Measuring Programming Experience

Janet Feigenspan, University of Magdeburg
Christian Kästner, Philipps University Marburg
Jörg Liebig and Sven Apel, University of Passau
Stefan Hanenberg, University of Duisburg-Essen

***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 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 128 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 factor analysis to extract a model of programming experience. With our analysis, we initiate a path toward measuring programming experience with a valid and reliable tool, so that we can control its influence on program comprehension.**

I. Introduction

In software-engineering experiments, program comprehension is frequently measured, for example, for the evaluation of programming-language constructs or software-development tools [3], [7], [13], [16], [26]. Program comprehension is an internal cognitive process that we cannot observe directly. Instead, controlled experiments are often conducted, in which we observe the behavior of subjects and draw conclusions about their program comprehension.

To conduct controlled experiments, we have to control confounding parameters, which influence the outcome of an experiment in addition to the evaluated concept [15]. One important confounding parameter is programming experience: The more experienced a subject, the better she understands a program compared to an inexperienced subject. (Accidentally) assigning more experienced subjects 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 asses 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 subjects completed a questionnaire that contained questions related to programming experience based on a literature review. Additionally, subjects solved programming tasks. Then, we compared the performance in the programming tasks with the answers in the questionnaire.

As result, we identified two questions as indicator for programming experience using stepwise regression: Self estimated programming experience compared to class mates and self estimated experience with logical programming. Furthermore, we present a five-factor model that describes programming experience using exploratory factor analysis. 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 the common questions to measure programming experience.
- Reusable experimental design to evaluate the questionnaire.
- Initial evaluation of this questionnaire with sophomores.
- Proposal toward two relevant questions and a five-factor model of programming experience.

II. 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 [20]. Due to space restrictions, we only give a short summary of the extraction process and the results. 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/Foundations on Software Engineering (FSE)*, *International Conference on Program Comprehension (ICPC),*[1] *International Symposium on Software Engineering and Measurement (ESEM),*[2] *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 subjects 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, subject, 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

[1] ICPC was a workshop until 2005 (IWPC), which we also included.
[2] ESEM first took place in 2007.

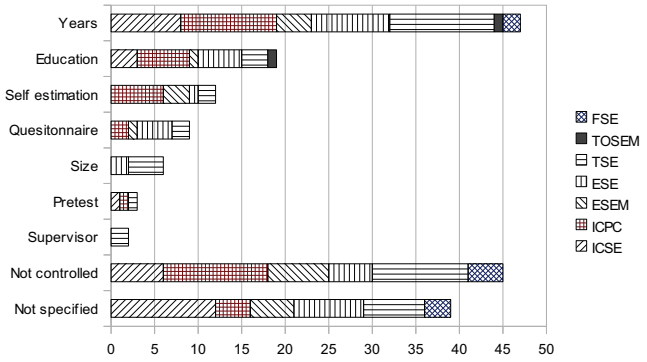


Fig. 1. Overview of how programming experience is operationalized.

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 subject 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 subject was programming professionally [26].
2) *Education:* The education of subjects 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 graduates as high-experience subjects [25].
3) *Self estimation:* In twelve experiments, subjects were asked to estimate their experience themselves. For example, Bunse let his subjects estimate their experience on a five-point scale [7].
4) *Unspecified questionnaire:* Some authors applied a questionnaire to assess programming experience. For example, Erdogmus and others let subjects fill out a questionnaire before the experiment [11]. However, it was not specified what the questionnaire looked like.
5) *Size:* The size of programs subjects had written was used as indicator in six papers. For example, Müller [23] asked how many lines of code the largest program has that subjects have implemented.
6) *Unspecified pretest:* In three experiments, a pretest was conducted to assess the subjects' programming experience. For example, Biffl and Grossmann [5] 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 experiments, in which professional programmers were recruited as subjects, the supervising manager estimated the experience of subjects [3], [17].
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 papers.

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.

III. 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 subjects (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 I, we summarize our questionnaire. We also show the scale of the answers, that is how subjects should answer the questions. In column *Abbreviation*, we show the abbreviation of each question, which we use in the remainder. The version of the questionnaire we used in our experiment is available at the project's website. Next, we explain each question in detail.

A. Years

Questions of this category mostly referred to how many years subjects were programming in general and professionally. Programming in general includes the time when subjects started programming, which includes hello-world-like

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

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

TABLE I
OVERVIEW OF QUESTIONS TO ASSESS PROGRAMMING-EXPERIENCE.

programs. Professional programming describes when subjects 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.

B. Education

This category contains questions that assess educational aspects. We asked subjects 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 subject has been enrolled). The number of courses roughly indicates how much source code subjects had implemented. With the years a subject is studying, we get an indicator of the education level: The longer a subject has been studying, the more experience she should have gained through her studies.

C. Self Estimation

In this category, we asked subjects to estimate their own experience level. With the first question, we asked subjects to estimate their programming experience on a scale from 1 to 10. We did not clarify what we mean by programming experience, but let subjects 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 subjects to estimate their experience in these more specific questions.

Next, we asked subjects to relate their programming experience to experienced programmers and their class mates to let subjects think more thoroughly about their level of experience.

Additionally, we asked subjects 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 subjects 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 subjects to list the number of programming languages in which they are experienced at least to a medium level. This way, we can assess familiarity with many languages without listing each of them. The same counts for familiarity with different programming paradigms.

D. Size

We asked subjects 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 [28].

In addition, we also included the age of subjects in the questionnaire. This way, we aim at having a more exhaustive understanding of programming experience.

IV. Empirical Validation

Constructing and validating a questionnaire is a long and tedious endeavor that requires several (replicated) experiments [24]. In this paper, we start this process.

There are different ways to validate a questionnaire. We could recruit programming experts and novices as subjects and compare their answers in the questionnaire. Since there is a difference in the experience between both groups, we should also see a difference in the questionnaire. Another way is to compare the answers in the questionnaire with performance in

tasks that are related to programming experience. The benefit is that we do not need different groups of subjects; one group is sufficient. We used the latter way with a group of students, because we found in our review that they are often recruited as subjects in software-engineering experiments. Hence, they represent an important sample. Furthermore, students can be comparable to experts under certain conditions [18], [27].

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 subjects (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 [19]. 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.

```
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.

A. 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 subjects should solve in a given time. For each task, we measure whether subjects solve a task correctly and how long they need to complete a task. The first underlying assumption is that the more experienced subjects are, the more tasks they solve correctly. Since experienced subjects have seen more source code compared to inexperienced subjects, they should have less trouble in analyzing what source code does and, hence, solve more tasks correctly. The second assumption is that experienced subjects 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.

B. Material

We designed 10 program-comprehension tasks. We gave subjects source code and asked what executing this code would print. Furthermore, subjects should 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 was result of running the program, ignoring whitespace. When an answer diverged from the expected result, a programming expert looked at subjects' explanation of the source code and decided whether the answer could be counted as correct.

To match the average experience level of undergraduates (who we recruited as subjects), we selected typical algorithms presented in introductory programming lectures. When replicating this experiment with programming experts, it might be necessary to adjust the tasks to match the high level of experience. Only the last two tasks required a higher experience level. In Task 9, we used command-line parameters, which are not typically taught at sophomore level. In the last task, we use source code of MobileMedia, a software for manipulating multi-media data on mobile devices [14]. It consists of 2,800 lines of code in 21 classes. We included the last two tasks to identify highly experienced subjects among sophomores, since some students start to program before there studies. We expected that only highly experienced subjects should be able to complete this task. All source code was in Java, the language that subjects were most familiar with.

We had 10 tasks so that only experienced subjects 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 subjects. To make sure that subjects 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 PROPHET [12]. It lets subjects enter answers, logs the time subjects spend on each task, and logs the behavior of subjects (e.g., opening files). This way, we control the influence of subjects' familiarity with an IDE. There may be other confounding parameters; our sample is large enough to control their influence.

C. Subjects

Subjects came from the University of Passau (27), Philipps University Marburg (31), and University of Magdeburg (70), so we had 128 subjects in total. All universities are located in Germany. Subjects from Passau and Marburg were in the end of their third semester and attended a course on software engineering. Subjects from Magdeburg were at the beginning of

their fourth semester and from different courses. The level of education of all subjects was comparable, because no courses took place between semesters and subject had to complete similar courses at all universities. All subjects 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 subjects participated voluntarily, were aware that they took part in an experiment, and could quit anytime. Data was logged anonymously.

Since, we recruited subjects from different universities, we actually have different samples. However, only the question s.ClassMates is specific for each university, because subjects 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 (χ^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.

D. 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 subjects complete the programming-experience questionnaire without knowing its specific purpose. Then, we gave subjects 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, subjects worked on the tasks on their own. Since we had time constraints, the time limit for the experiment was set to 40 minutes. After time ran out, subjects were allowed to finish the task they were currently working on. Two to three experimenters checked that subjects worked as planned. After the experiment, we revealed subjects the purpose of this experiment.

E. Deviation

We had a technical error for the presentation of the programming-experience questionnaire, such that we could not measure s.PE for all subjects. Hence, we only have the answer of 70 out of 128 subjects for this question.

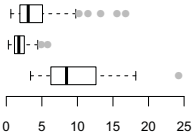
V. Experiment Results

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

A. Means and Standard Deviations

In Table II, we give an overview of how subjects solved the tasks. Column *Mean* contains the average time in minutes of subjects who completed a task. Since not all subjects 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 2000 lines of code. To solve Task 9, subjects needed on average 1.9 minutes; most likely, because its source code consisted of only 10 lines. Furthermore, only 11 subjects solved it correctly. To solve this task, subjects must be familiar with command-line parameters, which is not typical for the average sophomore. Considering the correctness of Task 4, we see that only 22 subjects 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 subjects did not analyze the insert algorithm thoroughly enough and assumed an ascending order of elements.

Variable	Response time Distribution	Mean	N	Correct
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 subjects who completed this task; Correct: number of subjects with correct solution.

TABLE II
Overview of response time for each task.

In Fig. 3, we show the number of correctly solved tasks per subject. As we expected, none of our subjects solved 9 or 10 tasks correctly. (cf. Section IV-B). Especially the last two tasks (Task 9: command-line parameters; Task 10: 2,800 lines of code) required an experience level beyond that of sophomores. 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 is rather low.

In Table III, we show the answers subjects gave in our questionnaire. The median for s.PE varies between 2 and 3, which we would expect from sophomores. In general, subjects felt very inexperienced with logical programming and experienced with object-oriented programming. The median of how long subjects are programming is 4 years, but only few subjects said they were programming for more then 10 years. Although subjects took a sophomore course, some subjects were enrolled for more than 3 years,[3] which could also explain why some subjects completed numerous courses in which they

[3]The German system allows subjects to take courses in somewhat flexible order and timing.

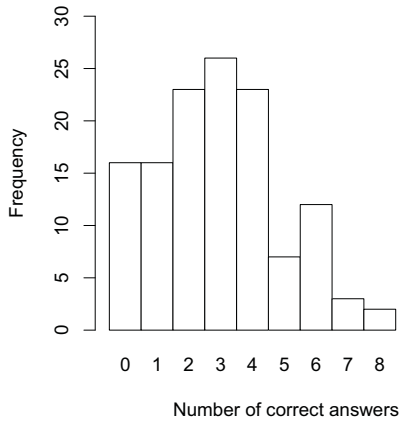


Fig. 3. Frequencies of number of correct answers.

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	o.Size		128
18	o.Age	19 40	128

TABLE III
OVERVIEW OF ANSWERS IN QUESTIONNAIRE.

had to implement source code.

B. Correlations

In Table IV, 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.[4] 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 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 V.

[4]Small: ±0.1 to ±0.3; medium: ±0.3 to ±0.5; strong: ±0.5 to ±1 [8].

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 subjects; gray cells denote significant correlations ($p < .05$).

TABLE IV
SPEARMAN CORRELATIONS OF NUMBER OF CORRECT ANSWERS WITH ANSWERS IN QUESTIONNAIRE.

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 subjects. Since we have a decreasing number of subjects with each task, we leave analyzing the response times for future experiments.

VI. EXPLORATORY ANALYSIS

In this section, we explore the data. For this analysis, we excluded question s.PE, because only 70 subjects answered this question (cf. Section IV-E). Alternatively, we could have removed subjects 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 indicator for program comprehension, but not time, since only few subjects 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. Furthermore, we would have to compute efficiency measure to deal with wrong answers [29].

For exploration, we use stepwise regression and exploratory factor analysis. Both approaches are standard in psychology, but rarely used in software-engineering research. Therefore, we start each section with an overview of the methods, before we present and interpret the results.

A. Stepwise Regression

Overview

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 ($> .30$)

No.	Question	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9	Task 10	Number of subjects
1	s.PE	-.279	-.417	-.042	.004	-.002	.016	.014	-.182	.071	.085	68 – 27
2	s.Experts	-.300	-.177	.047	-.026	.006	-.075	-.217	-.004	.206	.131	122 – 40
3	s.ClassMates	-.189	-.401	-.084	-.065	-.053	-.059	-.163	-.061	.161	.100	123 – 40
4	s.Java	.029	-.066	-.154	-.022	-.066	.003	-.040	.145	-.170	-.222	105 – 34
5	s.C	-.175	-.124	.018	.027	.126	-.108	-.056	-.052	.043	.108	123 – 40
6	s.Haskell	-.171	-.109	-.144	-.113	-.014	-.216	-.153	-.183	.019	.158	124 – 40
7	s.Prolog	-.174	-.141	-.079	-.104	-.027	-.039	.076	-.239	-.047	.146	124 – 40
8	s.NumLanguages	-.295	-.339	-.131	-.121	-.027	-.103	-.035	-.090	.232	.168	115 – 34
9	s.Functional	-.148	-.150	-.150	-.004	-.017	-.204	-.120	-.217	.027	.175	123 – 40
10	s.Imperative	-.283	.331	-.033	-.089	-.06	-.129	-.296	-.156	.126	.043	124 – 40
11	s.Logical	-.209	-.105	-.158	-.136	-.022	-.014	.058	-.257	-.191	.108	122 – 40
12	s.ObjectOriented	-.084	-.232	-.008	.012	-.093	-.034	.025	-.060	.156	.082	123 – 40
13	y.Prog	-.241	-.379	-.144	-.071	.010	-.113	-.258	-.159	.273	.180	120 – 38
14	y.ProgProf	-.217	-.196	-.012	-.119	-.130	.071	-.274	-.022	.044	-.010	123 – 39
15	e.Years	-.032	.001	.018	-.152	.059	.047	-.119	.037	-.092	-.173	122 – 40
16	e.Courses	-.146	-.088	-.040	-.062	.071	.028	-.053	-.004	.268	.058	120 – 38
17	z.Size	-.155	-.160	-.057	-.134	.059	.003	-.201	.046	.000	-.023	124 – 40
18	o.Age	.036	.014	.110	.082	.131	.102	-.081	.090	.059	.010	124 – 40

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

TABLE V
SPEARMAN CORRELATIONS OF RESPONSE TIMES FOR EACH TASK WITH ANSWERS IN QUESTIONNAIRE.

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 s.ObjectOriented with 0.552. Hence, we can assume both questions are not independent from each other. If we used both questions as indicator, we would overestimate the relationship of both questions with programming experience, that is we would count the common part of both questions twice, although we should count it only once.

To account for the correlations between questions, we use *stepwise regression* [22]. 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.Prof. 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 [9]. Thus, the correlation of y.Prog with the number of correct answers, cleaned from the influence of s.ClassMates, is computed. If this cleaned correlation is high enough, the question is included, else it is excluded. The goal is to include questions with a high 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.

Results and Interpretation

In Table VI, 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 [22]).

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

TABLE VI
RESULTING MODEL OF STEPWISE REGRESSION.

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 subjects 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 subjects and the tasks are taken from introductory programming lectures. Hence, if a subject estimates her experience better than her class mates, she should be better in solving the tasks. However, we need further research to be sure about that.

Why was s.Logical extracted and not s.Java, which is closer to our experiment? We believe that the reason is that our subjects 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 subject 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.

B. Exploratory Factor Analysis

Overview

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

Results and Interpretation

In Table VII, 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 of smaller than .32 are omitted, because they are too small to be relevant [10]. 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 then .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*.

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

Gray cells denote main factor loadings.

TABLE VII
FACTOR LOADINGS OF VARIABLES IN QUESTIONNAIRE.

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

Factor 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 subject's education. We can name this factor *experience from education*.

Now, we have to take a look at the cross loadings. As an example, we look at e.Years, which also loads on *functional experience*. This means that part of this variable can also be explained by this factor. Unfortunately, we cannot unambiguously define to which factor this variable belongs best, we can only state e.Years has a higher loading of 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.

VII. Threats to Validity

A first threat to validity is caused by the tasks. With other tasks, results may look different. However, we selected tasks representative for the experience level of undergraduates and with varying difficulty. Thus, more experienced subjects should perform better than less experienced subjects. Hence, we argue that our 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; in future work, pretest and supervisor can be compared to self estimation.

The major threat to external validity is our sample selection: We only recruited undergraduate students. Our results can be interpreted only in the context of subjects with similar experience, because our questions may have a different meanings for professionals. For example, s.ClassMates is not suitable for professional programmers, because they do not spent 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. However, since most experiments are conducted with students, our results are useful for many researchers.

Furthermore, the results of our exploratory analysis cannot be generalized without confirmatory analysis based on further experiments. If we used the same data set, we could not show that our model is valid in general, but for our specific data set. However, confirmatory analysis requires considerable effort, because we have to conduct another experiment. Furthermore, we need a large sample, depending on the complexity of the model. In our case, we would need at least 80 subjects to confirm our model (i.e., the five factors with the different questions that load on them) [4].

VIII. 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?

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

Second, we can recommend self estimation questions to judge programming experience among undergraduate students. In our experiment, several self-estimation questions correlated to a strong to medium degree (s.PE: .539; s.ClassMates: .403; s.ObjectOriented: .354) with the number of correct answers — much more than questions regarding the categories education, size, and other. Among undergraduate students, answers to questions from the latter categories differ only slightly. The only medium correlation beyond self estimation is y.Prog (.359), the number of years a subject is programming at all.

Third, if resource constraints allow it, researchers can combine multiple questions, of which some serve as control questions to see whether subjects answered honestly, which is custom in designing questionnaires [24]. For example, in our case, when using s.PE, s.ClassMates and s.ObjectOriented are suitable as control questions, since they both show a strong correlation with s.PE (s.ClassMates: .625; s.ObjectOriented: .696).

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

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

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 subjects. However, our recommendation 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 validate the questionnaire with other groups in further experiments. In this validation, we will reuse the experimental design and methodology developed in this work.

IX. Related Work

In general, related work to ours evaluated possible criteria that can be used to categorize subjects upfront. For example, Kleinschmager and Hanenberg analyzed the influence of self estimation, university grades, and pre-tests on historical data for programming experiments [21]. To this end, they analyzed the data of two previously conducted programming experiments with students as subjects. They compared self estimation, university grades, and pretests with the performance

of subjects in the experiments and found that self estimation was not worse than university grades or pre-tests in order to categorize subjects. These results complement ours, as we did not look into pretests and grades.

Höst and others analyze the suitability of students as subjects [18]. 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 subjects had no effect on their performance.

Bornat and others used a pre-test to categorize god and bad novice programmers [6]. 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.

X. Conclusion

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 within groups of undergraduate students, self estimation indicates programming experience well. Specifically, we extracted programming experience compared to class mates and experience with logical programming as relevant questions. In conjunction with control questions, such programming experience in general or experience with Prolog, they can be used as indicator for programming experience. Furthermore, we extracted a five-factor model with the factors *experience with mainstream languages*, *professional experience*, *functional experience*, *experience from education*, *logical experience*. The next step are confirmatory analyses, in which we aim at confirming the results of our experiment with different groups of subjects or different tasks or different ways to measure programming experience. To this end, we and other research groups can reuse our experimental design.

Acknowledgments

Feigenspan'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 LE 912/13. We thank Jana Schumann for her support in the literature study and all experimenters for their support in setting up and conducting the experiment.

References

[1] T. Anderson and J. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[2] T. W. 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. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Trans. Softw. Eng.*, 33(2):65–86, 2007.

[4] P. Bentler. Practical Issues in Structural Modeling. *Sociological Methods Research*, 16(1):78–117, 1987.

[5] S. Biffl 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.

[6] 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.

[7] C. Bunse. Using Patterns for the Refinement and Translation of UML Models: A Controlled Experiment. *Empirical Softw. Eng.*, 11(2):227–267, 2006.

[8] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge Academic Press, second edition, 1988.

[9] J. Cohen and P. Cohen. *Applied Multiple Regression: Correlation Analysis for the Behavioral Sciences*. Addison Wesley, second edition, 1983.

[10] A. B. Costello and J. W. 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.

[11] 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.

[12] J. Feigenspan et al. PROPHET: Tool Infrastructure To Support Program Comprehension Experiments. Int'l Symposium Empirical Software Engineering and Measurement (ESEM), Poster.

[13] J. Feigenspan et al. 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.

[14] E. Figueiredo et al. 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.

[15] C. Goodwin. *Research in Psychology: Methods and Design*. Wiley Publishing, Inc., second edition, 1999.

[16] 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.

[17] J. Hannay et al. Effects of Personality on Pair Programming. *IEEE Trans. Softw. Eng.*, 36(1):61–80, 2010.

[18] 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.

[19] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.

[20] B. Kitchenham and S. Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, 2007.

[21] 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.

[22] M. Lewis-Beck. *Applied Regression: An Introduction*. Sage Pubications, 1980.

[23] M. Müller. Are Reviews an Alternative to Pair Programming? *Empirical Softw. Eng.*, 9(4):335–351, 2004.

[24] R. Peterson. *Constructing Effective Questionnaires*. Sage Publications, 2000.

[25] F. Ricca et al. 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.

[26] J. Sillito, G. C. Murphy, and K. De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, 2008.

[27] W. Tichy. Hints for Reviewing Empirical Work in Software Engineering. *Empirical Softw. Eng.*, 5(4):309–312, 2000.

[28] A. von Mayrhauser and M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

[29] J. Yellott. Correction for Fast Guessing and the Speed Accuracy Tradeoff in Choice Reaction Time. *Journal of Mathematical Psychology*, 8:159–199, 1971.