measure the performance for these tasks, we excluded this participant from the analysis.

8.3.3.2 Analysis and Interpretation

To analyze the first and third task, we counted the number of elements that did not belong to the model, were missing, or were in the wrong order. Hence, the larger the number, the more errors participants made. For the second task, we counted the number of correctly solved comprehension questions. The differences between groups are small or non-existent, as we show in Table 8.8. For the first task, the group with the derivation made one error more than the group with the complete model. For the second task, both showed the same performance. For the third task, the difference is smaller than one error. A Mann-Whitney-U test adjusted for small sample sizes revealed that the differences

Group	Task	Median	Min	Max	U value	significant?
Derivation	1	4	2	4	7	no
Complete		3	2	5		
Derivation	2	4	2	6	10	no
Complete		4	4	5		
Derivation	3	0	0	1	19	no
Complete		0.5	0	1		

Table 8.8: *Experimental run 1: Overview of correctness of solutions.*

between groups are not significant for any of the tasks [Nachar, 2008]. Hence, we cannot accept our research hypotheses.

At first sight, this result means that the derivation does not provide a benefit, compared to presenting the complete model. However, taking a closer look at Upright, we see groups of similar elements. We have a group of clients, routing boxes, agreement nodes, quorum nodes, and servers. Hence, we have six different groups of boxes. It is possible that participants looked at the *group of boxes*, not the single boxes, which is called *chunking* [Miller, 1956]. Thus, working memory capacity might not have been exceeded, which means that the model could have been too simple to show a benefit of derivation. Looking at the estimation of difficulty for each task, it was perceived as medium to easy, except for the first task (difficult). Looking at the correctness of the first task, participants only made three to four errors. Thus, we think that the model was too easy to reveal a benefit of derivation. Hence, we conducted a follow-up experiment with a more complex model.

8.3.4 Experimental Run 2

To get more data about the effect of the derivation of a model, we conducted a second run of our experiment.

8.3.4.1 Experiment Design and Execution

In our second run, we extended Upright with a recovery feature, shown in Figure 8.16. Via a backward loop, a server replica can now recover from a crash. To this end, it sends a message to the agreement nodes via quorum nodes asking for the correct timestamp. Furthermore, the server replicas communicate amongst each other to get the correct status.

With the recovery feature, we included two more comprehension questions, so we had seven questions. Other than that, the experimental material was the same as before.

We recruited different students from the same software-engineering course as before; six worked with the derivation, four with the complete model. The mean age was 24.3 for the derivation group, and 21.8 for the other group. One participant in each group

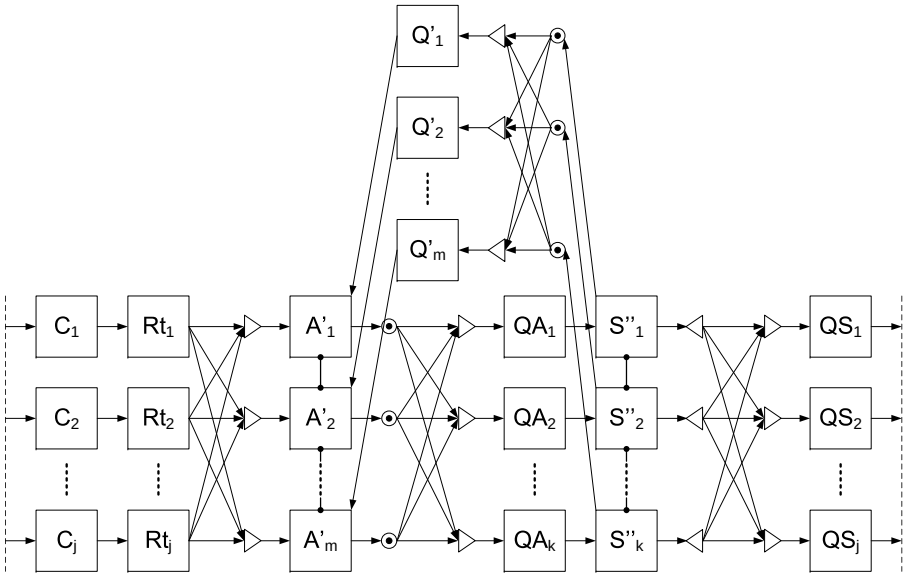


Figure 8.16: *Asynchronous crash-fault-tolerance server.*

Group	Task	Median	Min	Max	U value	significant?
Derivation	1	2.5	1	4	10.5	no
Complete		2.5	1	3		
Derivation	2	5	2	6	11.5	no
Complete		5	2	7		
Derivation	3	3	1	4	8	no
Complete		3	1	3		

Table 8.9: *Experimental run 2: Overview of correctness of solutions.*

was female. Participants estimated their experience with crash-fault-tolerance servers and modeling as low (1 and 3 for the derivation, 1.5 and 2.5 for the complete model).

The experimental sessions were now held in parallel. Hence, an additional expert on crash-fault-tolerance servers explained the derivation. No deviations occurred.

8.3.4.2 Analysis and Interpretation

To evaluate whether a task was solved correctly, we used a four-point scale for the first and third task. A solution could either be completely correct (4), almost correct (3), correct to some extent (2), or completely wrong (1). An expert evaluated to which category a solution belonged. In Table 8.9, we show an overview of the correctness. The medians for both groups are the same. Furthermore, the modification task seems to be easier than the memorization task. A Mann-Whitney-U test revealed no significant differences. Hence, we cannot accept our research hypotheses.

So, although we increased the complexity of the underlying model, we still did not observe a significant difference in the performance of participants. The perceived difficulty of the tasks is comparable with that of the first experimental run (medium to easy difficulty). Hence, the model might still have been too simple to show a benefit of its derivation.

8.3.5 Combining the Results

So far, we conducted two controlled experiments to evaluate our research hypotheses (i.e., that the derivation of a model is easier to memorize, comprehend, and modify than the model itself). We could not accept our research hypotheses, for which we suspect three possible reasons: First, the models were too simple; second, understanding the derivation required too much cognitive resources; third, there is no benefit of derivation.

First, the model of the crash-fault-tolerance servers could be too simple. In the first experimental run, the elements of the model could have been grouped, such that the working memory capacity was not exceeded. We made the model more complex in the second experimental run. However, with grouping, the number of elements still lies in the upper bound of the working memory capacity. Thus, we believe that replicating the experiment with a more complex model would reveal more insights into the relationship of the size of a model and the effects of its derivation.

Second, participants who worked with the derivation had to understand several transformations. It is possible that understanding the transformations required too much cognitive resources, such that the benefits of the derivation are erased. To test this hypothesis, we would have to conduct another experiment.

Third, it could also be possible that in our context (i.e., with students and our certain model), there simply is no benefit of using a derivation. Hence, whether we explain a model to students incrementally or all at once, might not matter—students might gain a comparable level of understanding with both approaches.

The bottom line is that we need to conduct further experiments to gain better insights into the effects of derivation on memorizing, understanding, and modifying models. To gain deeper insights, we are currently planning an experiment in which one group of students should implement the derivation of Gamma, the other group the complete model. As dependent variable, we plan to measure development time and code quality.

8.3.6 Threats to Validity

There are several threats to validity for both experiments. First, we used convenient sampling to create our samples, which might lead to incomparable groups. However, both groups have comparable familiarity with relevant techniques, such as modeling, which we confirmed with a questionnaire. Furthermore, group sizes are different, because we could not assign participants to the appointments. Since both groups have comparable familiarity with relevant techniques, the different group size should not have affected our result significantly.

Second, we did not test the memory skill of participants. However, memory skills can have a significant influence on our result, especially on the performance for the first task. Unfortunately, without a memory-skill test, there is no way to control this threat.

Third, we used multiple-choice questions to measure comprehension. This could have made the tasks too easy, such that we could have measured how well participants are able to rule out wrong answers, not how well participants understood the model. However, we made sure that all possible answers sounded plausible and had about the same length and detail. Furthermore, none of the participants noted that there were obvious wrong answers. Hence, we believe that we sufficiently controlled this threat.

8.3.7 Related Work

There is a large body of work dealing with comprehension of UML models. For example, Lange and Chaudron analyzed whether defects in UML models are found by developers who implemented the model [Lange and Chaudron, 2006]. To this end, participants saw fragments of UML diagrams and answered multiple-choice questions on how they would implement the model. The result is that some defect types are almost always detected, whereas other types are almost never detected. Sharif and Maletic compared how different layouts of UML models affect their comprehension [Sharif and Maletic, 2009]. Based on the answers of participants regarding comprehension questions, authors found a difference in the comprehension of different layouts. Genero and others assessed how size and structural complexity of UML models affect their comprehension [Genero et al., 2007]. Participants answered comprehension and modifying questions. Based on the answers, authors obtained a prediction model based on size and structural complexity of UML models for comprehension and modification time. All papers analyze how different facets affect comprehension of UML models, but none considered architectural models.

8.3.8 Conclusion

Based on limits to working memory capacity, we believed that the derivation of a model is easier to comprehend than a model itself. To evaluate our belief, we conducted two controlled experiments, in which we compared how participants memorize, understand, and modify the derivation of a model and a complete model. Our experiments found no evidence to support it.

In future work, we plan to let participants implement an architecture and analyze whether implementing its derivation has a benefit compared to implementing it at once. This way, we hope to get more insights into whether and how the derivation of a model affects its comprehension.

8.4 Applying our Framework

So how did our framework supported us for the experiments? First, we consulted the list of confounding parameters to decide what parameters are relevant and how we can control them. Even for our project regarding model comprehension, the list was useful, because model comprehension is closely related to program comprehension. Thus,

we could save time during the planning phase of our experiments. To avoid redundancy, we do not present the most important confounding parameters here, but refer to Appendix 10.2, in which we present an overview of all confounding parameters for all experiments.

Second, for the first two experiments, we measured programming experience according to our questionnaire. Since we did not need to develop the questionnaire from scratch, we saved time during planning our experiment. For the third experiment, we did not use the questionnaire. However, our experience during creating the questionnaire helped us to ask participants the right questions to assess their background.

Last, for the first two experiments, we used PROPHET to present source code, tasks, and questionnaires to participants. Thus, we did not have to develop our own tool infrastructure, but could easily customize the presentation of material and use it for replication. Furthermore, for the second experiment, the experiment designer was not present, but a colleague conducted the experiment. There were no problems during the conduct, indicating that PROPHET is intuitive and suitable for replicating experiments. For the third experiment, we could not use PROPHET, because it does not support drawing or modifying images; if it did support working with images, it would have eased the presentation of material, time measurement, and data logging.

Thus, our framework supports us in our ongoing projects, without being developed specifically for that purpose. This indicates that our framework is applicable in practice. Hence, we made an important contribution to the empirical-software-engineering community.

Chapter 9

Conclusion and Future Work

Program comprehension is an important factor in the software life cycle, because maintenance programmers spend most of their time with understanding code, and maintenance is the main cost factor during software development. Thus, by improving program comprehension, we can save time and cost during software development and maintenance. To this end, new programming techniques and languages were developed, such as feature-oriented software development. Among others, it aims at improving program comprehension by separating code along features.

In this thesis, we set out to evaluate how feature-oriented software development influences program comprehension. To this end, we conducted a series of controlled experiments. However, during our research, we found that there is no common baseline for conducting controlled experiments. Thus, in parallel to our experiments, we set out to develop a framework to support researchers in conducting controlled comprehension experiments. Specifically, we defined two goals:

- A framework to support controlled program-comprehension experiments.
- A knowledge base regarding how feature-oriented software development affects program comprehension.

With the following contributions, we fulfilled our goals:

1. *Recommendation for measuring program comprehension.*
   To evaluate whether software measures are suitable indicators for program comprehension, we conducted a controlled experiment, in which we let two groups of participants solve comprehension tasks on two comparable versions of MobileMedia (a program for the manipulation of multi-media data on mobile devices) with considerably different software measures. We did not find a difference in program comprehension between both groups as the software measures suggested it. Thus, we cannot recommend to use software measures to assess program comprehension.

   Furthermore, we are currently evaluating whether we can use fMRI to measure program comprehension. In first pilot studies (without an fMRI scanner), we selected tasks and material to measure program comprehension based on fMRI. Currently, we are conducting measurements inside the scanner.

2. *Overview of confounding parameters and control techniques.*
   To reliably measure program comprehension, we need to identify and control for confounding parameters. To support researchers, we conducted a literature survey and analyzed how confounding parameters are currently managed. We identified 39 confounding parameters, but found that only a fraction of them are mentioned in each paper, indicating that researchers do not sufficiently consider them. We gave recommendations how to control for confounding parameters and how to describe them in a report. This way, we support researchers in creating sound and reliable results.

3. *Initial questionnaire to measure programming experience.*
   To control for the major confounding parameter, programming experience, we developed a questionnaire and evaluated it in a controlled experiment with 128 second-year students. To this end, we compared the answers of participants in the questionnaire with the performance in 10 comprehension tasks (i.e., determine the output of small source-code fragments). To select suitable questions, we computed correlations and stepwise regression. As a result, we identified two questions as indicator for programming experience, which we recommend to use when measuring programming experience. Additionally, with an exploratory factor analysis, we extracted a five-factor model that describes programming experience. Furthermore, our proceeding act as guidelines to develop similar questionnaires that measure different confounding parameters.

4. *Tool support for comprehension experiments.*
   We developed PROPHET to support researchers in planning, conducting, and replicating experiments. Based on our literature survey, we analyzed requirements for program-comprehension experiments and implemented PROPHET to fulfill these requirements. Experimenters can comfortably customize how participants see material and extend PROPHET to address currently not fulfilled requirements.

5. *Empirical evidence of how feature-oriented software development affects program comprehension.*
   In three controlled experiments, we evaluated whether and how background colors improve program comprehension in preprocessor-based software. For locating feature code, background colors have a positive effect; for maintenance tasks, background colors have at best no effect. Thus, highlighting feature code with background colors can improve program comprehension. We used our results to implement FeatureCommander, the prototype of an integrated development environment, which provides a consistent and customizable use of background colors.

   Furthermore, we described an experimental design to evaluate whether physically separated code affects program comprehension compared to virtually separated code. In a pilot study with 8 students, we evaluated our setting and give recommendations for its improvement.

6. *Reusable experimental designs.*
   Our experimental settings are designed to be reusable. We used common guide-

lines to present our set up, and made all material available at the project's website (`http://fosd.net/experiments`). Furthermore, we demonstrated where possible how our framework supported us with our experiments.

Thus, we made valuable contributions to the software-engineering domain and feature-oriented-software-development domain, which we plan to extend in future work.

Future Work

We are continuing our current projects presented in Chapter 8. Regarding functional magnetic resonance imaging, we are conducting the measurement inside the fMRI scanner. Furthermore, we are planning to replicate our experiment regarding physical and virtual separation of concerns with students from the course *Modern Programming Paradigms* that will be held in the winter term 2012/13 at the University of Magdeburg. Regarding model comprehension, we designed implementing tasks of Gamma, such that one group of students implements Gamma at once, and another group stepwise. This way, we analyze whether the derivation of a model affects its understanding on the implementation level.

Furthermore, we are continuing our work on programming experience. To this end, we are currently letting different students from four different universities complete the programming-experience questionnaire. So far, over 100 students completed the questionnaire. Our goal is to confirm the five-factor model of programming experience in a different sample, thus completing the next step toward a model of programming experience.

Since we recruited students in all experiments, we cannot generalize our results to expert. To do so, we need to conduct experiments with professional programmers, which is an interesting option for future work. Whenever we have necessary resources, we plan to recruit expert programmers to generalize our results.

Additionally, our work provides numerous extension points, which we discuss for each chapter.

1. *Software measures and program comprehension* do not seem to be related. However, we focused on feature-oriented software measures and used only one software system. To extend our work, we can compute other software measures to evaluate how they relate to program comprehension and use different software systems. Furthermore, we can evaluate how other concepts that are also often assessed with software measures, such as maintainability or design stability, relate to software measures. This way, we can get a deeper understanding of how software measures help to assess such software quality facets, which might help us to develop more reliable software measures.

   Additionally, we can evaluate how different tasks measure program comprehension. In our settings, we usually ask participants to fix a bug. However, there are also other tasks, such as implementing source code, memorizing and recalling source code, or filling in blank lines in source-code fragments. To the best of our knowledge, there is no empirical comparison of the reliability of these different tasks.

2. Numerous *confounding parameters for program comprehension* exist. With our review, we identified 39 of them in 7 journals and conferences of the last 10 years in the software-engineering domain. We can extend our work to other journals, conferences, issues, and domains. An especially interesting domain is psychology, because experiments have a long tradition, which might help us to get a broader overview of how to manage confounding parameters.

3. *Programming experience* is an important confounding parameter. With our questionnaire, we made an important step toward its measurement. However, we recruited students as participants, so our questionnaire is valid only for students. Furthermore, we can evaluate whether our questionnaire is also useful to evaluate programming experience of job applicants. In this case, we would have to evaluate whether self estimation is still a reliable indicator. Additionally, we can develop questionnaires for other confounding parameters, such as ability or familiarity with the study object or tools.

4. *PROPHET* supports researchers in planning, conducting, and replicating experiments. Due to its plug-in architecture, it can be extended with functionality. Thus, we can identify more requirements by extending our literature survey or by asking experimenters what functionality they need.

5. Regarding *background colors in preprocessor-based software*, we found a positive effect on program comprehension. With the tool FeatureCommander, we implemented customizable and consistent use of background colors. To evaluate the effectiveness of the implemented concepts in FeatureCommander, such as the interactive sidebars or the explorer view ordered by features, we can conduct controlled experiments.

Thus, our thesis constitutes a starting point for interesting research in the context of program comprehension.

Chapter 10

Appendix

10.1 Chapter 4: Checklist of Confounding Parameters for Program Comprehension

The check lists of Tables 10.1 and 10.2 can help researchers to control the influence of confounding parameters. We should note how we controlled for a parameter and why we selected a technique. Furthermore, we should note how and why we measured a parameter or ensured that its influence is controlled.

10.2 Confounding Parameters for Conducted Experiments

In Tables 10.3 and 10.4, we summarize how we managed confounding parameters in our experiments. Since the parameters and control techniques are similar for all our experiments, we summarize them to avoid redundancy. For better comprehensibility, we explain some entries. For example, we controlled for color blindness by keeping it constant. We selected this control technique, because colors blindness is rare in the population, so our results are applicable to a large part of the sample; thus, we do not limit external validity to much. To measure it, we asked participants whether they are color blind, because it reliable and easy to apply. For intelligence, we used randomization, because we do not have the time to conduct an intelligence test. Thus, we did not measure it, so the last two columns are empty. For programming experience, we applied matching, because it is the most important confounding parameter. We could not use it as independent variable, because we do not have the resources to recruit expert programmers. To measure it, we applied a questionnaire (i.e., a preliminary version of our questionnaire), because it is a reliable technique. Last, for instrumentation, we avoided bias due to unsuitable instruments, because it is the most reliable technique to control for it. To ensure that we avoided it, we conducted a pilot study, in which we evaluated the suitability of our instruments.

Parameter	Control technique		Measured/Ensured	
	How?	Why?	How?	Why?
Personal background				
Color blindness				
Culture				
Gender				
Intelligence				
Personal knowledge				
Ability				
Domain knowledge				
Education				
Programming experience				
Reading time				
Personal circumstances				
Attitude toward study object				
Familiarity with study object				
Familiarity with tools				
Fatigue				
Motivation				
Occupation				
Treatment preference				

Table 10.1: *Checklist of personal confounding parameters.*

Parameter	Control technique		Measured/Ensured	
	How?	Why?	How?	Why?
Subject related				
Evaluation apprehension				
Hawthorne effect				
Process conformance				
Study-object coverage				
Ties to persistent memory				
Time pressure				
Visual effort				
Technical				
Data consistency				
Instrumentation				
Mono-method bias				
Mono-operation bias				
Technical problems				
Context related				
Learning effects				
Mortality				
Operationalization of study object				
Ordering				
Rosenthal				
Selection				
Study-object related				
Content of study object				
Language				
Layout of study object				
Size of study object				
Tasks				

Table 10.2: *Checklist of experimental confounding parameters.*

Parameter	Control technique		Measured/Ensured	
	How?	Why?	How?	Why?
Personal background				
Color blindness	Constant	Color blindness is rare	Asked participants	Reliable
Culture	Constant	Limited resources	Participants from German universities	Mostly German students
Gender	Constant/Matching	Mostly male population	Asked participants	Reliable
Intelligence	Randomization	No resources for measurement		
Personal knowledge				
Ability	Randomization	No resources for measurement		
Domain knowledge	Constant	Limited resources	Asked participants	Reliable
Education	Constant	No training necessary to ensure familiarity with study object	Participants of same course	Comparable knowledge
Programming experience	Matching	Major confound, limited resources	Questionnaire	Reliable
Reading time	Ignored	Negligible influence		
Personal circumstances				
Attitude toward study object	Analyzed afterwards	Can change during experiment	Questionnaire	Reliable
Familiarity with study object	Constant	No training necessary to ensure familiarity	Participants of same course	Have same familiarity
Familiarity with tools	Constant	No training necessary to ensure familiarity	Specific tool	Easy to get familiarized
Fatigue	Avoided	Reduce stress for participants	Short session duration	Humans can work concentrated for 90 minutes
Motivation	Analyzed afterwards	Can change during experiment	Questionnaire	Reliable
Treatment preference	Analyzed afterwards	Can change during experiment	Questionnaire	Reliable

Table 10.3: *Background colors: Personal confounding parameters.*

Parameter	Control technique		Measured/Ensured	
	How?	Why?	How?	Why?
Subject related				
Evaluation apprehension	Avoided	To get actual performance	Anonymized data	Performance not biased
Hawthorne effect	Avoided	Reliable	Not revealed hypotheses	Performance not biased
Process conformance	Observed participants	Reliable	Reminded participants to follow instructions	Ensure process conformance
Study-object coverage	Did not occur, all participants completed all tasks			
Ties to persistent memory	Ignored	Negligible influence		
Time pressure	Constant	Reliable	Enough time	No time pressure
Visual effort	Ignored	Negligible influence		
Technical				
Data consistency	Avoided	Reliable	Double checked data	Reliable
Instrumentation	Avoided	Reliable	Pilot study	Assure suitable instruments
Mono-method bias	Avoided	Reliable	Response time, correctness	Suitable indicators
Mono-operation bias	Avoided	Reliable	Different tasks	Reliable
Technical problems	Large sample/ avoided	Reliable	Pilot studies	Detect problems in time
Context related				
Learning effects	Constant	Reliable	Same order of tasks	Same learning effect
Mortality	Avoided	Reliable	One session	No drop outs
Operationalization of study object	Avoided	Reliable	Suitable based on literature	Learned from others
Ordering	Constant	Reliable	Same order of tasks	Same for all participants
Rosenthal	Avoided	Reliable	Standardized instructions	Reliable
Selection	Accepted	No according resources	Interpretation restricted to selected sample	
Study-object related				
Content of study object	Constant	Reliable	Same source code	Reliable
Language	Constant	Reliable	Languages participants knew	No training necessary
Layout of study object	Constant	Reliable	Same layout	Reliable
Size of study object	Constant	Reliable	Same size	Reliable
Tasks	Constant	Reliable	Same tasks	Reliable

Table 10.4: *Background colors: Experimental confounding parameters.*

10.3 Chapter 5: Measuring Programming Experience—Tasks

Figures 10.1 to 10.6 contain the algorithms we used in our experiment regarding programming experience (except for MobileMedia).

```
1  public class Class1 {
2    public static void main(String[] args) {
3      int array[] = {14,5,7};
4
5      for (int counter1 = 0; counter1 < array.length; counter1++) {
6        for (int counter2 = counter1; counter2 > 0; counter2--) {
7          if (array[counter2 - 1] > array[counter2]) {
8            int variable1 = array[counter2];
9            array[counter2] = array[counter2 - 1];
10           array[counter2 - 1] = variable1;
11         }
12       }
13     }
14
15     for (int counter3 = 0; counter3 < array.length; counter3++)
16       System.out.println(array[counter3]);
17   }
18 }
```

Figure 10.1: *Programming experience: Task 1.*

```
1  public class Class2 {
2
3    public static void main
4        (String args[]){
5      int [] variable1 = new int[128];
6      Class3 variable2 = new Class3(
           variable1, 0);
7
8      variable2.method1(7);
9      variable2.method1(2);
10     variable2.method1(8);
11
12     System.out.println(variable2.
           method2());
13     System.out.println(variable2.
           method2());
14     System.out.println(variable2.
           method2());
15   }
16 }
```

```
17 public class Class3 {
18
19   private int [] array;
20   private int index;
21
22   public Class3(int[] array,
23       int index) {
24     this.array = array;
25     this.index = index;
26   }
27
28     public void method1
29       (int variable3) {
30       array[index] = variable3;
31       index++;
32     }
33
34     public int method2 () {
35       index--;
36         return array[index];
37     }
38
39     public int methode3(int index) {
40       return array[index];
41     }
42 }
```

Figure 10.2: *Programming experience: Task 2.*

```
1  public class Stack {
2
3    private int [] elements;
4    private int index;
5
6    public Stack (int[] elements, int
         index) {
7      this.elements = elements;
8      this.index = index;
9    }
10
11     public void push (int item) {
12       elements[index] = item;
13       index++;
14     }
15
16     public int pop () {
17       index--;
18       return elements[index];
19     }
20
21     public int get (int index) {
22       return elements[index];
23     }
24 }
```

Figure 10.3: *Programming experience: Task 3.*

```
1  public class Test
2     public static void main(String[] args){
3       String ausgabe = null;
4       if (args[0].equals("Hello"))
5       {
6         ausgabe = args[0] + " World";
7       }
8       System.out.println(ausgabe);
9     }
10 }
```

Figure 10.4: *Programming experience: Task 9.*

10.4 Chapter 8: FMRI and Program Comprehension—Tasks

Figures 10.7 to 10.18 contain the source codes we use in our experiment in the fMRI scanner.

```
11 public class ListMain {
12
13   public static void main(String args[]) {
14     LinkedList list = new LinkedList();
15     AbstractElement a1 = new AbstractElement("A1");
16     AbstractElement a2 = new AbstractElement("A2");
17     AbstractElement a3 = new AbstractElement("A3");
18     AbstractElement a4 = new AbstractElement("A4");
19     AbstractElement a5 = new AbstractElement("A5");
20
21     list.insert(a5);
22     list.insert(a4);
23     list.insert(a3);
24     list.insert(a2);
25     list.insert(a1);
26
27     Iterator iter = list.iterator();
28     while (iter.hasNext()) {
29       System.out.println(iter.next().getElement());
30     }
31
32     list.insertAlgorithm(new AbstractElement("A6"));
33
34     iter = list.iterator();
35     while (iter.hasNext()) {
36       System.out.println(iter.next().getElement());
37     }
38
39 }
```

Figure 10.5: *Programming Experience: Tasks 4 to 8. The file* ListMain.java *changes slightly for each task.*

```
1  public class LinkedList {
2
3    Node head;
4
5    public LinkedList() {
6      this.head = new Node();
7    }
8
9    public void insert(AbstractElement e) {
10     Node newNode = new Node(e);
11     newNode.setNext(this.head.getNext());
12     head.setNext(newNode);
13   }
14
15   public void insertAlgorithm(AbstractElement ae){
16     Iterator iter = this.iterator();
17     Node previous = iter.next();
18     if (ae.compareTo(previous.getElement()) < 0) {
19       insert(ae);
20       return;
21     }
22     while (iter.hasNext()) {
23       Node tmp = iter.next();
24       if (ae.compareTo(tmp.getElement()) < 0) {
25         Node newNode = new Node(ae);
26         newNode.setNext(tmp);
27         previous.setNext(newNode);
28         break;
29       }
30       previous = tmp;
31     }
32      if (ae.compareTo(previous.getElement()) > 0) {
33       Node newNode = new Node(ae);
34       previous.setNext(newNode);
35        return;
36     }
37   }
38
39    public String removePos( int pos) {//...}
40    public int size() {//...}
41    public final void sortBubbleSort() {//... }
42    public Node getElementAt( int pos) {//...}
43
44    private void sort(AbstractElement[] array, int size) {
45      boolean swapped;
46      do {
47       swapped = false;
48        for ( int i = 0; i < size - 1; i++) {
49          if (array[i + 1].compareTo(array[i]) < 0) {
50           swap(array, i, i + 1);
51           swapped = true;
52         }
53       }
54     } while (swapped);
55   }
56
57    private void swap(AbstractElement[] array, int index1, int index2) {//...}
58
59 }
```

Figure 10.6: *Programming Experience: Tasks 4 to 8.*

```java
public static void main(String[] args) {
   int result = 1;
   int x = 4;

   while (x > 1) {
      result = result * x;
      x--;
   }
   System.out.println(result);
}
```

Figure 10.7: *FMRI: Task 1 (Compute faculty).*

```java
public static void main(String[] args) {
   int var1 = 23;
   int var2 = 42;
   int temp;
   temp = var1;
   var1 = var2;
   var2 = temp;
   System.out.println(var1);
}
```

Figure 10.8: *FMRI: Task 2 (Swap the value of two variables).*

```java
public static void main(String[] args) {
   String word = "Hello";
   String result = new String();

   for (int j = word.length() - 1; j >= 0; j--)
      result = result + word.charAt(j);

   System.out.println(result);
}
```

Figure 10.9: *FMRI: Task 3 (Reverse string).*

```java
public static void main(String[] args) {
   int[] array = {1, 6, 4};

   for (int i = 0; i <= array.length/2 - 1; i++) {
      int tmp = array[array.length - i - 1];
      array[array.length - i - 1] = array[i];
      array[i] = tmp;
   }

   for (int i = 0; i <= array.length - 1; i++)
      System.out.println(array[i]);
}
```

Figure 10.10: *FMRI: Task 4 (Reverse entries in array).*

```java
public static void main(String[] args) {
   String word = "Programming in Java";
   String key1 = "Java";
   String key2 = "Pascal";

   int index1 = word.indexOf(key1);
   int index2 = word.indexOf(key2);

   if (index1 != -1)
      System.out.println("Substring is contained: " + key1);
   else
      System.out.println("Substring is not contained: " + key1);

   if (index2 != -1)
      System.out.println("Substring is contained: " + key2);
   else
      System.out.println("Substring is not contained: " + key2);
}
```

Figure 10.11: *FMRI: Task 5 (Test whether substring is contained).*

```java
public static void main(String[] args){
   int i = 4;
   String result = "";

   while (i > 0) {
      if (i % 2 == 0)
         result = "0" + result;
      else
         result = "1" + result;
      i = i/2;
   }

   System.out.println(result);
}
```

Figure 10.12: *FMRI: Task 6 (Convert from decimal to binary).*

```java
public static void main (String[] args){
   int array[] = {2, 19, 5, 17};
   int result = array[0];
   for (int i = 1; i < array.length; i++)
      if (array[i] > result)
         result = array[i];
   System.out.println(result);
}
```

Figure 10.13: *FMRI: Task 7 (Find largest number in array).*

```java
public static void main(String[] args) {
   int number = 323;
   int result = 0;

   while (number != 0) {
      result = result + number % 10;
      number = number/10;
   }
   System.out.println(result);
}
```

Figure 10.14: *FMRI: Task 8 (Compute cross sum).*

```java
public static void main(String[] args) {
   int number = 11;
   boolean result = true;
   for (int i = 2; i < number; i++) {
      if (number % i == 0) {
         result = false;
         break;
      }
   }
   System.out.println(result);
}
```

Figure 10.15: *FMRI: Task 9 (Check for prime number).*

```
1  public static void main(String[] args) {
2     int[] array= {1, 2, 4, 5, 6, 10};
3     sort(array);
4
5     float b;
6     if (array.length % 2 == 1)
7        b = array[array.length/2];
8     else
9        b = (array[array.length/2 - 1] +
             array[array.length/2])/2;
10
11    System.out.println(b);
12 }
```

Figure 10.16: *FMRI: Task 10 (Find medium number).*

```
1  public static void main(String[] args) {
2     int num1 = 2;
3     int num2 = 3;
4     int result = num1;
5     for (int i = 1; i < num2; i++) {
6        result = result * num1;
7     }
8     System.out.println(result);
9  }
```

Figure 10.17: *FMRI: Task 11 (Compute power).*

```
1  public static void main(String[] args) {
2     int[] array= {1, 2, 4, 5, 6, 10};
3     sort(array);
4
5     float b;
6     if (array.length % 2 == 1)
7        b = array[array.length/2];
8     else
9        b = (array[array.length/2 - 1] + array[array.length/2])/2;
10
11    System.out.println(b);
12 }
```

Figure 10.18: *FMRI: Task 12 (Find median).*

Bibliography

Bram Adams, Bart Van Rompaey, Celina Gibbs, and Yvonne Coady. Aspect Mining in the Presence of the C Preprocessor. In *Proc. AOSD Workshop on Linking Aspect Technology and Evolution*, pages 1–6. ACM Press, 2008.

Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed Hassan. Can We Refactor Conditional Compilation into Aspects? In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 243–254. ACM Press, 2009.

Theodore Anderson and Jeremy Finn. *The New Statistical Analysis of Data*. Springer, 1996.

Theodore Anderson and Herman Rubin. Statistical Inference in Factor Analysis. In *Proc. Berkeley Symposium on Mathematical Statistics and Probability, Volume 5*, pages 111–150. University of California Press, 1956.

Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(4):1–36, 2009.

Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. Int'l Symposium Software Composition (SC)*, pages 20–35. Springer, 2008.

Sven Apel, Christian Kästner, and Salvador Trujillo. On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns. In *Proc. Int'l Workshop Assessment of Contemporary Modularization Techniques (ACoM)*, pages 1–7. IEEE CS, 2007.

Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008a.

Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An Algebra for Features and Feature Composition. In *Proc. Int'l Conf. Algebraic Methodology and Software Technology (AMAST)*, pages 36–50. Springer, 2008b.

Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-Independent, Automatic Software Composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231. IEEE CS, 2009.

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.

Erik Arisholm, Dag Sjøberg, Gunnar Carelius, and Yngve Lindsjørn. SESE: An Experiment Support Environment for Evaluating Software Engineering Technologies. In *Proc. Nordic Workshop Programming and Software Development Tools and Techniques*, pages 81–98, 2002.

Erik Arisholm, Hans Gallis, Tore Dybå, and Dag Sjøberg. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Trans. Softw. Eng.*, 33(2):65–86, 2007.

American Psychological Association. *Publication Manual of the American Psychological Association*. American Psychological Association, sixth edition, 2009.

David Atkins, Thomas Ball, Todd Graves, and Audris Mockus. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Trans. Softw. Eng.*, 28(7):625–637, 2002.

Lerina Aversano, Massimiliano Di Penta, and Ira Baxter. Handling Preprocessor-Conditioned Declarations. In *Proc. IEEE Int'l Workshop on Source Code Analysis and Manipulation*, pages 83–92. IEEE CS, 2002.

John Backus, Robert Beeber, Sheldon Best, Richard Goldberg, Lois Haibt, Harlan Herrick, Robert Nelson, David Sayre, Peter Sheridan, H. Stern, Irving Ziller, R. Hughes, and Roy Nutt. The FORTRAN Automatic Coding System. In *Western Joint Computer Conf. Techniques for Reliability (IRE-AIEE-ACM)*, pages 188–198. ACM Press, 1957.

Alan Baddeley. Is Working Memory Still Working? *The American Psychologist*, 56(11): 851–864, 2001.

Elisa Baniassad, Gail Murphy, and Christa Schwanninger. Design Pattern Rationale Graphs: Linking Design to Source. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 352–362. IEEE CS, 2003.

Victor Basili. Software Modeling and Measurement: The Goal/Question/Metric Paradigm. Technical Report CS-TR-2956 (UMIACS-TR-92-96), University of Maryland at College Park, 1992.

Don Batory, Jacob Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

Ira Baxter and Michael Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290. IEEE CS, 2001.

John Belliveau, David Kennedy, Robert McKinstry, Bradley Buchbinder, Robert Weisskoff, Mark Cohen, Michael Vevea, Thomas Brady, and Bruce Rosen. Functional Mapping of the Human Visual Cortex by Magnetic Resonance Imaging. *Science*, 254(5032): 716–719, 1991.

Peter Bentler and Chih-Ping Chou. Practical Issues in Structural Modeling. *Sociological Methods Research*, 16(1):78–117, 1987.

Ivo Bertoncello, Marcelo Dias, Patrick Brito, and Cecília Rubira. Explicit Exception Handling Variability in Component-based Product Line Architectures. In *Proc. Int'l Workshop Exception Handling (WEH)*, pages 47–54. ACM Press, 2008.

Nicolas Bettenburg and Ahmed Hassan. Studying the Impact of Social Structures on Software Quality. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 124–133. IEEE CS, 2010.

Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability Management with Feature Models. *Science of Computer Programming*, 53(3):333–352, 2004.

Stefan Biffl and Wilfried Grossmann. Evaluating the Accuracy of Defect Estimation Models Based on Inspection Data from Two Inspection Cycles. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 145–154. IEEE CS, 2001.

Stefan Biffl and Michael Halling. Investigating the Defect Detection Effectiveness and Cost Benefit of Nominal Inspection Teams. *IEEE Trans. Softw. Eng.*, 29(5):385–397, 2003.

Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. Impact of Limited Memory Resources. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 83–92. IEEE Computer Society, 2008.

David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To CamelCase or Under_score. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 158–167. IEEE CS, 2009.

Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

Richard Bornat, Saeed Dehnadi, and Simon. Mental Models, Consistency and Programming Aptitude. In *Proc. Conf. on Australasian Computing Education: Volume 78*, pages 53–61. Australian Computer Society, Inc., 2008.

Jürgen Bortz. *Statistik: für Human- und Sozialwissenschaftler*. Springer, sixth edition, 2004.

Lionel C. Briand, Yvan Labiche, Massimiliano Di Penta, and Han (Daphne) Yan-Bondoc. An Experimental Investigation of Formality in UML-Based Development. *IEEE Trans. Softw. Eng.*, 31(10):833–849, 2005.

Percy Bridgman. *The Logic of Modern Physics*. Macmillan, 1927.

Ruven Brooks. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 196–201. IEEE CS, 1978.

Ruven Brooks. Towards a Theory of the Comprehension of Computer Programs. *Int'l Journal of Man-Machine Studies*, 18(6):543–554, 1983.

Avi Bryant, Claudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Thais Batista, and Carlos Lucena. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 109–121. ACM Press, 2006.

David Budgen, Andy Burn, Pearl Brereton, Barbara Kitchenham, and Rialette Pretorius. Empirical Evidence about the UML: A Systematic Literature Review. *Software–Practice & Experience*, 41(4):363–392, 2011.

Christian Bunse. Using Patterns for the Refinement and Translation of UML Models: A Controlled Experiment. *Empirical Softw. Eng.*, 11(2):227–267, 2006.

Roberto Cabeza and Lars Nyberg. Imaging Cognition II: An Empirical Review of 275 PET and fMRI Studies. *Journal of Cognitive Neuroscience*, 12(1):1–47, 2000.

William Carlson, Larry Druffel, David Fisher, and William Whitaker. Introducing Ada. In *Proc. ACM Annual Conference (ACM)*, pages 263–271. ACM Press, 1980.

Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. The Effectiveness of Source Code Obfuscation: An Experimental Assessment. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 178–187. IEEE CS, 2009.

Fanny Chevalier, Pierre Dragicevic, Anastasia Bezerianos, and Jean-Daniel Fekete. Using Text Animated Transitions to Support Navigation in Document Histories. In *Proc. Conf. Human Factors in Computing Systems (CHI)*, pages 683–692. ACM Press, 2010.

Mark Chu-Carroll, James Wright, and Annie Ying. Visual Separation of Concerns through Multidimensional Program Storage. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 188–197. ACM Press, 2003.

Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison Wesley, second edition, 2006.

Paul Clements and Linda Northrop. *Software Product Lines: Practice and Patterns*. Addison Wesley, 2001.

Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge Academic Press, second edition, 1988.

Jacob Cohen and Patricia Cohen. *Applied Multiple Regression: Correlation Analysis for the Behavioral Sciences*. Addison Wesley, second edition, 1983.

Samuel Conte, Herbert Dunsmore, and Vincent Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., 1986. ISBN 0-8053-2162-4.

David Coppit, Robert Painter, and Meghan Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 754–757. IEEE CS, 2007.

Anna Costello and Jason Osborne. Best Practices in Exploratory Factor Analysis: Four Recommendations for Getting the Most from your Analysis. *Practical Assessment, Research & Evaluation*, 10(7):173–178, 2005.

Marcus Couto, Marco Valente, and Eduardo Figueiredo. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 191–200. IEEE CS, 2011.

Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 422–437. Springer, 2005.

John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 20–29. IEEE CS, 1995.

Brian de Alwis, Gail Murphy, and Martin Robillard. A Comparative Study of Three Program Exploration Tools. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 103–112. IEEE CS, 2007.

Andrea De Lucia, Carmine Gravino, Rocco Oliveto, and Genoveffa Tortora. An Experimental Comparison of ER and UML Class Diagrams for Data Modelling. *Empirical Softw. Eng.*, 15(5):455–492, 2010.

Uri Dekel and James Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 320–330. IEEE CS, 2009.

David Dewitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma Database Machine Project. *Data & Knowledge Engineering*, 2(1):44–62, 1990.

Arilo Claudio Dias-Neto and Guilherme Horta Travassos. Evaluation of {Model-based} Testing Techniques Selection Approaches: An External Replication. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 269–278. IEEE CS, 2009.

Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Softw. Eng.*, 10(4):405–435, 2005.

Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments. *IEEE Trans. Softw. Eng.*, 36(5):593–617, 2010.

Alastair Dunsmore and Marc Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS 35-2000, Department of Computer Science, University of Strathclyde, 2000.

Alastair Dunsmore, Marc Roper, and Murray Wood. Further Investigations into the Development and Evaluation of Reading Techniques for Object-oriented Code Inspection. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 47–57. ACM Press, 2002.

Tore Dybå, Vigdis By Kampenes, and Dag Sjøberg. A Systematic Review of Statistical Power in software Engineering Experiments. *Journal of Information and Software Technology*, 48(8):745–755, 2006.

Robert Dyer, Mehdi Bagherzadeh, Hridesh Rajan, and Yuanfang Cai. A Preliminary Study of Quantified, Typed Events. In *AOSD Workshop Empirical Evaluation of Software Composition Techniques (ESCOT)*, pages 34–35, 2010. http://www.cs.iastate.edu/ design/papers/ESCOT-10/.

Wojciech Dzidek, Erik Arisholm, and Lionel Briand. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *IEEE Trans. Softw. Eng.*, 34(3): 407–432, 2008.

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.

Stephen Eick, Joseph Steffen, and Eric Sumner. SeeSoft: A Tool for Visualizing Line Oriented Software Statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

Mohamed El-Attar and James Miller. A Subject-Based Empirical Evaluation of SSUCD's Performance in Reducing Inconsistencies in Use Case Models. *Empirical Softw. Eng.*, 14 (5):477–512, 2009.

Brian Ellis, Jeffrey Stylos, and Brad Myers. The Factory Pattern in API Design: A Usability Evaluation. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 302–312. IEEE CS, 2007.

Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the Effectiveness of the Test-First Approach to Programming. *IEEE Trans. Softw. Eng.*, 31(3):226–237, 2005.

Michael Ernst, Greg Badros, and David Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, 2002. ISSN 0098-5589.

Jean-Marie Favre. The CPP paradox. In *Proc. European Workshop on Software Maintenance*, 1995.

Jean-Marie Favre. Understanding-In-The-Large. In *Proc. Int'l Workshop on Program Comprehension*, page 29. IEEE CS, 1997.

Janet Feigenspan. Empirical Comparison of FOSD Approaches Regarding Program Comprehension – A Feasibility Study. Master's thesis, University of Magdeburg, 2009.

Janet Feigenspan and Norbert Siegmund. Supporting Comprehension Experiments with Human Subjects. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 244–246. IEEE CS, 2012. Tool demo.

Janet Feigenspan, Christian Kästner, Sven Apel, and Thomas Leich. How to Compare Program Comprehension in FOSD Empirically - An Experience Report. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 55–62. ACM Press, 2009.

Janet Feigenspan, Christian Kästner, Mathias Frisch, Raimund Dachselt, and Sven Apel. Visual Support for Understanding Product Lines. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 34–35. IEEE CS, 2010. Tool demo.

Janet Feigenspan, Sven Apel, Jörg Liebig, and Christian Kästner. Exploring Software Measures to Assess Program Comprehension. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE CS, 2011a. paper 3.

Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Proc. Int'l Conf. Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75. Institution of Engineering and Technology, 2011b.

Janet Feigenspan, Norbert Siegmund, and Jana Fruth. On the Role of Program Comprehension in Embedded Systems. In *Proc. Workshop Software Reengineering (WSR)*, pages 34–35, 2011c. `http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/FeSiFr11.pdf`.

Janet Feigenspan, Norbert Siegmund, Andreas Hasselberg, and Markus Köppen. PROPHET: Tool Infrastructure To Support Program Comprehension Experiments. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, 2011d. Poster.

Janet Feigenspan, Don Batory, and Taylor Riché. Is the Derivation of a Model Easier to Understand than the Model Itself? In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 47–52. IEEE CS, 2012a.

Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Softw. Eng.*, 2012b. DOI: 10.1007/s10664-012-9208-x.

Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring Programming Experience. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 73–82. IEEE CS, 2012c.

Norman Fenton, Shari Pfleeger, and Robert Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, 11(4):86–95, 1994.

Eduardo Figueiredo, Alessandro Garcia, Uira Kulesza, and Carlos Lucena. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. In *ECOOP Workshop Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, pages 58–69, 2005. `http://www.iro.umontreal.ca/~sahraouh/qaoose2005`.

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 *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 261–270. ACM Press, 2008a.

Eduardo Figueiredo, Claudio Sant'Anna, Alessandro Garcia, Thiago Bartolomei, Walter Cazzola, and Alessandro Marchetto. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 183–192. IEEE CS, 2008b.

Eduardo Figueiredo, John Whittle, and Alessandro Garcia. ConcernMorph: Metrics-based Detection of Crosscutting Patterns. In *Proc. Europ. Software Engineering Conf./ Foundations of Software Engineering (ESEC/FSE)*, pages 299–300. ACM Press, 2009.

Donald Fisher and Kay Tan. Visual Displays: The Highlighting Paradox. *Human Factors*, 31(1):17–30, 1989.

Ronald Fisher. On the Interpretation of χ^2 from Contingency Cables, and the Calculation of P. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922.

Ismênia Galvâo, Pim van den Broek, and Mehmet Akşit. A Model for Variability Design Rationale in SPL. In *Proc. Europ. Conf. Software Architecture (ECSA)*, pages 332–335. ACM Press, 2010.

Alessandro Garcia, Claudio Sant'Anna, Eduardo Figueiredo, Uira Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 3–14. ACM Press, 2005.

Alejandra Garrido and Ralph Johnson. Analyzing Multiple Configurations of a C Program. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 379–388. IEEE CS, 2005.

Marcela Genero, Esperanza Manso, Aaron Visaggio, Gerardo Canfora, and Mario Piattini. Building Measure-based Prediction Models for UML Class Diagram Maintainability. *Empirical Softw. Eng.*, 12(5):517–549, 2007.

Bruce Goldstein. *Sensation and Perception*. Cengage Learning Services, fifth edition, 2002.

James Goodwin. *Research in Psychology: Methods and Design*. Wiley Publishing, Inc., second edition, 1999.

Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Claudio Sant'Anna, Sergio Soares, Paulo Borba, Uira Kulesza, and Awais Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 176–200. Springer, 2007.

Gürcan Güleşir, Klaas Berg, Lodewijk Bergmans, and Mehmet Akşit. Experimental evaluation of a tool for the verification and transformation of source code in event-driven systems. *Empirical Softw. Eng.*, 14(6):720–777, 2009.

Atul Gupta and Pankaj Jalote. An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 285–294. IEEE CS, 2007.

Maurice Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

Stefan Hanenberg. An Experiment about Static and Dynamic Type Sytems: Doubts about the Positive Impact of Static Type Systems on Development Time. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 22–35. ACM Press, 2010.

Stefan Hanenberg, Sebastian Kleinschmager, and Manuel Josupeit-Walter. Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 156–167. IEEE CS, 2009.

Jo Hannay, Erik Arisholm, Harald Engvik, and Dag Sjøberg. Effects of Personality on Pair Programming. *IEEE Trans. Softw. Eng.*, 36(1):61–80, 2010.

Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 943–944. ACM Press, 2008. Tool demo.

Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.

Sallie Henry, Matthew Humphrey, and John Lewis. Evaluation of the Maintainability of Object-Oriented Software. In *IEEE Region 10 Conf. Computer and Comm. Systems*, pages 404–409. IEEE CS, 1990.

Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Toolchain-independent Variant Management with the Leviathan Filesystem. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 18–24. ACM Press, 2010.

Martin Höst, Björn Regnell, and Claes Wohlin. Using Students as Subjects: A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Softw. Eng.*, 5(3):201–214, 2000.

Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Laguë. C/C++ Conditional Compilation Analysis using Symbolic Execution. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 196–206. IEEE CS, 2000.

Shinobu Ishihara. *Test for Colour-Blindness*. Kanehara Shuppan Co., 1972.

Juha Itkonen, Mika Mantyla, and Casper Lassenius. Defect Detection Efficiency: Test Case Based vs. Exploratory Testing. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 61–70. IEEE CS, 2007.

Patricia Jablonski and Daqing Hou. Aiding Software Maintenance with Copy-and-Paste Clone-Awareness. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 170–179. IEEE CS, 2010.

Adolf Jäger, Heinz-Martin Süß, and Andre Beauducel. *Berliner Intelligenzstruktur-Test*. Hogrefe, 1997.

Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 810–811. IEEE CS, 2003.

Sebastien Jeanmart, Yann-Gael Gueheneuc, Houari Sahraoui, and Naji Habra. Impact of the Visitor Pattern on Program Comprehension and Maintenance. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 69–78. IEEE CS, 2009.

Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.

Eric Jensen. *Teaching with the Brain in Mind*. Atlantic Books, 1998.

Ralph Johnson and Brian Foote. Designing Reuseable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

Natalia Juristo and Ana Moreno. *Basics of Software Engineering Experimentation*. Kluwer, 2001.

Dennis Kafura. A Survey of Software Metrics. In *Proc. ACM Annual Conference (ACM) Range of Computing: Mid-80's Perspective*, pages 502–506. ACM Press, 1985.

Vigdis Kampenes, Tore Dybå, Jo Hannay, and Dag Sjøberg. A Systematic Review of Quasi-Experiments in Software Engineering. *Information and Software Technology*, 51(1): 71–82, 2009.

Christian Kästner. *Virtual Separation of Concerns: Preprocessors 2.0*. PhD thesis, University of Magdeburg, 2010.

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

Christian Kästner, Sven Apel, and Martin Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009a.

Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 611–614. IEEE CS, 2009b. Tool demo.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopez, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.

Barbara Kitchenham and Stuart Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

Barbara Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic Literature Reviews in Software Engineering: A Systematic Literature Review. *Journal of Information and Software Technology*, 51(1):7–15, 2009.

Sebastian Kleinschmager and Stefan Hanenberg. How to Rate Programming Skills in Programming Experiments? A Preliminary, Exploratory, Study Based on University Marks, Pretests, and Self-Estimation. In *Proc. ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 15–24. ACM Press, 2011.

Jens Knodel, Dirk Muthig, and Matthias Naab. An Experiment on the Role of Graphical Elements in Architecture Visualization. *Empirical Softw. Eng.*, 13(6):693–726, 2008.

Andrew Ko and Bob Uttl. Individual Differences in Program Comprehension Strategies in Unfamiliar Programming Systems. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 175–184. IEEE CS, 2003.

Andrew Ko, Brad Myers, Michael Coblenz, and Htet Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, 2006.

Jürgen Koenemann and Scott Robertson. Expert Problem Solving Strategies for Program Comprehension. In *Proc. Conf. Human Factors in Computing Systems (CHI)*, pages 125–130. ACM Press, 1991.

Maren Krone and Gregor Snelting. On the Inference of Configuration Structures from Source Code. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 49–57. IEEE CS, 1994.

Charles Krueger. Easing the Transition to Software Mass Customization. In *Proc. Int'l Workshop on Software Product-Family Eng.*, pages 282–293. Springer, 2002.

Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 223–233. IEEE CS, 2006.

Ludwik Kuzniarz, Miroslaw Starin, and Claes Wohlin. An Empirical Study on Using Stereotypes to Improve Understanding of UML Models. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 14–23. IEEE CS, 2004.

Steve Lang and Anneliese von Mayrhauser. Building a Research Infrastructure for Program Comprehension Observations. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 165–169. IEEE CS, 1997.

Christian Lange and Michel Chaudron. Effects of Defects in UML Models: An Experimental Investigation. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 401–411. ACM Press, 2006.

Thomas LaToza and Brad Myers. Developers Ask Reachability Questions. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 185–194. ACM Press, 2010.

Dawn Lawrie, Henry Feild, and David Binkley. Leveraged Quality Assessment Using Information Retrieval Techniques. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 149–158. IEEE CS, 2006.

Krystle Lemon, Edward Allen, Jeffrey Carver, and Gary Bradshaw. An Empirical Study of the Effects of Gestalt Principles on Diagram Understandability. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 156–165. IEEE CS, 2007.

Michael Lewis-Beck. *Applied Regression: An Introduction*. Sage Pubications, 1980.

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. ACM Press, 2010.

Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM Press, 2011.

Rensis Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 22 (140):1–55, 1932.

Panos Livadas and David Small. Understanding Code Containing Preprocessor Constructs. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 89–97. IEEE CS, 1994.

Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall, 1994.

Kim Lui, Keith Chan, and John Nosek. The Effect of Pairs in Program Design Tasks. *IEEE Trans. Softw. Eng.*, 34(2):197–211, 2008.

Jutta Markgraf, Hans-Peter Musahl, Friedrich Wilkening, Karin Wilkening, and Viktor Sarris. Studieneinheit Versuchsplanung, 2001. FIM-Psychologie Modellversuch, Universität Erlangen-Nürnberg.

Nick Matthijssen, Andy Zaidman, Margaret-Anne Storey, Ian Bull, and Arie van Deursen. Connecting Traces: Understanding Client-Server Interactions in Ajax Applications. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 216–225. IEEE CS, 2010.

Thomas McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.*, pages 308–320, 1976.

Bill McCloskey and Eric Brewer. ASTEC: A New Approach to Refactoring C. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 21–30. ACM Press, 2005.

Steve McConnell. What does 10x Mean? Measuring Variations in Programmer Productivity. In *Making Software*, pages 567–574. O'Reilly & Associates, Inc., 2011.

Austin Melton, David Gustafson, James Bieman, and Albert Baker. A Mathematical Perspective For Software Measures Research. *Software Engineering Journal*, 5(5):246–254, 1990.

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

George Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, 63(2):81–97, 1956.

Andrew Mohan and Nicolas Gold. Programming Style Changes in Evolving Source Code. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 236–240. IEEE CS, 2004.

Ambra Molesini, Alessandro Garcia, Christina Chavez, and Thais Batista. On the Quantitative Analysis of Architecture Stability in Aspectual Decompositions. In *Proc. Working IEEE/IFIP Conf. on Software Architecture (WICSA)*, pages 29–38. IEEE CS, 2008.

Douglas Mook. *Motivation: The Organization of Action*. W.W. Norton & Co., second edition, 1996.

Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming Dynamically Adaptive Systems Using Models and Aspects. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 122–132. IEEE CS, 2009.

Jarrod Moss, Christian Schunn, Walter Schneider, Danielle McNamara, and Kurt VanLehn. The Neural Correlates of Strategic Reading Comprehension: Cognitive Control and Discourse Comprehension. *NeuroImage*, 58(2):675–686, 2011.

Samar Mouchawrab, Lionel Briand, and Yvan Labiche. Assessing, comparing, and combining statechart- based testing and structural testing: An experiment. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 41–50. IEEE CS, 2007.

Matthias Müller. Are Reviews an Alternative to Pair Programming? *Empirical Softw. Eng.*, 9(4):335–351, 2004.

Isabel Myers and Mary McCaulley. *Manual: A Guide to the Development and Use of the Myers-Briggs Type Indicator*. Consulting Psychologists Press, 1985.

Nadim Nachar. The Mann-Whitney U: A Test for Assessing whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology*, 4(1):13–20, 2008.

John von Neumann. *First Draft of a Report on the EDVAC*, 1945.

Tim Ng, Shing-chi Cheung, Wing-Kwong Chan, and Yuen Yu. Work Experience versus Refactoring to Design Patterns: A Controlled Experiment. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 12–22. ACM Press, 2006.

Bruce Oberg and David Notkin. Error Reporting with Graduated Color. *IEEE Software*, 9 (6):33–38, 1992.

Michael O'Brien and Jim Buckley. Inference-Based and Expectation-Based Processing in Program Comprehension. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 71–78. IEEE CS, 2001.

Johan och Dag and Thomas Thelin und Björn Regnel. An Experiment on Linguistic Tool Support for Consolidation of Requirements from Multiple Sources in Market-driven Product Development. *Empirical Softw. Eng.*, 11(2):303–329, 2006.

Mari Otero and José Dolado. An Initial Experimental Assessment of the Dynamic Modelling in UML. *Empirical Softw. Eng.*, 7(1):27–47, 2002.

Jeffrey Overbey and Ralph Johnson. Generating Rewritable Abstract Syntax Trees. In *Software Language Engineering*, pages 114–133. Springer, 2009.

David Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

Chris Parnin and Spencer Rugaber. Programmer Information Needs after Memory Failure. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 123–132. IEEE CS, 2012.

Troy Pearse and Paul Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 270–277. IEEE CS, 1997.

Nancy Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

Robert Peterson. *Constructing Effective Questionnaires*. Sage Publications, 2000.

Gerardo Porras and Yann-Gaël Guéhéneuc. An Empirical Study on the Efficiency of Different Design Pattern Representations in UML Class Diagrams. *Empirical Softw. Eng.*, 15(5):493–522, 2010.

Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, 2002.

Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.

Hazem Qattous, Philip Gray, and Ray Welland. An Empirical Study of Specification by Example in a Software Engineering Tool. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 16:1–16:10. ACM Press, 2010.

Jochen Quante. Do Dynamic Object Process Graphs Support Program Understanding? A Controlled Experiment. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 73 –82. IEEE CS, 2008.

Václav Rajlich and Norman Wilde. The Role of Concepts in Program Comprehension. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 271–278. IEEE CS, 2002.

Gerard Rambally. The Influence of Color on Program Readability and Comprehensibility. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*, pages 173–181. ACM Press, 1986.

John 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.

Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent Feature Modularization. In *Proc. Int'l Conf. Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, pages 11–18. ACM Press, 2010.

Filippo Ricca, Massimiliano Di Penta, Marco Torchiano, Paolo Tonella, and Mariano Ceccato. The Role of Experience and Ability in Comprehension Tasks Supported by UML Stereotypes. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 375–384. IEEE CS, 2007.

John Rice. Display Color Coding: 10 Rules of Thumb. *IEEE Software*, 8(1):86–88, 1991.

Charles Rich. Inspection Methods in Programming. Master's thesis, Massachusetts Institute of Technology, 1987.

Taylor Riché, Don Batory, Rui Goncalves, and Bryan Marker. Architecture Design by Transformation. Technical Report TR-10-39, University of Texas at Austin, Department for Computer Science, 2010.

Fabrizio Riguzzi. A Survey of Software Metrics. Technical Report DEIS-LIA-96-010, Universita di Bologna, 1996.

Martin Robillard and Gail Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 406–416. ACM Press, 2002.

Martin Robillard and Gail Murphy. FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 822–823. IEEE CS, 2003.

Martin Robillard and Gail Murphy. Representing Concerns in Source Code. *ACM Trans. Softw. Eng. & Methodology*, 16(1):1–38, 2007.

Martin Robillard, Wesley Coelho, and Gail Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004.

Fritz Roethlisberger. *Management and the Worker*. Harvard University Press, 1939.

Robert Rosenthal and Lenore Jacobson. Teachers' Expectancies: Determinants of Pupils' IQ Gains. *Psychological Reports*, 19(1):115–118, 1966.

Hal Sackman, Warren Erikson, and Eugene Grant. Exploratory Experimental Studies Comparing Online and Offline Programming Performance. *Commun. ACM*, 11(1):3–11, 1968.

Jorma Sajaniemi and Raquel Prieto. An Investigation into Professional Programmers' Mental Representations of Variables. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 55–64. IEEE CS, 2005.

David Salomon. *Assemblers and Loaders*. Ellis Horwood, 1992.

Anita Sarma, David Redmiles, and André van der Hoek. Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 113–123. ACM Press, 2008.

Panagiotis Sfetsos, Ioannis Stamelos, Lefteris Angelis, and Ignatios Deligiannis. An Experimental Investigation of Personality Types Impact on Pair Effectiveness in Pair Programming. *Empirical Softw. Eng.*, 14(9):187–226, 2009.

William Shadish, Thomas Cook, and Donald Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin Company, 2002.

Teresa Shaft and Iris Vessey. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6(3):286–299, 1995.

Samuel Shapiro and Martin Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591–611, 1965.

Bonita Sharif and Johnathon Maletic. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 196–205. IEEE CS, 2010.

Bonita Sharif and Jonathon Maletic. An Empirical Study on the Comprehension of Stereotyped UML Class Diagram Layouts. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 268–272. IEEE CS, 2009.

Ben Shneiderman and Richard Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *Int'l Journal of Parallel Programming*, 8 (3):219–238, 1979.

Janet Siegmund, André Brechmann, Sven Apel, Christian Kästner, Jörg Liebig, Thomas Leich, and Gunter Saake. Toward Measuring Program Comprehension with Functional Magnetic Resonance Imaging. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM Press, 2012a. New ideas paper. Submitted 29.6., Accepted 14.8.

Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM Press, 2012b. Submitted 02.07, Accepted 07.08.

Janet Siegmund, Norbert Siegmund, Jana Fruth, Sven Kuhlmann, Jana Dittmann, and Gunter Saake. Program Comprehension in Preprocessor-Based Software. In *Proc. Int'l Workshop Digital Engineering (IWDE)*. Springer, 2012c. Submitted 03.06., Accepted 03.07.

Jonathan Sillito, Gail Murphy, and Kris De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, 2008.

Nieraj Singh, Graeme Johnson, and Yvonne Coady. CViMe: Viewing Conditionally Compiled C/C++ Sources through Java. In *Companion ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 730–731. ACM Press, 2006.

Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-CLR: A Tool for Navigating Highly Configurable System Software. In *Proc. Workshop Aspects, Components, and Patterns for Infrastr. Software*. ACM Press, 2007.

Dag Sjøberg, Jo Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette Rekdal. A Survey of Controlled Experiments in Software Engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753, 2005.

Alvy Smith. Color gamut transform pairs. In *Proc. Annual Conf. Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 12–19. ACM Press, 1978.

Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, 1984.

Maarten Someren, Yvonne Barnard, and Jacobijn Sandberg. *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic Press, 1994.

Henry Spencer and Geoff Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, pages 185–198. USENIX Association, 1992.

Thomas Standish. An Essay on Software Reuse. *IEEE Trans. Softw. Eng.*, SE–10(5):494–497, 1984.

Michael Stengel, Janet Feigenspan, Mathias Frisch, Christian Kästner, Sven Apel, and Raimund Dachselt. View Infinity: A Zoomable Interface for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1031–1033. ACM Press, 2011.

Mikael Svahnberg and Claes Wohlin. An Investigation of a Method for Identifying a Software Architecture Candidate with Respect to Quality Attributes. *Empirical Softw. Eng.*, 10(2):149–181, 2005.

Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. Using Students as Subjects: An Empirical Evaluation. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 288–290. ACM Press, 2008.

Franklin Tamborello and Michael Byrne. Adaptive but Non-optimal Visual Search with Highlighted Displays. *Cognitive Systems Research*, 8(3):182–191, 2007.

Peri Tarr and Harold Ossher. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 729–730. IEEE CS, 2001.

Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time Configurable System Software. In *Proc. EuroSys 2011 Conference (EuroSys '11)*, pages 47–60. ACM Press, 2011.

Thomas Thelin. Team-Based Fault Content Estimation in the Software Inspection Process. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 263–272. IEEE CS, 2004.

Thomas Thelin, Per Runeson, Claes Wohlin, Thomas Olsson, and Carina Andersson. Evaluation of Usage-based Reading: Conclusions after Three Experiments. *Empirical Softw. Eng.*, 9(1-2):77–110, 2004.

Rebecca Tiarks. What Programmers Really Do: An Observational Study. In *Proc. Workshop Software Reengineering (WSR)*, pages 36–37, 2011.

Walter Tichy. Hints for Reviewing Empirical Work in Software Engineering. *Empirical Softw. Eng.*, 5(4):309–312, 2000.

Marco Torchiano. Empirical Assessment of UML Static Object Diagrams. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 226–230. IEEE CS, 2004.

John Tukey. *Exploratory Data Analysis*. Addison Wesley, 1977.

László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus Schema for C/C++ Preprocessing. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 75–84. IEEE CS, 2004.

Padmal Vitharana and Keshavamurthy Ramamurthy. Computer-Mediated Group Support, Anonymity, and the Software Inspection Process: An Empirical Investigation. *IEEE Trans. Softw. Eng.*, 29(2):167–180, 2003.

Padmal Vitharana, Fatemeh Zahedi, and Hemant Jain. Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis. *IEEE Trans. Softw. Eng.*, 29(7):649–664, 2003.

Marek Vokáč, Walter Tichy, Dag I. K. Sjøberg, Erik Arisholm, and Magne Aldrin. A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns—A Replication in a Real Programming Environment. *Empirical Softw. Eng.*, 9(3):149–195, 2004.

Anneliese von Mayrhauser and Marie Vans. From Program Comprehension to Tool Requirements for an Industrial Environment. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 78–86. IEEE CS, 1993.

Anneliese von Mayrhauser and Marie Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

Anneliese von Mayrhauser, Marie Vans, and Adele Howe. Program Understanding Behaviour during Enhancement of Large-scale Software. *Journal of Software Maintenance: Research and Practice*, 9(5):299–327, 1997.

Andrew Walenstein. Observing and Measuring Cognitive Support: Steps Toward Systematic Tool Evaluation and Engineering. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 185–194. IEEE CS, 2003.

Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann, 2000.

David Wechsler. *The Measurement of Adult Intelligence*. American Psychological Association, third edition, 1950.

Urban Wiesing. *Die Ethik-Kommissionen – Neuere Entwicklungen und Richtlinien*. Deutscher Ärzte-Verlag, 2003.

Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

Robert Woodworth. *Experimental Psychology*. Henry Holt, 1939.

Wilhelm Wundt. *Grundzüge der Physiologischen Psychologie*. Engelmann, 1874.

Wilhelm Wundt. *Grundriß der Psychologie*. Kröner, 1914.

Shaohua Xie, Eileen Kraemer, and Richard Stirewalt. Empirical Evaluation of a UML Sequence Diagram with Adornments to Support Understanding of Thread Interactions. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 123–134. IEEE CS, 2007.

Wuu Yang. How to Merge Program Texts. *Journal of Systems and Software*, 27(2):129–135, 1994.

John Yellott. Correction for Fast Guessing and the Speed Accuracy Trade-off in Choice Reaction Time. *Journal of Mathematical Psychology*, 8(2):159–199, 1971.

Bo Zhang. Extraction and Improvement of Conditionally Compiled Product Line Code. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 257–258. IEEE CS, 2012.

Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting Defects for Eclipse. In *Proc. Int'l Workshop Predictor Models in Software Engineering (PMSE)*, pages 9–15. IEEE CS, 2007.

Horst Zuse. *Software Complexity: Measures and Methods*. Walter de Gruyter & Co., 1991.