

12-2014

Measuring and Modeling Programming Experience

Janet Siegmund
Universität Passau

Christian Kästner
Carnegie Mellon University

Jörg Liebig
Universität Passau

Sven Apel
Universität Passau

Stefan Hanenberg
Universität Duisburg-Essen

Follow this and additional works at: <http://repository.cmu.edu/isr>

 Part of the [Software Engineering Commons](#)

Published In

Empirical Software Engineering, 19, 5, 1299-1334.

This Article is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Institute for Software Research by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Measuring and Modeling Programming Experience

Janet Siegmund · Christian Kästner ·
Jörg Liebig · Sven Apel · Stefan
Hanenberg

Received: date / Accepted: date

Abstract Programming experience is an important confounding parameter in controlled experiments regarding program comprehension. In literature, ways to measure or control programming experience vary. Often, researchers neglect it or do not specify how they controlled for it. We set out to find a well-defined understanding of programming experience and a way to measure it. From published comprehension experiments, we extracted questions that assess programming experience. In a controlled experiment, we compare the answers of computer-science students to these questions with their performance in solving program-comprehension tasks. We found that self estimation seems to be a reliable way to measure programming experience. Furthermore, we applied exploratory and confirmatory factor analyses to extract and evaluate a model of programming experience. With our analysis, we initiate a path toward validly and reliably measuring and describing programming experience to better understand and control its influence in program-comprehension experiments.

1 Introduction

In software-engineering experiments, program comprehension is frequently measured, for example, for the evaluation of programming-language constructs or software-development tools [3, 9, 18, 22, 35]. Program comprehension is an internal cognitive process that we cannot observe directly. Instead, we often conduct controlled experiments, in which we observe the behavior of participants and draw conclusions about their program comprehension.

To conduct controlled experiments, we have to control for confounding parameters, which influence the outcome of an experiment in addition to the evaluated concept [21]. One important confounding parameter is programming experience, which can be defined as “the amount of acquired knowledge regarding the development of programs, so that the ability to analyze and create programs is improved”. The more experienced a participant, the better she understands a program compared to an inexperienced participant. (Accidentally)

Janet Siegmund (born Feigenspan), Jörg Liebig, and Sven Apel
University of Passau
E-mail: siegmunj@fim.uni-passau.de, joliebig@fim.uni-passau.de, apel@uni-passau.de

Christian Kästner
Carnegie Mellon University

Stefan Hanenberg
University of Duisburg-Essen
E-mail: stefan.hanenberg@icb.uni-due.de

assigning more experienced participants to one treatment can seriously bias the results. Hence, programming experience should always be considered in such kind of experiments.

However, there is no agreed way to measure programming experience. Instead, researchers use different, sometimes not specified, measures or do not assess it at all. However, a common understanding of programming experience can increase the validity of experiments and helps interpreting results.

Our goal is to evaluate how reliable different ways to measure programming experience are. To this end, we conducted a controlled experiment, in which participants completed a questionnaire that contained questions related to programming experience based on a literature review. Additionally, participants solved programming tasks. Then, we compared the performance in the programming tasks with the answers in the questionnaire.

As a result, we identified two questions as an indicator for programming experience using stepwise regression: Self-estimated programming experience compared to class mates and self-estimated experience with logical programming. Furthermore, we propose a five-factor model that describes programming experience using exploratory and confirmatory factor analyses. The contributions of this paper are the following:

- Literature review about the state of the art of measuring and controlling the influence of programming experience.
- A questionnaire that contains common questions to measure programming experience.
- Reusable experimental design to evaluate the questionnaire.
- Initial evaluation of this questionnaire with undergraduate students.
- Proposal and evaluation of two relevant questions to measure programming experience.
- Proposal and evaluation of a model of programming experience.

This article is an extended version of the conference paper “Measuring Programming Experience” presented at the International Conference on Program Comprehension [17]. The first four contributions were already made in the conference paper. In this version, we add an evaluation of the two relevant questions to measure programming experience based on additional data (Section 6.3). This way, we can confirm that the two questions are suitable indicators for programming experience. Furthermore, we evaluate the extracted model of programming experience with a confirmatory factor analysis, for which we collected new data of 148 students (Section 7.2). Thus, we get a better understanding of the nature of programming experience.

2 Literature Review

To get an overview of whether and how researchers measure programming experience, we conducted a literature review based on the guidelines for systematic literature reviews provided by Kitchenham and Chartes [28]. We considered the years 2001 to 2010 of highly ranked conferences and journals in the domain of (empirical) software engineering and program comprehension: *International Conference on Software Engineering (ICSE)*, *European Software Engineering Conference/Symposium on the Foundations on Software Engineering (FSE)*, *International Conference on Program Comprehension (ICPC)*,¹ *International Symposium on Software Engineering and Measurement (ESEM)*,²

¹ ICPC was a workshop until 2005 (IWPC), which we also included.

² ESEM first took place in 2007.

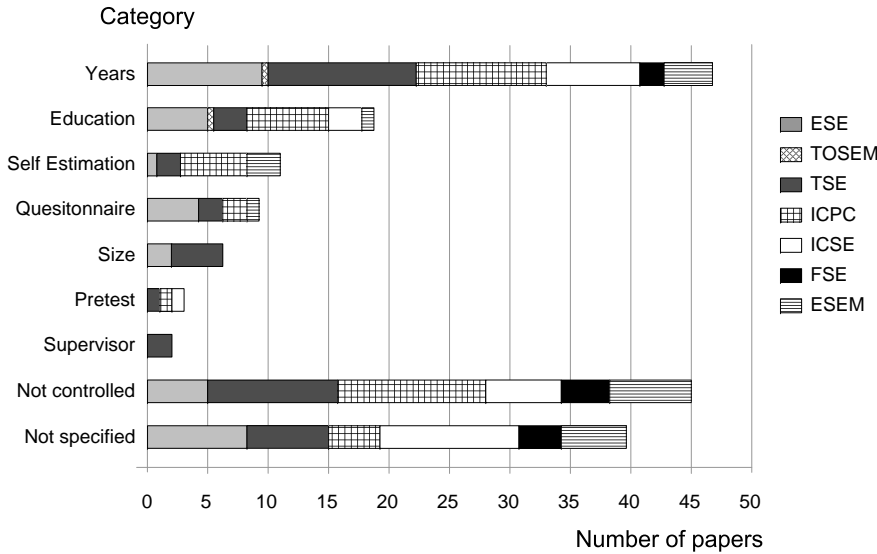


Fig. 1: Overview of how programming experience is operationalized.

Empirical Software Engineering Journal (ESE), *Transactions on Software Engineering (TSE)*, and *Transactions on Software Engineering and Methodology (TOSEM)*.

To extract the papers, we read title and abstract of each paper. If an experiment with human participants was mentioned, we included the paper in our selection. If the abstract was not conclusive, we skimmed the paper and searched for the keywords (programming) experience, expert, expertise, professional, participant, and participant, which are typical for program-comprehension experiments. We extracted 288 (of 2161) papers. We read each paper of our selection and excluded those papers that evaluated a concept too far away from program comprehension (e.g., cost estimation of software projects). When uncertain whether a concept was too far away, we discussed it until we reached an agreement. The literature-review team consisted of the first author and a research assistant. When still in doubt, we included the paper to have a broad overview of the understanding of programming experience. The final selection consists of 161 papers. An overview of our initial and final selection of papers is available at the project's website (<http://fosd.net/PE/>).

In the selected papers, we found several ways of managing programming experience, which we divide into 9 categories (Fig. 1). The categories are not disjoint; when authors combined indicators, the according paper counts for each category.

1. *Years*: In many papers (47), the years a participant was programming at all or programming in a company or certain language was used to measure programming experience. For example, Sillito and others assessed the number of years a participant was programming professionally [35].
2. *Education*: The education of participants was used to indicate their experience in 19 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 [33].
3. *Self estimation*: In twelve papers, participants were asked to estimate their experience themselves. For example, Bunse let his participants estimate their experience on a five-point scale [9].

4. *Unspecified questionnaire*: Some authors applied a questionnaire to assess programming experience (9 papers). For example, Erdogmus and others let participants fill out a questionnaire before the experiment [14]. However, it was not specified what the questionnaire looked like.
5. *Size*: The size of programs participants had written was used as an indicator in six papers. For example, Müller [31] asked how many lines of code the largest program has that participants have implemented.
6. *Unspecified pretest*: In three papers, a pretest was conducted to assess the participants' programming experience. For example, Biffi and Grossmann [7] used a pretest to create three groups of skill levels (excellent, medium, little). However, it was not specified in the papers what the pretest looked like.
7. *Supervisor*: In two papers, in which professional programmers were recruited as participants, the supervising manager estimated the experience of participants [3, 23].
8. *Not specified/not controlled*: Often, the authors state that they measured programming experience, but did not specify how. This was the case in 39 papers. Even more often (45 papers), programming experience was not mentioned at all, which may threaten the validity of the corresponding 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, we encountered similarities in the opinion, which we summarized in the following preliminary definition:

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

Note that experience and ability appear very similar, but differ slightly. Experience describes what developers have done and learned when programming, not how good they were at it.

To summarize, the measurement of programming experience is diverse. This could threaten the validity of experiments, because researchers use their own definition of programming experience without validating it. Furthermore, conducting meta analysis on these experiments is difficult, because the influence of programming experience is not clearly defined, making the results across different experiments not comparable. To evaluate the measurement of programming experience, we created a questionnaire based on the results of the literature review.

3 Questionnaire

Most measurements of programming experience 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. 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 following categories: *years*, *education*, *self estimation*, and *size*. For each category,

we selected multiple questions we found in literature. Additionally, we added questions that we found in previous experiments to be related to programming experience. This way, we aim at having a more exhaustive set of indicators for programming experience and, consequently, a better definition of programming experience. Some questions are specific to students; when working with different participants (e.g., experts), they need to be adapted.

Our goal is to evaluate which questions from which categories have the highest prediction power for programming experience. In the long run, we plan to evolve our questionnaire (by removing questions with little prediction power and potentially adding others) into a standard questionnaire.

In Table 1, we summarize our questionnaire. We also show the scale of the answers, that is, how participants should answer the questions. In column “Abbreviation”, we show the abbreviation of each question, which we use in the remainder of this paper. The version of the questionnaire we used in our experiment is available at the project’s website. Next, we explain each question in detail.

3.1 Years

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

3.2 Education

This category contains questions that assess educational aspects. We asked participants to state the number of courses they took in which they implemented source code and the year in which they enrolled (recoded into number of years a participant has been enrolled). The number of courses roughly indicates how much source code participants had implemented. With the years a participant is studying, we get an indicator of the education level: The longer a participant has been studying, the more experience she should have gained through her studies.

Table 1: Questions to assess programming experience.

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

3.3 Self Estimation

In this category, participants were asked to estimate their own experience level. We included several questions in this category. With the first question, we asked participants to estimate their programming experience on a scale from 1 to 10. 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 we think that 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 and their class mates 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, C, Haskell, and Prolog, because these are common and are taught at the universities our participants were enrolled at. The more programming languages developers 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 list the number of programming languages in which they are experienced at least to a medium level. The same counts for familiarity with different programming paradigms.

3.4 Size

We asked participants with professional experience about the size of their projects. We used the categorization into small, medium, and large based on the lines of code according to van Mayrhauser and Vans [38]. Since with increasing size, software systems tend to get more complex, a larger size may also mean higher programming experience.

In addition, we also included the age of participants in the questionnaire, because the older participants are, the more time they had to increase their programming experience. This way, we aim at having a more exhaustive understanding of programming experience.

4 Empirical Validation

Constructing and validating a questionnaire is a long and tedious endeavor that requires several (replicated) experiments [32]. In this paper, we start this process.

To this end, we use a two-step approach: First, we recruited undergraduate computer-science students and compared their answers in the questionnaire with performance in tasks that are related to programming experience. Based on these data, we extracted questions of the questionnaire that showed a sufficient correlation to programming experience (in terms of performance in programming tasks).

Second, we compared the answers of undergraduate and graduate students in the questionnaire. Since graduate students have more programming experience than undergraduate students, the extracted questions should reflect the difference in programming experience—otherwise, they cannot differentiate between the experience level of graduate and undergraduate students.

We recruited students, because we found in our review that they are often recruited as participants in software-engineering experiments. Hence, they represent an important sample. Furthermore, students can be comparable to experts under certain conditions [24,37].

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

To present our experiment, we use the guidelines suggested by Jedlitschka and Ciolkowski [27]. For brevity, we describe only necessary details to understand our experiment. More information (e.g., tasks, overview of statistical analysis) is available at the project's website.

4.1 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. This operationalization of programming experience is based on two assumptions: First, 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. Second, 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.

4.2 Material

We designed 10 program-comprehension tasks which we presented in a fixed order. We gave participants source code and asked what executing this code would print. To succeed, participants had to mentally simulate the code—executing or editing it was not possible. Furthermore, participants had to explain what the source code is doing. In Figure 2, we show the source code of the first task to give an impression (all other tasks are available on the project's website). The source code sorts an array of numbers, so the correct answer is 5, 7, 14. The remaining tasks were roughly similar: Two tasks were about a stack, five about a linked list, one involved command-line parameters, and the last was a bug-fixing task. An answer was correct when it matches the outcome of the given program, ignoring whitespace. When an answer diverged from the expected result, a programming expert looked at participants' explanation of the source code and decided whether the answer could be counted as correct.

To match the average experience level of undergraduate students (who we recruited as participants), we selected typical algorithms presented in introductory programming lectures. Of course, these simple tasks are not appropriate in every context, for example, when working with graduate students or professional programmers. Furthermore, not letting participants execute the code limits external validity. However, while running a program and having sophisticated tool support may help with understanding source code, programmers

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

Fig. 2: Source code for the first task.

still have to read code by themselves for understanding. Additionally, while large tasks are usually preferable, an experiment with large tasks is not easy to conduct and unlikely fits into a university schedule.

To identify highly experienced participants among second-year undergraduate students (since some students start programming before their study), we included two tasks that required a higher experience level: In Task 9, we used command-line parameters, which are not typically taught at undergraduate level. In the last task, we use source code of MobileMedia, a software for manipulating multi-media data on mobile devices [20]. It consists of 2 800 lines of code in 21 classes. We expected that only highly experienced participants should be able to complete this task. All source code was in Java, the language that participants were most familiar with.

We had 10 tasks so that only experienced participants would be able to complete all tasks in the given time, 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 given time.

To present the questionnaire, tasks, and source code, we used our tool infrastructure PROPHET³, which we designed to conduct comprehension experiments [19]. It lets participants enter answers, logs the time participants spend on each task, and logs the behavior of participants (e.g., opening files). This way, we control the influence of participants' familiarity with an IDE. There may be other confounding parameters, such as intelligence or ability; however, with our large sample, the influence of these confounders should be ruled out.

4.3 Participants

Participants came 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

³ <http://www.infosun.fim.uni-passau.de/spl/janet/prophet/>

of all participants was comparable, because no courses took place between semesters and participants had to complete similar courses at all universities. Thus, we reduced the variation of programming experience, which makes it easier to reveal possible relationships between questions in the questionnaire and answers to tasks. Since this is the first experiment to evaluate the programming-experience questionnaire, restricting the sample to a homogeneous group is a legitimate step. We discuss resulting limitations of external validity in Section 8.

All students 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 students participated voluntarily, were aware that they took part in an experiment, and could quit anytime. Data was logged anonymously.

Since we recruited participants from different universities, we actually have 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 for `s.ClassMates` revealed no significant differences between the three universities ($\chi^2 = 1.275$, $df = 2$, $p = 0.529$) [1]. 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.

To control for cognitive bias, that is, that students over- or underestimate their experience, we asked our subjects to do their best effort in giving correct and honest answers. Furthermore, since our sample is large, we assume that over/underestimation behave like a random variable and, thus, cancel each other out.

4.4 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. They were told to work as quickly and as correctly as possible. Since we had time constraints, the time limit for the experiment was set to 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 participants.

4.5 Deviation

The presentation of the programming-experience questionnaire had a bug, such that we could not measure `s.PE` for all participants. Hence, we only have the answer of 70 out of 128 participants for this question.

5 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

Table 2: Overview of response time for each task.

Variable	Response time		N	Correct
	Distribution	Mean		
Task 1		4.44	124	70
Task 2		3.65	123	90
Task 3		5.02	121	97
Task 4		6.17	117	22
Task 5		4.06	118	46
Task 6		4.72	111	40
Task 7		2.34	92	31
Task 8		4.1	82	69
Task 9		1.94	78	11
Task 10		9.64	30	22

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

tasks. This way, we get an impression of how important each question is as an indicator for programming experience in our sample.

5.1 Means and Standard Deviations

In Table 2, 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 the large underlying size of the source code for the last task with over 2800 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. Furthermore, only 11 participants solved it correctly. To solve this task, participants must be familiar with command-line parameters, which is not typical for the average second-year undergraduate student. Considering the correctness of Task 4, we see that only 22 participants solved this task 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 Fig. 3, we show the number of correctly solved tasks per participant. As we expected, none of our participants solved more than eight tasks correctly. (cf. Section 4.2). Especially the last two tasks (Task 9: command-line parameters; Task 10: 2,800 lines of code) 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 tasks that a student solved correctly lies in this interval.

In Table 3, 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 4 years, but only few

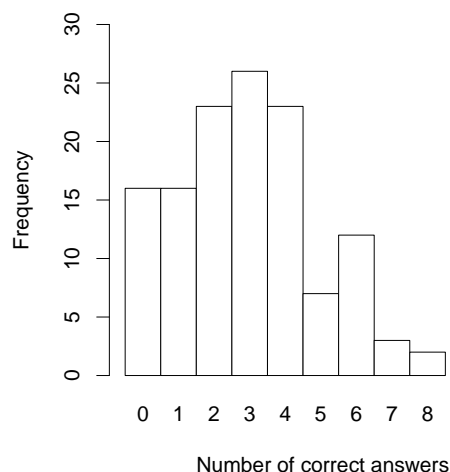


Fig. 3: Frequencies of number of correct answers.

Table 3: Overview of answers in questionnaire.

No.	Question	Distribution	N
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

participants said they were programming for more than 10 years. Although participants took an undergraduate course, some participants were enrolled for more than 3 years,⁴ which could also explain why some participants completed numerous courses in which they had to implement source code.

5.2 Correlations

In Table 4, 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 [1]. For about half of the questions of self estimation, we obtain small to strong correlations.⁵ The highest correlation with number of correct answer 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

⁴ The German system allows students to take courses in a flexible order and timing.

⁵ Small: ± 0.1 to ± 0.3 ; medium: ± 0.3 to ± 0.5 ; strong: ± 0.5 to ± 1 [10].

Table 4: Spearman correlations of number of correct answers with answers in questionnaire.

No.	Question	ρ	N
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.Haskell	.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
13	y.Prog	.359	123
14	y.ProgProf	.004	127
15	e.Years	-.058	126
16	e.Courses	.135	123
17	z.Size	-.108	128
18	o.Age	-.116	128

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

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. Only 23 correlations, of 180, are significant, which is in the range of coincidence, given the common α 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 large number of participants. Since we have a decreasing number of participants with each task, we leave analyzing the response times for future experiments. We discuss resulting limitations in Section 8.

6 Which Questions Measure Programming Experience?

In this section, we apply stepwise regression to find the most relevant questions to measure programming experience. For this analysis, we excluded question s.PE, because only 70 participants answered this question (cf. Section 4.5). 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. Furthermore, we only use the number of correct answers as an indicator for program comprehension, but not time, since only few participants completed all tasks. We decided not to compute the average response time for a task or to analyze the response times for each task, because that would be too inaccurate.

6.1 Overview of 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 ($> .30$) with the number of correctly solved task, because they are typically considered relevant. However, the questions themselves might correlate with each other. For example, the s.ClassMates correlates with

Table 5: Spearman correlations of response times for each task with answers in questionnaire.

Question	1	2	3	4	5	6	7	8	9	10	N
s.PE	-.279	-.417	-.042	.004	-.002	.016	.014	-.182	.071	.085	68 – 27
s.Experts	-.300	-.177	.047	-.026	.006	-.075	-.217	-.004	.206	.131	122 – 40
s.ClassMates	-.189	-.401	-.084	-.065	-.053	-.059	-.163	-.061	.161	.100	123 – 40
s.Java	.029	-.066	-.154	-.022	-.066	.003	-.040	.145	-.170	-.222	105 – 34
s.C	-.175	-.124	.018	.027	.126	-.108	-.056	-.052	.043	.108	123 – 40
s.Haskell	-.171	-.109	-.144	-.113	-.014	-.216	-.153	-.183	.019	.158	124 – 40
s.Prolog	-.174	-.141	-.079	-.104	-.027	-.039	.076	-.239	-.047	.146	124 – 40
s.NumLanguages	-.295	-.339	-.131	-.121	-.027	-.103	-.035	-.090	.232	.168	115 – 34
s.Functional	-.148	-.150	-.150	-.004	-.017	-.204	-.120	-.217	.027	.175	123 – 40
s.Imperative	-.283	.331	-.033	-.089	-.06	-.129	-.296	-.156	.126	.043	124 – 40
s.Logical	-.209	-.105	-.158	-.136	-.022	-.014	.058	-.257	-.191	.108	122 – 40
s.ObjectOriented	-.084	-.232	-.008	.012	-.093	-.034	.025	-.060	.156	.082	123 – 40
y.Prog	-.241	-.379	-.144	-.071	.010	-.113	-.258	-.159	.273	.180	120 – 38
y.ProgProf	-.217	-.196	-.012	-.119	-.130	.071	-.274	-.022	.044	-.010	123 – 39
e.Years	-.032	.001	.018	-.152	.059	.047	-.119	.037	-.092	-.173	122 – 40
e.Courses	-.146	-.088	-.040	-.062	.071	.028	-.053	-.004	.268	.058	120 – 38
z.Size	-.155	-.160	-.057	-.134	.059	.003	-.201	.046	.000	-.023	124 – 40
o.Age	.036	.014	.110	.082	.131	.102	-.081	.090	.059	.010	124 – 40

N: highest and smallest number of participants depending on the task;
 Gray cells denote significant correlations ($p < .05$).

s.ObjectOriented with 0.552. Hence, we can assume both questions are not independent from each other. If we used both questions as indicators, 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* [30]. Stepwise regression builds a model of the influence of the questions on the number of correct answers in a stepwise manner. It starts by including the question with the highest correlation, which, in our case, is s.ClassMates. Then, it considers the question with the next highest correlation, which is y.Prog. Using this question, it computes the *partial correlation* with the number of correct answers, describing the correlation of two variables cleaned from the influence of a third variable [11]. Thus, the correlation of y.Prog with the number of correct answers, cleaned from the influence of s.ClassMates, is computed. If this partial 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.

6.2 Results and Interpretation

In Table 6, we show the results for our questionnaire. With stepwise regression (specifically, we used *stepwise* as inclusion method), we extracted two questions: Experience with logical programming (s.Logical) and self-estimated experience compared to class mates (s.ClassMates). 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 (explaining the meaning of the values exceeds the scope of this paper; see [30]).

Table 6: Resulting model of stepwise regression.

Question	Beta	t	p value
s.ClassMates	.441	3.219	.002
s.Logical	.286	2.241	.030

Hence, the result of the stepwise-regression algorithm is that the 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 programming, 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 a participant estimates her experience better than her class mates, she 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.

The model 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 $0.441 * 4 + 0.286 * 2 = 2.336$ (we omitted a constant to add as part of the model for simplicity).


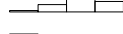



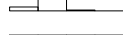
Hence, we have identified two questions that explain 24.1% of the variance of the number of correct answers. We could include more questions to improve the amount of explained variance, but none of the questions contribute a significant amount of variance. Since a model should be parsimonious, stepwise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

6.3 Evaluation of Extracted Questions

To evaluate whether both questions measure programming experience also in other samples, we consulted further data sets of 110 new participants in total. In our previous experiments regarding program comprehension, we always used variants of the programming-experience questionnaire [15,16,18,34] (also available at all the projects' websites: <http://www.infosun.fim.uni-passau.de/spl/janet/>). As participants, we mostly recruited graduate students (third-year undergraduate students also participated, but for better readability, we only use the term graduate students in this section), who typically have more programming experience than undergraduate students. Thus, the programming-experience value based on s.ClassMates and s.Logical should differ between the second-year undergraduate students of the current experiment and the graduate students of the previous experiments.

In Table 7, we compare the answers to both questions and the resulting programming-experience values computed as combination of both questions according to stepwise regression (i.e., $0.441*s.ClassMates + 0.286*s.Logical$) of both data sets. For s.ClassMates, the graduate students estimate their ex-

Table 7: Overview of answers in questionnaire.

Data set	Question	Distribution	N	U	p value
Undergraduate	s.ClassMates		123	2909	0.000
Graduate			110		
Undergraduate	s.Logical		123	6451	0.467
Graduate			110		
Undergraduate	Programming experience		123	2943	0.000
Graduate			110		

0 1 2 3 4 5

perience higher than undergraduate students. For s.Logical, the experience of graduate and undergraduate students appears comparable. The resulting programming-experience value of the graduate students appears to be higher compared to the undergraduate students. A Mann-Whitney-U test shows that the differences for s.ClassMates and programming experience are significant, but not for s.Logical. This is also reflected in the results of stepwise regression, such that s.Logical has a lower beta value and, hence, a lower influence. Thus, both questions combined can differentiate between the experience level of undergraduate and graduate students. In future work, it is interesting to evaluate whether the predictive power of s.ClassMates alone also suffices to describe programming experience.

It may seem odd that graduate students estimate better programming experience than their class mates compared to undergraduate students. We believe that graduate students feel more confident with programming than undergraduate students, because they worked with more source code. This higher confidence is reflected in the estimated programming experience compared to class mates.

7 Model of Programming Experience

In the previous section, we extracted two questions to measure programming experience in the specific setting of our experiments. The goal was to develop an easy-to-apply instrument to reliably measure programming experience. In this section, we start the development of a model that describes programming experience. We abstract from the focus of the experiments and target a general understanding of the underlying factors that influence programming experience. This way, we hope to get a better, more general understanding of programming experience and its influence as confounding parameter for program-comprehension experiments. Furthermore, we get an impression of what kind of questions are relevant to measure programming experience in other experimental settings. In the long run, we hope to ease the process of selecting questions to conveniently and reliably measure programming experience in different experimental settings, and, thus, increase the validity and comparability of experimental results.

To develop the model, we use a two-stage approach: First, we use *exploratory factor analysis* to extract a model of programming experience from the data (Section 7.1). Second, we use *confirmatory factor analysis* on a new data set to evaluate whether the extracted model is general or rather appeared randomly (Section 7.2).

The results of both factor analyses indicate that a five-factor model appears to best describe programming experience.

Table 8: Factor loadings of variables in questionnaire.

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.

7.1 Exploratory Factor Analysis

To extract a model of programming experience, we conducted an exploratory factor analysis [2]. The goal 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. However, the meaning of the factors is not a result of the analysis, but relies on interpretation.

In Table 8, we show the results of our exploratory factor analysis. The numbers in the table denote correlations or *factor loadings* of the variables in our questionnaire with identified factors. By convention, factor loadings that have an absolute value of smaller than .32 are omitted, because they are too small to be relevant [12]. 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 .32. The higher the main loading and the smaller the number 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.

The first factor of our analysis summarizes 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. We conjecture that s.Experts also loads on this factor, because it explains the confidence level with mainstream programming languages. We can name this factor *experience with mainstream languages*.

The second factor contains the variables y.ProgProf, z.Size, s.NumLanguages, and s.ClassMates. These variables fit together well, because the longer a participant is programming professionally, the more likely she has worked

with large projects and the more languages she has 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*.

Factors three and five 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* with -0.460 . 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 summarize the exploratory factor analysis, we extracted five factors: *experience with mainstream 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 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 for our specific data set. Next, we describe the confirmatory factor analysis and our experimental design to collect new data.

7.2 Confirmatory Factor Analysis

In the previous section, we extracted a five-factor model of programming experience. To evaluate whether this model holds in different data sets, not only in our specific case (i.e., to *confirm* the five-factor model), we conducted a follow-up experiment. To this end, we gave the questionnaire to over a hundred undergraduate students with a major in computer science. With this new data, we conducted a *confirmatory factor analysis*, which evaluates whether the five-factor model can explain the new data of our follow-up study. We describe the setting of our follow-up study next.

7.2.1 Experimental Design

Objective. The objective of the follow-up experiment was to evaluate whether the five-factor model is general or whether it is valid only for our specific data set (cf. Section 7.1). Thus, the research question is the following:

RQ: Can we confirm the five-factor model of programming experience?

To answer the question, we collected the data of 148 undergraduate students of different German universities, so that we have new data to evaluate the generality of the five-factor model.

Material. As material, we use our programming-experience questionnaire without any modification. To present it, we used PROPHEt. We did not need the programming tasks again, because our goal was not to relate the answers in

the questionnaire to programming experience based on performance, but to confirm the five-factor model, which is based only on the answers in the questionnaire.

Participants. As participants, we recruited once more 148 students from the universities of Passau, Duisburg-Essen, Magdeburg, and Braunschweig. For participation, students could enter a raffle for an Amazon gift card. A Kruskal-Wallis test to evaluate showed no significant difference in s.ClassMates ($\chi^2 = 0.618$, $df = 3$, $p = 0.892$). Thus, we can treat the students as one sample.

Execution. We conducted the experiment as online survey over the course of three months (summer 2012). Since during this time most students had no class⁶ and to meet time constraints, it was the best way to contact as many students as possible. To motivate students to complete the survey, we offered students to enter a raffle for gift cards, one per university. For this purpose, participants could enter their e-mail address at the end of the questionnaire. Unfortunately, that led several students to only click through the questionnaire without answering any questions, but only enter their e-mail address at the end. We ignored those responses, leading to 148 relevant participants.

7.2.2 Descriptive Statistics

To get an impression of the answers in the questionnaire, we show frequencies, medians, and dispersion in Table 9. For better comparability, we also show the descriptive statistics of the first experimental run in the two right most columns of Table 9. For most questions, participants of both experiments show a similar answer pattern. Differences appear for questions s.PE, s.ClassMates, s.C, s.Imperative. Furthermore, there are outliers for questions s.NumLanguages, y.ProgProf, e.Years, and e.Courses. If we ignore these outliers, the dispersion for the according questions is comparable with the first experiment. Thus, the answers of participants between the first run and the follow-up experiment are similar, indicating that both samples are comparable. Hence, we can continue our evaluation with the confirmatory factor analysis.

7.2.3 Results of Confirmatory Factor Analysis

In this section, we describe our confirmatory factor analysis (see, e.g., [6]). In a nutshell, it takes as input the five-factor model of programming experience and the new data set. As output, it computes the *model fit* in terms of *fit indices*, which describe how well the model can explain the correlations in the data set (i.e., how well the model *fits* to the data). In the next paragraphs, we give a detailed overview of the procedure of the confirmatory factor analysis.

First, we specify the input for the confirmatory factor analysis, starting with the five-factor model, which we visualize in Figure 4. It shows the five factors we extracted and the questions that load on each factor. This is also referred to as the *measurement model*, because it describes how each factor is measured. We specify a model upfront, because we already have a hypothesis about what factors to expect and which questions load on which factors. This is in contrast to exploratory factor analysis, in which we explore data regarding the presence of factors.

An important step in specifying the measurement model is to give the factors a scale. Since they are latent variables (i.e., they are abstract and cannot be observed directly), they have no scale by themselves. Typically, either the variances of the factors are set to 1, or the factor loadings of one

⁶ At German universities, semester starts mid October.

Table 9: Follow-up study: Overview of answers in questionnaire. The two columns on the right show the answers of the first experiment for better comparability.

No.	Question	Distribution	N	Distribution	N
1	s.PE		120		70
2	s.Experts		116		126
3	s.ClassMates		117		127
4	s.Java		120		124
5	s.C		116		127
6	s.Haskell		119		128
7	s.Prolog		118		128
8	s.NumLanguages		120		118
9	s.Functional		120		127
10	s.Imperative		118		128
11	s.Logical		118		126
12	s.ObjectOriented		120		127
13	y.Prog		106		123
14	y.ProgProf		111		127
15	e.Years		120		126
16	e.Courses		115		123
17	z.Size		120		127
18	o.Age		120		127

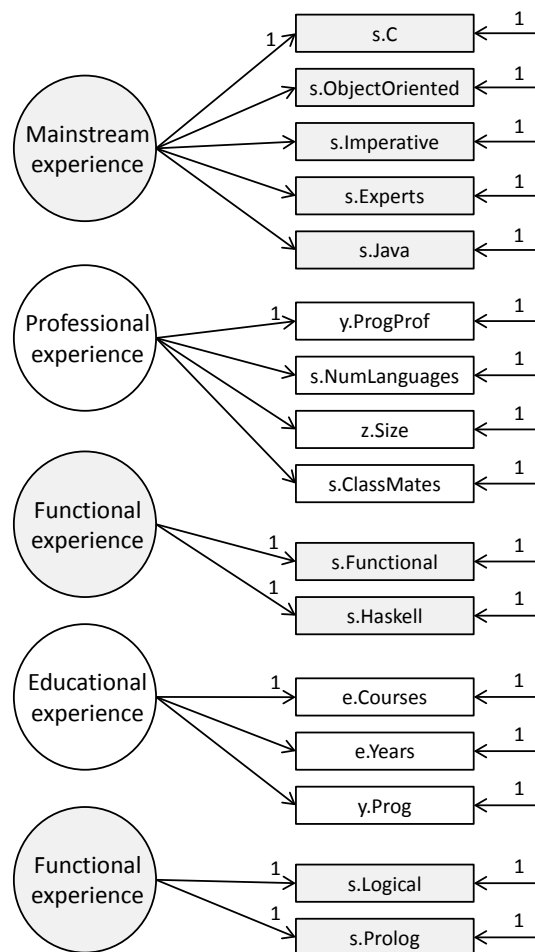


Fig. 4: Follow-up study: Measurement model for programming experience. Gray background color is used for better readability. 1 on an arrow indicates fixed loadings.

Table 10: Follow-up study: Fit indices of confirmatory factor analysis.

Fit index	Value	Threshold	Confirm?
χ^2	494.73 (< .05)	p value > .05	no
RMSEA	0.158	< .05	no
NNFI	0.538	> .95	no
SRMR	0.207	< .08	no
CFI	0.592	> .95	no

indicator per factor are set to 1. Since it does not make a difference, we set the factor loadings to 1. Furthermore, since we found in our exploratory study that s.Functional and s.Haskell, as well as s.Logical and s.Prolog, have similar factor loadings, we add the constraint that both factor loadings are equal.

Next, we have to consider the measurement error. The value we measured (i.e., the answer to a question) is not necessarily the true value, but it is biased. For example, one participant might underestimate her experience with logical programming to be modest, another participant might feel overconfident with logical programming and overestimate her experience. Thus, we add error terms to each question, symbolized by incoming arrows on each question in Figure 4. Since these errors are also latent (i.e., not directly observable, and, consequently, without scale), we also have to fix the correlation of the errors with the indicators, visualized by the 1.

After specifying the model, we have to treat missing values, because some participants did not answer all questions. We can either remove according participants from the sample or replace the missing values with neutral values, which do not change the dispersion of answers. Since we need a large sample size, we did not exclude participants, but replaced missing values with the median of according questions.

Now we can run the confirmatory factor analysis, for which we used Amos.⁷ We summarize the most important fit indices⁸ in Table 10. In column “Fit index”, we show the abbreviation of the fit index, in column “Value”, we show the according value, in column “Threshold” the threshold that an index must exceed to accept a model, and in column “Confirm?”, we show whether the model can be confirmed according to the value of the fit index.

First, the χ^2 index determines whether the data deviate significantly from the specified model. Thus, it is a significance test; if the p value is smaller than .05, model, the data deviate significantly and the model should be rejected. However, the χ^2 index is sensitive to large sample sizes and minor deviations, leading to model rejection often. Second, the *Root Mean Square Error of Approximation (RMSEA)* is insensitive to sample size, but sensitive to the number of estimated parameters in a model (i.e., the lack of parsimony) [36]. A value smaller than .05 indicates good model fit. Third, the *Non-normed Fit Index (NNFI)* also evaluates parsimony, but is sensitive to sample size; a value larger than .95 indicates a good model fit. Fourth, the *Standardized Root Mean Square Residual (SRMR)* expresses the residual terms; the smaller these terms (i.e., SRMR smaller than .08), the better the model fit. Last, the *Comparative Fit Index (CFI)* compares the fit of the measurement model and a theoretical model (specifically, the *independence model*, in which the correlations between all variables are assumed to be 0); a value larger than .95 indicates good model fit.

⁷ <http://http://www-01.ibm.com/software/analytics/spss/products/statistics/amos/>. The script of the analysis is available at the project’s website.

⁸ There is a heated ongoing debate about fit indices. Since there is not *the* best index, it is common to present different indices that focus on different aspects [5,25,26].

Table 11: Follow-up study: Mann-Whitney-U test for questions between first experiment and follow-up experiment.

Question	U value	p value	Question	U value	p value
s.Java	8647	0.322	s.PE	4226.5	0.027
s.C	7354.5	0.002	s.Experts	8696.5	0.777
s.Haskell	5832	0.000	s.ClassMates	7669.5	0.102
s.Prolog	7229.5	0.000	y.Prog	7236.5	0.007
s.NumLanguages	3266.5	0.393	y.ProgProf	627	0.668
s.Functional	6220.5	0.000	e.Years	6387.5	0.000
s.Imperative	8847.5	0.657	e.Courses	4602.5	0.000
s.Logical	7736	0.002	z.Size	757.5	0.253
s.ObjectOriented	8699	0.365	o.Age	5249	0.000

Since all fit indices indicate bad model fit, we cannot confirm the five-factor model of programming experience. Next, we discuss the results and implications of both factor analyses.

7.3 Discussion and Exploration

To summarize, the exploratory factor analysis extracted a five-factor model, which, however, the confirmatory factor analysis could not confirm. This may have three reasons: First, when conducting confirmatory factor analysis, minor deviations of data from the measurement model can be overestimated, leading to a false rejection. This is a typical problem with confirmatory factor analysis and can only be avoided in part by considering a combination of fit indices. Second, in the exploratory factor analysis, we had cross loadings, which we did not consider in the measurement model with the confirmatory factor analysis. Since we are looking for a parsimonious model, it is customary not to allow cross loadings [6]. However, we introduced further restrictions this way, namely that the cross loadings are 0, which makes it harder to fit data to a model. Last, the model of programming experience might simply not be valid, but only randomly occurred with the exploratory factor analysis.

To get a better impression of why we could not confirm the five-factor model, we took a detailed look at our data. To this end, we compared the answers of participants of the first experiment with the answers of participants from the follow-up study with a Mann-Whitney-U test. We show the results in Table 11. For several questions, the answers of participants between the first and the follow-up experiment differ, for example, in all questions that load on factor *experience from education* according to the exploratory factor analysis. We expect that especially in the case in which there are significant differences on *all* indicators of a factor (i.e., factors *experience from education*, *functional experience*, *logical experience*), according factors are problematic and may not be present in the data of the follow-up study.

We hypothesize that these significant differences, together with the constrained cross loadings, led to a rejection of the model. Further steps could include allowing cross loadings or omitting questions with a high cross loading. However, this is too close to “fishing for results” [13], and we would fit the model to our specific data set. However, we want a general model of programming experience that accounts—at least—for all bachelor students of computer science at German universities. In future work, we hope that other researchers apply our questionnaire, so that there are more data to evaluate and improve the model.

Thus, instead of fishing for results, we conducted as last step another exploratory factor analysis on the data of the second study to evaluate if we can find a similar factor pattern as for the first data set. If we can find a similar

Table 12: Follow-up study: Results of exploratory factor analysis with new data.

Variable	Main./Prev.	Prof./Prev.	Func./Prev.	Educ./Prev.	Logi./Prev.
s.C	.458 / .723	.494			
s.ObjectOriented	.835 / .700			.403	
s.Imperative	.623 / .673	.443	.333	.303	
s.Experts	.336 / .600	.326			-.328
s.Java	.801 / .540		.427		
y.ProgProf		.874 / .859			
z.Size		.608 / .764			
s.NumLanguages	.335	.890 / .489		.403	
s.ClassMates		.306 / .449	.676 / .403	.424	
s.Functional			.882 / .880		
s.Haskell			.928 / .879		
e.Courses				.631 / .795	
e.Years			-.460	.857 / .573	
y.Prog		.776 / .493			
s.Logical					.821 / .905
s.Prolog					.792 / .883

Gray cells denote main factor loadings. Prev./numbers in light gray show loadings of the first exploratory factor analysis. Main.: Experience with mainstream languages, Prof.: Professional experience, Func.: Functional experience, Educ.: Experience from education, Logi.: Logical experience.

pattern, this indicates that we are on the right track with a model of programming experience, because the probability of randomly obtaining the same or a similar model twice with different data sets is low. For this analysis, we exclude question s.PE, because we also did not consider it in the first analysis. This limits external validity, but we can better compare the results of both analyses (cf. Section 8 for a discussion).

In Table 12, we summarize the results of our second exploratory factor analysis. For better comparability, we order the table according to the results of the first exploratory analysis and also show the loadings of the previous analysis in light gray. We can see similarities to the first model. First, the factors *logical experience* and *functional experience* are present like for the first experiment with similar factor loadings, except that s.ClassMates now loads on factor *functional experience*. Thus, the higher participants estimate their experience compared to their class mates, the higher they estimate their experience with Haskell and functional programming. This seems reasonable, because functional programming is only a minor part of the curriculum, so not all students are experienced with it. Thus, a student who is experienced with functional programming is typically more experienced than her class mates. In the first exploratory factor analysis, we also found a loading on factor *functional experience*, but it was a *cross loading*. Thus, s.ClassMates does not show a completely different result from the first exploratory factor analysis. Furthermore, factor *professional experience* differs from the first exploratory analysis, in which question s.ClassMates had a main loading on this factor.

Second, we have the factor *experience with mainstream languages*. The difference to the first exploratory factor analysis is that question s.C shows only a cross loading on this factor and a main loading on factor *professional experience*. However, both loadings have a similar value. In the first analysis, there was no cross loading of question s.C. When we compare the answer to question s.C between both samples (cf. Table 11), we see a significant difference, which may have caused the difference between both exploratory factor analyses.

Last, question y.Prog has a loading on factor *professional experience*, but not on *experience from education*. Thus, the more *professional experience* students have, the longer they have been programming in general, which sounds reasonable. When looking at the first exploratory factor analysis, we found a cross loading of y.Prog on *professional experience*, so this question also shows a similar behavior (as does s.ClassMates).

In summary, we found most parts of the five-factor model of programming experience again in a new data set. Only questions s.C, s.ClassMates, and y.Prog seem to be ambiguous when considering the results of both exploratory factor analyses, but do not absolutely contradict the five-factor model. This result indicates that a five-factor model of programming experience appears valid, but we have to keep in mind that the confirmatory factor analysis rejected the five-factor model. Hence, we cannot state a final model of programming experience, but we need further investigations and data to derive a model of programming experience, which we discuss next.

7.4 Next Steps Toward a Model of Programming Experience

Specific suggestions for future work are to state models for programming experience and evaluate them with further data. For example, questions with cross loadings in both exploratory factor analyses appear to be unsuitable as indicators for programming experience and could be omitted (s.ClassMates, s.Imperative).

Of course, we could modify the model and conduct a confirmatory factor analysis with data from both experiments, but that would again be too close to fishing for results, as we could easily adjust the model such that it fits the data set. Instead, we need more data to define a model of programming experience. That may appear like an infinite endeavor, but finding a model of programming experience requires a lot of work and data and is easily a worthy topic for a complete PhD thesis.

8 Threats to Validity

8.1 Internal Validity

A first threat to internal 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, the task selection is appropriate for our purpose.

Another threat is that we did not compare self estimation with all identified ways to measure programming experience. For practical reasons, we neglected pretests and supervisor assessment in this work, because this would have required too much effort. Despite those, we considered all other identified ways. Thus, we believe we controlled this threat sufficiently.

A further threat to internal validity is that we could not control for all confounding parameters related to the person of students, for example intelligence, ability, or evaluation apprehension. However, our samples are rather large, so possible influences of such parameters are negligible. Especially gender appears critical, because women typically underestimate their experience compared to males. However, we found that for most questions, there is no difference in the self-estimation questions, but due to the large difference in the number of males and females, we need further investigations to get a better impression of the influence of gender.

Furthermore, we did not randomize the question order, but assume that most questions are orthogonal and ordering effects are minimal, but we cannot check with our design.

Additionally, we let participants finish a task after time had run out, which may bias the measured programming experience (i.e., number of correctly solved tasks in a given time frame). However, to minimize frustration of participants, we decided to not automatically abort a task after time had run out.

Another threat to internal validity is that we found almost no correlation of the response time of tasks with the answers in the questionnaire. However, there are simply too many correlations to reasonably interpret them. Furthermore, with higher task number, we have less data, because we set a time limit for all tasks, so not all participants worked on all tasks. With the current setting, we cannot exclude this threat to validity.

A threat for the confirmatory factor analysis is caused by the selection of participants. Since a considerable number of students just clicked through the questionnaire to enter the raffle for the gift card, we do not have the data of all students, but only the motivated ones. This is in contrast to our first study, in which most of the students genuinely completed the experiment and the questionnaire. However, since we conducted the follow-up study online to contact as many students as possible, we had less control over participating students. Thus, the participants might differ too much from our participants of the first run to confirm the model.

8.2 Statistical Conclusion Validity

A first threat to statistical conclusion validity is caused by the stepwise inclusion method of stepwise regression, as it is order sensitive. To reduce this threat, we validated the selected variables and their beta values with backward and forward inclusion.

A threat for the confirmatory factor analysis is caused by our sample size. Although we have 148 participants, it is not enough for a measurement model of our size. Westland suggests as lower bound to have 172 participants for a model of our complexity (i.e., depending on the number of latent variables and indicators) [39]. The sample size might have led to falsely rejecting the model instead of confirming it. With more participants, some fit indices might confirm a model of programming experience.

8.3 External Validity

The major threat to external validity for our experiment and the follow-up study is the sample: We recruited only undergraduate students. Our results can be interpreted only in the context of participants with similar experience, because our questions may have a different meaning for professionals. For example, s.ClassMates is not suitable for professional programmers, because they do not spend their time with their class mates, but with their colleagues. We could ask professional programmers to estimate their experience compared to their colleagues, but it is also not clear whether it has the same meaning as asking students to estimate their experience compared to their class mates. When applying the results to professional programmers, other indicators, such as the years of programming (professionally) may be a better indicator than self estimation. Furthermore, experience might not grow linearly past the university stage, but rather unpredictable, which may further increase the difficulty of measuring programming experience with expert programmers. However, since

most experiments are conducted with students, our results are useful for many researchers.

A further threat is that we evaluated the five-factor model of programming experience only with one additional data set (with the first data, we extracted the model). As a result, we cannot be sure how general it is. To increase external validity, we need to conduct additional confirmatory analysis with different data.

Furthermore, we cannot be sure that we included all relevant questions for programming experience. With more or different questions, the model might look different. However, since we based the development of the questionnaire on an intensive literature review, we have included the most relevant questions. Thus, we believe that we sufficiently controlled this threat. Nevertheless, one way to extend our work is to evaluate which other questions might be relevant for a model of programming experience.

Another threat is caused by omitting s.PE also in the second exploratory factor analysis. However, since we compared the data of the first experiment and follow-up study, we excluded this question. To increase external validity, we conducted another exploratory factor analysis of the new data with question s.PE, and the results show a similar factor pattern, with question s.PE loading on factor *experience with mainstream languages*. (cf. Appendix). Thus, s.PE aligns well with the five-factor model.

9 Recommendations

So far, we have 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. What have we learned in terms of recommendations for future research?

1. We showed that in literature, there are many different ways to measure and control for programming experience. Furthermore, in many cases, the methods are not reported. We recommend mixing 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.
2. We can recommend self estimation questions to judge programming experience among undergraduate students. In our experiment, several self-estimation questions correlated to 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.
3. We extracted two relevant questions, s.ClassMates and s.Logical, that can distinguish between the experience level of undergraduate and graduate students. In our case, s.ClassMates alone differentiated between undergraduate and graduate students. Thus, this one question might suffice to reliably measure programming experience in a student population. If resource constraints allow it, researchers can combine multiple questions, of which some serve as control questions to see whether participants answered honestly, which is custom in designing questionnaires [32]. For example, in our case, when using s.ClassMates, s.PE and s.ObjectOriented are suitable control questions, since they both show a strong correlation with s.ClassMates (s.PE: .632; s.ObjectOriented: .544).
4. Since correlations between questions confound the strength of a question as an indicator for programming experience (cf. Section 6), we extracted and

evaluated, based on different data, 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). Furthermore, based on factor analyses, we found that other questions are ambiguous and potentially unsuitable for measuring programming experience (e.g., `s.C`, `s.Imperative`, `y.Prog`).

5. Our exploratory and confirmatory factor analyses indicate a five-factor model of 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 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.

Overall note that while our literature review 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 education, years, and size have more predictive power. Whether self estimation remains a good indicator in this setting remains an open question for future work.

We plan to further validate the questionnaire and five-factor model of programming experience with other groups in further experiments. To this end, we will reuse the experimental design and methodology developed in this work.

10 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 [29]. 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 in order to categorize participants. These results complement ours, as we did not look into pretests and grades.

Askar and Davenport developed a questionnaire to measure self-efficacy for Java programming [4]. They recruited engineering and computer-science students to evaluate factors related to self-efficacy for Java, such as computer experience or gender. In contrast to our work, Askar and Davenport focused on explaining self-efficacy for Java programming, whereas we focus on measuring and describing programming experience.

Höst and others analyze the suitability of students as participants [24]. The authors 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. They found no differences between groups. Thus, classification of participants had no effect on their performance.

Bornat and others used a pretest to categorize good and bad novice programmers [8]. It relates to our work, in that we also aim at measuring good and bad programmers, with the difference that we seek a simple-to-apply questionnaire.

11 Conclusion and Future Work

There is a strong need to assess programming experience in an easy and cost-efficient way. Often, researchers 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.

In a controlled experiment, we evaluated the measurement of programming experience found in literature. We found that for our setting, self estimation indicates programming experience well. Specifically, we extracted two relevant questions:

1. Self-estimated programming experience compared to class mates
2. Self-estimated experience with logical programming

When comparing the answers of undergraduate and graduate students to both questions, we found a significant difference, indicating that both questions differentiate between the experience level of undergraduate and graduate students. In conjunction with control questions, such as programming experience in general or experience with Prolog, they can be used as an indicator for programming experience.

Furthermore, we started the development of a model to describe programming experience based on exploratory and confirmatory factor analyses. Results indicate that the five factors *experience with mainstream languages*, *professional experience*, *functional experience*, *educational experience* and *logical experience* seem to describe programming experience as we measured it.

To continue our work, we can collect more data for a model of programming experience, which in turn can help to derive more reliable questionnaires. Furthermore, since we already developed programming tasks, we and other researchers can use them as pretest in conjunction with different indicators for programming experience to evaluate how the identified category pretest reflects programming experience. Moreover, we can recruit programming experts and compare their performance and answers to questions to novice programmers. This way, we can get a more holistic view of programming experience.

Acknowledgments

We thank all the reviewers for their constructive feedback. We thank Jana Schumann for her support in the literature study and all experimenters for their support in setting up and conducting the experiment. Thanks to Veit Köppen for his support in the analysis of data. Siegmund's work is supported by BMBF project 01IM10002B, Kästner's work by ERC grant #203099, and Apel's work by DFG projects AP 206/2, AP 206/4, and AP 206/5.

References

1. T. Anderson and J. Finn. *The New Statistical Analysis of Data*. Springer, 1996.
2. T. Anderson and H. 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.
3. E. Arisholm, H. Gallis, T. Dybå, and D. Sjøberg. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Trans. Softw. Eng.*, 33(2):65–86, 2007.
4. P. Askar and D. Davenport. An Investigation of Factors Related to Self-Efficacy for Java Programming among Engineering Students. *The Turkish Online Journal of Educational Technology*, 8(1):26–32, 2009.
5. R. Bagozzi and Y. Yi. Specification, Evaluation, and Interpretation of Structural Equation Models. *Journal of the Academy of Marketing Science*, 40(1):8–34, 2012.
6. D. Bartholomew and I. M. Martin Knott. *Latent Variable Models and Factor Analysis: A Unified Approach*. Wiley Publishing, Inc., 2011.

7. S. Biff and W. 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.
8. R. Bornat, S. 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.
9. C. Bunse. Using Patterns for the Refinement and Translation of UML Models: A Controlled Experiment. *Empirical Softw. Eng.*, 11(2):227–267, 2006.
10. J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge Academic Press, second edition, 1988.
11. J. Cohen and P. Cohen. *Applied Multiple Regression: Correlation Analysis for the Behavioral Sciences*. Addison Wesley, second edition, 1983.
12. A. Costello and J. 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.
13. S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008.
14. H. Erdogmus, M. Morisio, and M. Torchiano. On the Effectiveness of the Test-First Approach to Programming. *IEEE Trans. Softw. Eng.*, 31(3):226–237, 2005.
15. J. Feigenspan, S. Apel, J. Liebig, and C. 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.
16. J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachsel, M. Papendieck, T. Leich, and G. Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Softw. Eng.*, 2012. DOI: 10.1007/s10664-012-9208-x.
17. J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring Programming Experience. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 73–82. IEEE CS, 2012.
18. J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachsel, V. Köppen, and M. 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.
19. J. Feigenspan and N. Siegmund. Supporting Comprehension Experiments with Human Subjects. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 244–246. IEEE CS, 2012. Tool demo.
20. E. Figueiredo, N. Cacho, M. Monteiro, U. Kulesza, R. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. 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.
21. J. Goodwin. *Research in Psychology: Methods and Design*. Wiley Publishing, Inc., second edition, 1999.
22. S. Hanenberg, S. Kleinschmager, and M. 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.
23. J. Hannay, E. Arisholm, H. Engvik, and D. Sjøberg. Effects of Personality on Pair Programming. *IEEE Trans. Softw. Eng.*, 36(1):61–80, 2010.
24. M. Höst, B. Regnell, and C. 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.
25. L. Hu and P. M. Bentler. CutoffC riteria for Fit Indexes in Covariance Structure Analysis: Conventional Criteria Versus New Alternatives. *Structural Equation Modeling*, 6(1):1–55, 1999.
26. L. Hu and P. M. Bentler. Fit Indexes in Covariance Structure Modeling: Sensitivity to Underparameterized Model Misspecification. *Psychological Methods*, 3(4):424–453, 1998.
27. A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.
28. B. Kitchenham and S. Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.
29. S. Kleinschmager and S. 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.
30. M. Lewis-Beck. *Applied Regression: An Introduction*. Sage Publications, 1980.
31. M. Müller. Are Reviews an Alternative to Pair Programming? *Empirical Softw. Eng.*, 9(4):335–351, 2004.
32. R. Peterson. *Constructing Effective Questionnaires*. Sage Publications, 2000.

33. F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. 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.
34. J. Siegmund, C. Kästner, J. Liebig, and S. Apel. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 17–24. ACM Press, 2012.
35. J. Sillito, G. Murphy, and K. De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, 2008.
36. J. H. Steiger and J. Lind. Statistically-based Tests for the Number of Common Factors. Presented at the Annual Meeting of the Psychometric Society, 1980.
37. W. Tichy. Hints for Reviewing Empirical Work in Software Engineering. *Empirical Softw. Eng.*, 5(4):309–312, 2000.
38. A. von Mayrhauser and M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.
39. C. Westland. Lower Bounds on Sample Size in Structural Equation Modeling. *Electronic Commerce Research and Applications*, 9(6):476–487, 2010.

12 Appendix

We show the results of an exploratory factor analysis, including question s.PE, which loads on factor *experience with mainstream languages*.

Table 13: Follow-up study: Factor loadings of variables in questionnaire (including s.PE).

Variable	Factor 1	Factor 2	Factor 3	Factor 4	Factor 5
s.C	.491	.462			
s.ObjectOriented	.825				
s.Imperative	.657	.402			
s.Experts	.368			-.329	
s.Java	.779				
s.PE	.726	.395			
y.ProgProf		.877			
z.Size	.310	.590			
s.NumLanguages		.884			
s.ClassMates	.682				
s.Functional			.881		
s.Haskell			.927		
e.Courses				.621	
e.Years				.860	
y.Prog	.327	.758			
s.Logical					.821
s.Prolog					.792

Gray cells denote main factor loadings.