Experience from Measuring Program Comprehension—Toward a General Framework

Janet Siegmund*$^{\sigma}$, Christian Kästner$^{\omega}$, Sven Apel$^{\pi}$, André Brechmann$^{\theta}$, Gunter Saake$^{\sigma}$
$^{\sigma}$University of Magdeburg, $^{\omega}$Carnegie Mellon University, Pittsburgh
$^{\pi}$University of Passau, $^{\theta}$Leibniz Institute for Neurobiology, Magdeburg

Abstract: Program comprehension plays a crucial role during the software-development life cycle: Maintenance programmers spend most of their time with comprehending source code, and maintenance is the main cost factor in software development. Thus, if we can improve program comprehension, we can save considerable amount of time and cost. To improve program comprehension, we have to measure it first. However, program comprehension is a complex, internal cognitive process that we cannot observe directly. Typically, we need to conduct controlled experiments to soundly measure program comprehension. However, empirical research is applied only reluctantly in software engineering. To close this gap, we set out to support researchers in planning and conducting experiments regarding program comprehension. We report our experience with experiments that we conducted and present the resulting framework to support researchers in planning and conducting experiments. Additionally, we discuss the role of teaching for the empirical researchers of tomorrow.

1 Introduction

Today, we are surrounded by computers. They are in our cars, credit cards, and cell phones. Thus, there is a lot of source code that needs to be implemented and maintained. In the software-development life cycle, maintenance is the main cost factor [Boe81]. Furthermore, maintenance developers spend most of their time with understanding source code [vMVH97, Sta84, Tia11]. Thus, if we support program comprehension, we can save considerable amount of time and cost of the software-development life cycle.

To improve program comprehension, various programming techniques and tools were improved since the first programmable computers. From machine code over assembly languages, procedural programming, and contemporary object-oriented programming [Mey97], modern programming paradigms, such as feature-oriented and aspect-oriented programming emerged [Pre97, KLM^{+}97]. In the same way, tools have been developed to support programmers in working with source code. For example, Eclipse, FeatureIDE [KTS^{+}09], or CIDE [KAK08] target, among others, better comprehension of software.

These new techniques and tools are only rarely evaluated empirically regarding their

*This author published previous work as Janet Feigenspan effect on program comprehension. However, program comprehension is an internal cognitive process and can be evaluated only in controlled experiments—plausibility arguments are not sufficient, because they can prove wrong in practice. One reason for the reluctance of using empirical evaluation is the effort for conducting experiments, which usually takes several months from the planning phase to results.

With our work, we want to raise awareness of the necessity of empirical research and initiate a discussion on how to motivate and support other researchers to conduct more empirical studies in software engineering. We focus on the following contributions:

- Description of typical problems when planning controlled experiments based on our experience.
- Framework to reduce the effort of conducting controlled experiments.
- Establish and revive the empirical-research community in software engineering.

Of course, the appeal for conducting more empirical research in software engineering is not new [Kit93, TLPH95, Tic98]. Nevertheless, in a recent study, Sjoberg and others found that the amount of papers reporting controlled experiments is still low (1.9 %). Thus, there is still a reluctance to apply empirical methods. By reporting our experience and the resulting framework, we hope to remove the obstacles of planning and conducting experiments and, thus, motivate other researchers to apply controlled experiments to evaluate their new techniques and tools regarding the effect on programmers.

This paper is structured as follows: First, we take a closer look at program comprehension and the logic of experiments, followed by our experience with program-comprehension experiments. This way, we want to raise the awareness for the difficulties that accompany controlled experiments. Then, we present the framework, which we developed based on our experience and which supports researchers with conducting experiments. All material we present here is also available at the project's website (`http://fosd.net/experiments`).

2 Program Comprehension and its Measurement

In this section, we recapitulate important concepts related to program comprehension to ensure the same level of familiarity for our readers. Readers familiar with program comprehension and controlled experiments may skip this section. Our intention is not to give an exhausting overview of program comprehension, but to give an impression of the complexity of the process. We are aware that we are focusing on one aspect of program comprehension (other aspects are described, e.g., by Fritz and others [FOMMH10]). In the same way, we only shortly discuss software measures.

To understand source code, developers typically use either top-down or bottom-up comprehension. When developers are familiar with a program's domain, they use top-down comprehension; so they start with stating a general hypothesis about a program's purpose [Bro78, SV95, SE84]. By looking at details, developers refine this hypothesis. If they encounter a part of a program from an unfamiliar domain, they switch to bottom-up comprehension. In this case, developers analyze source code statement by statement and group statements to semantic chunks, until they understand the code fragments [Pen87, SM79].

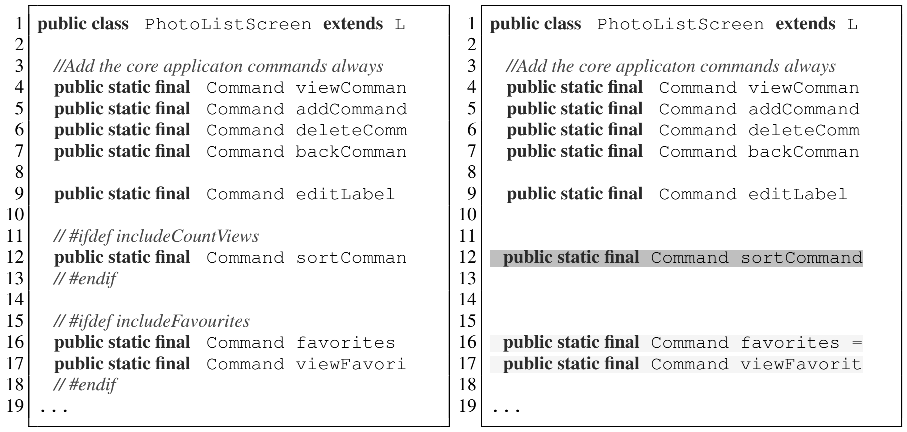
Thus, program comprehension is a complex process. To measure it, we need to find an indicator. Often, software measures are used, such as lines of code or cyclomatic complexity [HS95]. It seems reasonable that the more lines of code or the more branching statements a program has, the more difficult it is to understand. However, software measures cannot fully capture the complex process of understanding source code [Boy77, FALK11]. Thus, we should not rely solely on software measures to measure program comprehension.

Another way is to observe in controlled experiments how human participants understand source code. This way, we do not rely on properties of source code, but consider the comprehension process itself. Often, tasks, such as corrective or enhancing maintenance tasks are used [DR00, FSF11]. The idea is that if participants succeed in solving a task, they must have understood the source code—otherwise, they would not be able to provide a correct solution. Another way is to use think-aloud protocols, in which participants verbalize their thoughts [ABPC91]. This allows us to observe the process of comprehension. Both techniques, tasks and think-aloud protocols only indirectly assess program comprehension. So far, there is no way to look into participants' brain while they are understanding source code.

Designing controlled experiments with human participants requires considerable effort and experience, because there are *confounding* parameters that need to be considered. Confounding parameters *influence* the behavior of participants and can bias the result. For example, an expert programmer understands source code different than a novice programmer; a participant who is familiar with a program's domain uses top-down comprehension, whereas a participant who is unfamiliar with a domain uses bottom-up comprehension. Even details that appear irrelevant may influence behavior, such as violating coding conventions (the performance of expert programmers can decrease to the level of novice programmers [SE84]) or using under_score vs. camelCase identifier style [BDLM09, SM10].

After identifying relevant confounding parameters, we need to select suitable control techniques. For example, to avoid bias due to differences in programming experience, we could only recruit novice programmers. However, this would limit the applicability of our results to novice programmers. Instead of recruiting only novices, we could also measure programming experience and evaluate how it influences the result. However, measuring programming experience takes additional time and requires that a measurement instrument exists. Additionally, we need more participants and it increases the complexity of the experimental design and analysis methods. Thus, there is always a trade off between generalizability, accuracy of results, and available resources.

Next, we present our experiences with our controlled experiments and how we addressed this trade off.



```
 1 public class  PhotoListScreen  extends  L
 2
 3   //Add the core applicaton commands always
 4   public static final  Command viewComman
 5   public static final  Command addCommand
 6   public static final  Command deleteComm
 7   public static final  Command backComman
 8
 9   public static final  Command editLabel
10
11   // #ifdef includeCountViews
12   public static final  Command sortComman
13   // #endif
14
15   // #ifdef includeFavourites
16   public static final  Command favorites
17   public static final  Command viewFavori
18   // #endif
19 ...
```

```
 1 public class  PhotoListScreen  extends  L
 2
 3   //Add the core applicaton commands always
 4   public static final  Command viewComman
 5   public static final  Command addCommand
 6   public static final  Command deleteComm
 7   public static final  Command backComman
 8
 9   public static final  Command editLabel
10
11
12   public static final  Command sortCommand
13
14
15
16   public static final  Command favorites =
17   public static final  Command viewFavorit
18
19 ...
```

Figure 1: Left: Source code with #ifdef directives; right: source code with background colors. In the colored version, Line 12 is annotated with orange background color, Lines 16 and 17 with yellow.

3 Experience

When we started our work on program comprehension in 2009, we set out to evaluate how modern programming paradigms, such as feature-oriented programming [Pre97] or aspect-oriented programming [KLM$^+$97], and new tools, such as CIDE [KAK08], influence program comprehension. However, during the planning phase, we soon learned that such broad questions are unsolvable in a single experiment—we would need over one million participants to account for all confounding parameters [Fei09].

Thus, we narrowed our research question. In a first experiment, we evaluated how background colors support program comprehension compared to no background colors in preprocessor-based code [FKA$^+$12]. As material, we used one medium-sized Java program that was implemented with preprocessor directives.[1] From that program, we created a second version, in which we used background colors instead of preprocessor directives. Everything else was the same. In Figure 1, we show a screen shot of both version to give an impression.

By narrowing our research questions, we controlled for a lot of confounding parameters and can attribute program comprehension, operationalized by correctness and response time of solutions, only to the difference in the source code we used (i.e., background colors vs. textual #ifdef directives). As a drawback, however, our results are limited to the circumstances of our study: Java as the programming language, medium-sized program, students as participants, etc.

Designing this seemingly simple study took us several months and a master's thesis. The

[1]Java is a popular language to develop Apps for mobile devices. To meet the resource constraints, preprocessors, such as Antenna or Munge, are also used for Java.

```
public class PhotoListScreen{
  /* ... */
  // #ifdef includeFavourites
  public static final Command favoriteCommand = new
        Command("Set Favorite",Command.ITEM,1);
  public static final Command viewFavoritesCommand = new
        Command("View Favorites",Command.ITEM,1);
  // #endif
  /* ... */
  public void initMenu() {
    /* ... */
    // #ifdef includeFavourites
    this .addCommand(favoriteCommand);

    // #endif
    /* ... */
  }
}
```

Figure 2: Bug for M3: `viewFavoritesCommand` is not added.

most difficult problems were to find suitable material, present the material to participants, control for programming experience as one of the most important parameters, and find a suitable indicator for program comprehension. We discuss each problem in more detail.

First, we needed to find suitable material. Participants should not understand it at first sight nor be overwhelmed by the amount of source code. Furthermore, the source code should be implemented in Java, because participants are familiar with it. After several weeks of searching and comparing source code, we found MobileMedia, which has a suitable size, was code reviewed to ensure coding conventions, and was implemented in Java ME with preprocessors [FCM$^+$08]. Hence, finding suitable material is tedious and can take some time.

Second, we needed to present source code to participants. If we had used Eclipse, participants who were familiar with it (e.g., knew how to use call hierarchies or regular expressions to search for code fragments), would have performed better independent of whether they worked with background colors of #ifdef directives. Eventually, we used a browser setting with syntax highlighting and basic source-code navigation, but no other tool support. Thus, an intuitive solution may not be optimal and a seemingly awkward solution may be the better choice.

Third, we needed to control for programming experience, because participants with more experience typically understand source code better. Since we did not find any validated questionnaire or test to measure programming experience, we constructed our own based on a literature survey and by consulting programming experts. We again needed to investigate a couple of weeks, but as result we had a questionnaire that we can reuse in subsequent experiments. Hence, we should take great care to control for confounding parameters, especially if we believe they have an important influence on our result.

Last, to measure program comprehension, we used tasks, which participants could only solve if they understood the underlying source code first. In a pilot study, we observed that our tasks fulfilled that criterion, but some of them were too difficult. To give an impression of the nature of the tasks, we present the source code of one task in Figure 2. The bug description was that a command was not shown, although it is implemented. The bug was located in Line 12, where the command was not added to the menu. Thus, we adapted the tasks and evaluated their difficulty in a second pilot study. Eventually, after several months of careful planning, the design of our study was finished.

In subsequent experiments [FSP$^+$11, FALK11, FKL$^+$12, SFF$^+$11, FKA$^+$12, SBA$^+$12, SKLA12], we did profit from our experiences of the first experiment. For example, we often used different variants of MobileMedia and the programming-experience questionnaire. Furthermore, we developed a feeling for how difficult tasks can be when working with students of computer science, so that often one small pilot study sufficed. Additionally, we could reuse the scripts for analyzing the data. Thus, the effort and time invested in the first experiment payed off for subsequent experiments. With our work, we aim at reducing the effort of the first experiment for other researchers, for whom we developed the framework that we present next.

4 A Framework to Support Controlled Experiments

Based on our experiences, we developed a framework to support researchers to plan and conduct experiments in the context of program comprehension. It consists of four parts: First, we developed a questionnaire to reliably measure programming experience. Second, we implemented a tool for presenting source code, tasks, and questionnaires to participants. Third, we documented confounding parameters for program comprehension. Last, we developed teaching material and holding according lectures to train the empirical researchers of tomorrow. We discuss each part of the framework in more detail in this section.

4.1 Programming-Experience Questionnaire

The first part of our framework is a questionnaire to measure programming experience, one of the major confounding parameters for program comprehension. To this end, we conducted a literature survey of seven major journals and conferences of the last ten years [FKL$^+$12]. We reviewed all papers that reported program-comprehension experiments and extracted how programming experience was defined and measured. Based on these insights, we refined the questionnaire we developed for our first experiment.

It consists of the following four categories (in Appendix 8.1, we show the complete questionnaire):

- Years (related to the amount of time participants spent with programming)
- Education (related to experience participants gained from education)
- Self estimation (participants had to estimate their experience with several topics)

No.	Question	ρ	N
Self estimation			
1	s.PE	.539	70
2	s.Experts	.292	126
3	s.ClassMates	.403	127
4	s.Java	.277	124
5	s.C	.057	127
6	s.Hasekll	.252	128
7	s.Prolog	.186	128
8	s.NumLanguages	.182	118
9	s.Functional	.238	127
10	s.Imperative	.244	128
11	s.Logical	.128	126
12	s.ObjectOriented	.354	127
Years			
13	y.Prog	.359	123
14	y.ProgProf	.004	127
Education			
15	e.Years	-.058	126
16	e.Courses	.135	123
Size			
17	z.Size	-.108	128

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

Table 1: Spearman correlations of number of correct answers with answers in questionnaire.

- Size (size of projects participants had worked with)

To evaluate whether this questionnaire is suitable for measuring programming experience, we evaluated it in a controlled experiment with over 100 undergraduate computer-science students. Specifically, we compared the answers of students in the questionnaire with the number of correct answers for ten programming tasks—the more tasks participants are able to solve correctly within the given time frame (40 minutes), the higher their programming experience should be, and the higher they should estimate their experience in the questionnaire. In Table 1, we show the correlations with the number of correct answers with each question in the questionnaire (in Table 4 in the Appendix, we explain the abbreviations.). A high correlation indicates that a question is suitable to describe programming experience (operationalized by the number of correct answers).

So far, self estimation appears to be a good indicator to assess programming experience. However, questions also correlate among each other. For example, participants who estimate a high experience with logical programming also estimate a high experience with Prolog (a logical programming language typically taught at German universities). Thus, using simply all questions with a high correlation with the number of correct answers is not sufficient. Instead, we need to use *partial correlations*, which is the correlation of two variables that is *cleaned* by the influence of a third variable [Bor04].

Thus, to extract the most relevant questions that best describe programming experience, we used stepwise regression, which is based on partial correlations of variables. With stepwise regression, we identified two relevant questions: The self-estimated experience with logical programming and the self-estimated experience compared to the class mates of students. These two questions can be supplied with control questions (e.g., self estimated experience with Prolog or self estimated programming experience) and used to measure the programming experience of participants. In future work, we plan to further validate our questionnaire and confirm that these two questions are the best indicator for programming experience.

4.2 Program-Comprehension Experiment Tool

As a second part of our framework, we developed the tool PROPHET to present source code, questionnaires, and tasks to participants [FS12]. It is a complex tool for planning and conducting experiments. To support a variety of program-comprehension experiments, we consulted the papers of our literature review again and analyzed the requirements for conducted experiments. Based on these requirements, we implemented PROPHET, which is available at the project's website.

PROPHET has two views, one for the experimenter and one for the participants. In the experimenter view, experimenters can decide how to present source code to participants by selecting check boxes (e.g., with or without syntax highlighting or allowing a search function or not). To give an impression, we show one tab of the experimenter view in Figure 3. In Appendix 8.2, we show additional screen shots.

To customize how participants see source code and tasks, experimenters select check boxes in the preferences tab of the experimenter view, shown in Figure 3 of the Appendix. For example, experimenters can define a file that is displayed when a task begins, choose what behavior of participants to log, whether participants see line numbers or are allowed to use the search, as well as specify a time limit for a task or the complete experiment.

In the view for participants, source code is presented as specified by the experimenter. In a second window, the tasks, questions, and forms for answers are presented.

We used PROPHET for our experiments since 2011 and found it very helpful to prepare the experiments. We did not have to implement any new source code or adapt source code of our existing tool infra structures. Instead of days to prepare the tasks and questions, we now need only hours. Thus, PROPHET saved us a considerable amount of time. We also found that other researchers used PROPHET for their experiments, indicating that PROPHET is general enough to support other researchers. We encourage researchers to use PROPHET for their experiments and give us feedback about their experience.

4.3 List of Confounding Parameters

The next part of our framework is a list of confounding parameters for program comprehension. To this end, we again consulted the paper of our literature review and, this time, extracted parameters that authors treated as confounding parameters.

We identified two categories of parameters: personal and experimental parameters. First, personal parameters are related to the participants, such as programming experience, intelligence, or domain knowledge. In total, we extracted 16 personal parameters. Second, experimental parameters are related to the setting of the experiment, such as the familiarity of participants with tools used, fatigue, or the layout of the study object. We found 23 experimental parameters.

In Appendix 8.3, we present an overview of all currently identified confounding parameters. To give an impression of the nature of confounding parameters, we present the most important parameters based on how often researchers considered them in their study. First, in 112 (of 158) papers, programming experience was mentioned as confounding parameter. Programming experience describes the experience participants have with implementing and understanding source code. The higher the experience, the better participants comprehend source code. Second, familiarity of study object (76) describes how familiar participants are with the concepts being studied, such as Oracle or UML. The more familiar they are, the better they might perform in an experiment. Both, programming experience and familiarity with study object, are personal parameters.

The third parameter is the programming language (72). Participants who are familiar with a language perform different than participants who are not. Fourth, the size of the study object (67) describes how large the study object is, for example in terms of lines of code or number of classes. Last, learning effects were mentioned in 65 papers, which describe that participants learn during an experiment. Thus, they might perform better at the end of an experiment, because they learned how to solve tasks.

With a list in which we describe each possible parameter, researchers can decide for each parameter on the list whether it is relevant for their study and select a suitable control technique. They do not have to put too much effort in identifying the parameters. When we plan our experiments, we are now traversing the list and discuss for each parameter whether it is relevant or not. This saved us considerable time and effort.

To describe how confounding parameters are managed, we suggest a pattern like the one in Table 2. It contains the applied control technique and rationale for the decision, as well as the used measurement technique and the rationale for the technique. This way, readers of a report can get a quick overview of how confounding parameters were managed.

This work will be continued as long as researchers publish experiments, because there might always be parameters that have not been considered so far. Thus, the list of confounding parameters is intended to grow. On the project's website, we present the status of the work, including the currently reviewed papers and extracted parameters. We explicitly encourage other researchers to extend the review with new papers of new venues.

Parameter (Abbreviation)	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 2: *Pattern to describe confounding parameters.*

4.4 Teaching Material

In the German computer-science curricula, empirical methods are under-represented or even neglected at most universities. However, empirical methods are an important aspect also beyond computer science. For example, in psychology, students learn from the first semester how to plan experiments, how to solve the difficulties, how to analyze data, how to develop questionnaires, and so forth.

To improve the current situation, we designed a lecture, in which we teach students the basic methods of empirical research. For example, we teach how to apply the think-aloud protocol or how to set up performance measurements. Furthermore, we teach methods to analyze the data and conduct hypothesis tests to differentiate a random effect from a real effect. So far, students were interested and engaged in the topics of the lecture.

To let students apply the learned methods, students designed, conducted, and analyzed their own experiments and submitted a report. When looking at the report of students, we found that the experiments were carefully designed and analyzed and that some of the experiments are even good enough to be published. Thus, there is evidence that there is a need and interest to learn and teach empirical methods. In the future, we hope to motivate students to select an empirical topic for their bachelor or master's thesis and maybe continue to use empirical methods in a PhD thesis.

So far, this lecture took place at the Philipps University Marburg (held by Christian Kästner). Currently, the lecture takes place at the University of Magdeburg (held by Janet Siegmund). Since we had positive feedback from our students and since we could reuse the material of the lecture, we are planning to offer the lecture again. So far, the lecture is planned at the University of Passau and Carnegie Mellon University. We also are happy to share our experience and teaching material with other researchers to support others in training the empirical researchers of tomorrow.

5 Applying the Framework

To show that our framework helps to conduct experiments, we discuss how it helped us to design two of our experiments. First, we set up and pilot tested an experiment to evaluate how physical and virtual separation of concerns affect program comprehension [SKLA12]. We could consult the list of confounding parameters and decide for each parameter how important its influence is and how we can control for it. We also could apply the questionnaire to measure programming experience and create balanced groups with a comparable experience level. Third, for presenting the task material, and programming-experience questionnaire, we could use PROPHET. The pilot study was conducted at the University of Passau, without the responsible experimenter being present. Instead, a colleague in Passau conducted the experiment without any difficulties.

Second, we conducted an experiment to evaluate whether the derivation of a model is easier to understand than the model itself [FBR12]. We could also consult the list of confounding parameters, because the selection criteria for papers of the literature survey were broad enough to also include experiments on model comprehension. We could also reuse the questionnaire for programming experience, but had to add questions related to model comprehension. We could not use the tool PROPHET, because it does not support modifying images, which was one of the tasks—currently, we can only *display* images. However, we are working on PROPHET to also support the modification of images.

In summary, we now only needed weeks instead of months to set up the experiments. We also encourage other researchers to use the framework and give us feedback how it helped

6 Conclusion and Vision

Although there is effort to establish an empirical research community, empirical research is still reluctantly applied to evaluate new techniques and tools that target, among others, better comprehensibility. We believe that one reason is the effort that accompanies applying empirical methods, such as controlled experiments.

Based on our experience, we developed a framework to help other researchers in overcoming the obstacle of designing the first controlled experiment. The framework consists of a list of confounding parameters and a questionnaire to measure programming experience, the most important confounding parameter. Furthermore, we developed the tool PROPHET to support researchers during planning and conducting experiments. Additionally, we invest our effort in training the empirical researchers of tomorrow.

In future work, we want to explore further strategies to measure program comprehension. In cognitive neuro science, researchers successfully use *functional magnetic resonance imaging* to visualize cognitive processes. In collaboration with André Brechmann, a neuro biologist, we are currently exploring whether functional magnetic resonance imaging can also be used to measure program comprehension [SBA$^+$12].

7 Acknowledgments

Thanks to Norbert Siegmund for fruitful discussions. Siegmund's and Saake's work is funded by BMBF project 01IM10002B, Kästner's work partly by ERC grant #203099, and Apel's work by the German Research Foundation (DFG – AP 206/2, AP 206/4, and LE 912/13).

References

[ABPC91] Neil Anderson, Lyle Bachman, Kyle Perkins, and Andrew Cohen. An Exploratory Study into the Construct Validity of a Reading Comprehension Test: Triangulation of Data Sources. *Language Testing*, 8(1):41–66, 1991.

[BDLM09] 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.

[Boe81] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[Bor04] Jürgen Bortz. *Statistik: für Human- und Sozialwissenschaftler*. Springer, sixth edition, 2004.

[Boy77] John Boysen. *Factors Affecting Computer Program Comprehension*. PhD thesis, Iowa State University, 1977.

[Bro78] 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.

[DR00] Alastair Dunsmore and Marc Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS 35-2000, Department of Computer Science, University of Strathclyde, 2000.

[FALK11] 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, 2011. paper 3.

[FBR12] 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, 2012.

[FCM$^+$08] Eduardo Figueiredo, Nelio Cacho, Mario Monteiro, Uira Kulesza, Ro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 261–270. ACM Press, 2008.

[Fei09] Janet Feigenspan. Empirical Comparison of FOSD Approaches Regarding Program Comprehension – A Feasibility Study. Master's thesis, University of Magdeburg, 2009.

[FKA$^+$12] 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.*, 2012. DOI: 10.1007/s10664-012-9208-x.

[FKL$^+$12] 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, 2012.

[FOMMH10] Thomas Fritz, Jingwen Ou, Gail Murphy, and Emerson Murphy-Hill. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 385–394. IEEE CS, 2010.

[FS12] 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.

[FSF11] 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. GI, 2011. `http://wwwiti.cs.uni-magdeburg.de/iti\_db/publikationen/ps/auto/FeSiFr11.pdf`.

[FSP$^+$11] 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, 2011.

[HS95] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.

[KAK08] 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.

[Kit93] Barbara Kitchenham. A Methodology for Evaluating software Engineering Methods and Tools. In H. Rombach, Victor Basili, and Richard Selby, editors, *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, volume 706 of *Lecture Notes in Computer Science*, pages 121–124. Springer, 1993.

[KLM$^+$97] 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.

[KTS$^+$09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 611–614. IEEE CS, 2009. Tool demo.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[Pen87] Nancy Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.

[SBA$^+$12] 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, 2012. New ideas track. Submitted 29.6., Accepted 14.8.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, 1984.

[SFF$^+$11] 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.

[Sie12] Janet Siegmund. *Framework for Measuring Program Comprehension*. PhD thesis, University of Magdeburg, Submitted in August 2012.

[SKLA12] 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, 2012. Submitted 02.07, Accepted 07.08.

[SM79] 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.

[SM10] 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.

[Sta84] Thomas Standish. An Essay on Software Reuse. *IEEE Trans. Softw. Eng.*, SE–10(5):494–497, 1984.

[SV95] 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.

[Tia11] Rebecca Tiarks. What Programmers Really Do: An Observational Study. In *Proc. Workshop Software Reengineering (WSR)*, pages 36–37. GI, 2011.

[Tic98] Walter F. Tichy. Should Computer Scientists Experiment More? *Computer*, 31(5):32–40, 1998.

[TLPH95] Walter Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[vMVH97] 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.

8 Appendix

8.1 Programming-Experience Questionnaire

In this section, we show the programming-experience questionnaire. In Table 4, we summarize all questions, including the category to which it belongs, the scale for the answer, and the abbreviation, which we use in the remaining tables.

In Table 3, we show the results of an exploratory factor analysis. Our goal was to look for clusters of questions that show a high correlation among themselves. This way, we intend to develop a model that describes programming experience. The next step is to confirm our model with a different sample. To this end, we are currently collecting data of students from different German universities.

Variable	Factor 1	Factor 2	Factor 3	Factor 4	Factor 5
s.C	.723				
s.ObjectOriented	.700			.403	
s.Imperative	.673	.333		.303	
s.Experts	.600	.326			
s.Java	.540		.427		
y.ProgProf		.859			
z.Size		.764			
s.NumLanguages	.335	.489		.403	
s.ClassMates		.449	.403	.424	
s.Functional			.880		
s.Haskell			.879		
e.Courses				.795	
e.Years			-.460	.573	
y.Prog		.493		.554	
s.Logical					.905
s.Prolog					.883

Gray cells denote main factor loadings.

Table 3: Factor loadings of variables in questionnaire.

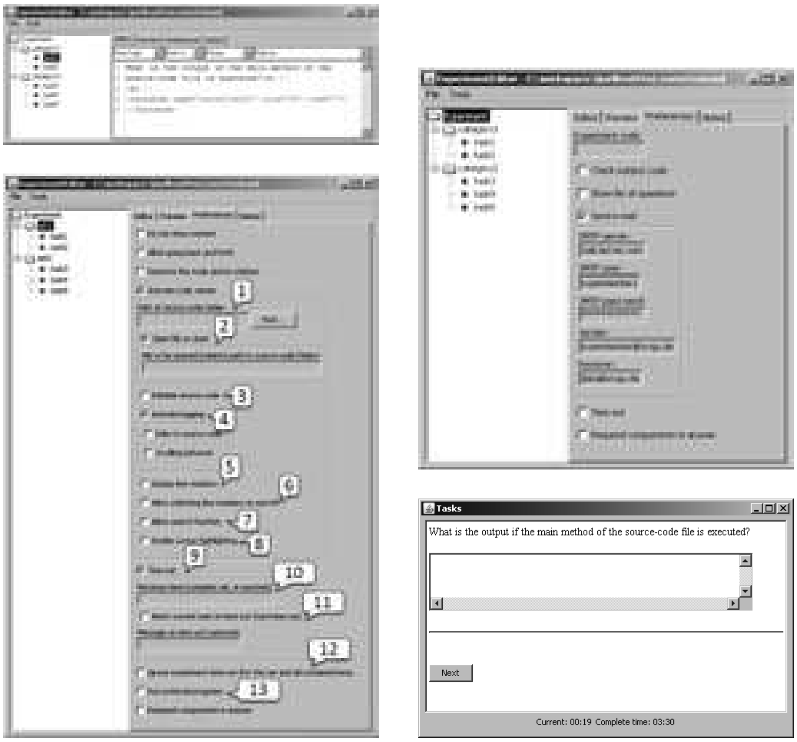
Source	Question	Scale	Abbreviation
Self estimation	On a scale from 1 to 10, how do you estimate your programming experience?	1: very inexperienced to 10: very experienced	s.PE
	How do you estimate your programming experience compared to experts with 20 years of practical experience?	1: very inexperienced to 5: very experienced	s.Experts
	How do you estimate your programming experience compared to your class mates?	1: very inexperienced to 5: very experienced	s.ClassMates
	How experienced are you with the following languages: Java/C/Haskell/Prolog	1: very inexperienced to 5: very experienced	s.Java/s.Prolog/ s.C /s.Haskell
	How many additional languages do you know (medium experience or better)?	Integer	s.NumLanguages
	How experienced are you with the following programming paradigms: functional/imperative/logical/object-oriented programming?	1: very inexperienced to 5: very experienced	s.Functional / s.Imperative / s.Logical / s.ObjectOriented
Years	For how many years have you been programming?	Integer	y.Prog
	For how many years have you been programming for larger software projects, e.g., in a company?	Integer	y.ProgProf
Education	What year did you enroll at university?	Integer	e.Years
	How many courses did you take in which you had to implement source code?	Integer	e.Courses
Size	How large were the professional projects typically?	NA, <900, 900-40000, >40000	z.Size
Other	How old are you?	Integer	o.Age

Integer: Answer is an integer; The abbreviation of each question encodes also the category to which it belongs.

Table 4: Overview of questions to assess programming-experience.

8.2 Prophet

Our tool PROPHET (short for PROgram ComPreHension Experiment Tool) supports experimenters in creating experiments and presenting material to participants. In the experimenter view, experimenters can set up the tasks with HTML. We provide common HTML elements in drop-down lists, as shown in Figure 3.



What is the output if the main method of the source-code file is executed?

Current: 00:19 Complete time: 03:30

Figure 3: Top left: task definition; bottom left: preferences for categories; top right: preferences for the complete experiment; bottom right: task viewer for participants.

Experimenters can also specify settings for a complete experiment. By selecting the check box “Send e-mail”, the data of participants are zipped and sent from the specified sender address to the specified receiver address without requiring interaction from participants. Additionally, experimenters can set a time limit for the complete experiment (check box “time out”), after which the experiment is aborted.

Last, we show the view for participants, which shows the task as specified. Participants enter their answer in the text area and navigate forward with the button (labeled “Next”).

8.3 Confounding Parameters

As last part of the appendix, we show the confounding parameters we extracted for each journal and conference. For better overview, we divide personal parameters (shown in Table 5 into the categories personal background (i.e., parameters that are defined with the birth and that typically do not change), personal knowledge (i.e., parameters that change only slowly over the course of weeks), and personal circumstances (i.e., parameters that change rapidly, even within minutes). Furthermore, we divide experimental parameters into the categories subject related (i.e., parameters related to the person of the participants, but that occur only because they take part in an experiment), technical (i.e., parameters related to the setting of the experiment), context related (i.e., parameters that typically occur in nearly all experiments), and study-object related (i.e., parameters related to properties of the object under evaluation)[2].

To create a sound experimental design, we recommend traversing this list of parameters and discuss for each parameter whether it is relevant for the experiment and document this process. Additionally, we should also document how we controlled for a confounding parameter. This way, other researchers can consult this documentation when designing experiments and may not trip over neglecting confounding parameters.

Parameter	ESE	TOSEM	TSE	ICPC	ICSE	ESEM	FSE	Sum
Personal background								
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								
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								
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 5: *Number of personal confounding parameters mentioned per journal/conference.*

[2]We described each parameter in detail in our PhD thesis [Sie12]

Parameter	ESE	TOSEM	TSE	ICPC	ICSE	ESEM	FSE	Sum
Subject related								
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								
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								
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								
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 6: *Experimental confounding parameters.*