Using Students as Experimental Subjects in Software Engineering Research — A Review and Discussion of the Evidence

Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel

Abstract

Should students be used as experimental subjects in software engineering? Given that students are in many cases readily available and cheap it is no surprise that the vast majority of controlled experiments in software engineering use them. But they can be argued to constitute a convenience sample that may not represent the target population (typically "real" developers), especially in terms of experience and proficiency. This causes many researchers (and reviewers) to have reservations about the external validity of student-based experiments, and claim that students should not be used. Based on an extensive review of published works that have compared students to professionals, we find that picking on "students" is counterproductive for two main reasons. First, classifying experimental subjects by their status is merely a proxy for more important and meaningful classifications, such as classifying them according to their abilities, and effort should be invested in defining and using these more meaningful classifications. Second, in many cases using students is perfectly reasonable, and student subjects can be used to obtain reliable results and further the research goals. In particular, this appears to be the case when the study involves basic programming and comprehension skills, when tools or methodologies that do not require an extensive learning curve are being compared, and in the initial formative stages of large industrial research initiatives — in other words, in many of the cases that are suitable for controlled experiments of limited scope.

keywords: Controlled experiment; Experimental subjects; Students; Experimental methodology.

1 Introduction

Experimentation, and especially experiments involving human subjects, are not widely used in computer science [180, 80]. But two sub-fields are an exception: human-computer interaction and software engineering. In these disciplines the recognition that humans are inherently "in the

loop" has led to significant experimental work involving humans. In particular, in software engineering controlled experiments in the lab are used to learn about the effectiveness of different methodologies, procedures, and tools [16, 18].

However, experimenting with humans is difficult. One of the salient problems is finding subjects who are representative of the target population. While a specific target population is often left undefined, in software engineering it is typically understood to be the community of software developers, and especially those who do it for a living. But recruiting professional developers for experiments is hard, because they need to be paid a competitive fee [14], and there may be additional constraints such as not interfering with normal production work [69].

An intriguing alternative is to use computer science or software engineering students, who are naturally much more accessible in an academic setting, and in addition may be argued to constitute the next generation of software professionals [109, 179]. Experiments can sometimes even be done as part of class requirements (e.g. [102, 40, 7, 185]). And indeed, a literature study of the period 1993–2002 found that in 87% of the experiments reported students were used as subjects [170]. But this leads to the question of whether the results of student-based experiments have external validity and generalize to real-life settings. A similar situation occurs in psychology, where a majority of studies are based on students who are WEIRD (Western, Educated, Industrialized, Rich, and Democratic) and may not reflect general human behavior [92].

The problem of what experimental subjects to use has been recognized long ago. In one of the first papers to report on a controlled experiment in software engineering, Myers writes concerning "the question of whether one can extrapolate, to a typical industrial environment, experimental results obtained from trainee programmers or programmers with only a few years of experience" [136]. Some time later, Curtis provocatively entitled a review of empirical software engineering "By the way, did anyone study any real programmers?" [61]. More recently, Di Penta et al. write in the context of advising on experimental procedures that "a subject group made up entirely of students might not adequately represent the intended user population" [68], and Ko et al., in the context of discussing experimentation for evaluating software engineering tools, write that "undergraduates, and many graduate students, are typically too inexperienced to be representative of a tool's intended user" [112]. Jedlitschka and Pfahl, in suggested reporting guidelines for empirical research, use "Generalization of results is limited due to the fact that undergraduate students participated in the study" as an example of a "limitations" clause that should be included in structured abstracts [98]. Finally, in a literature survey focused on confounding factors in program comprehension studies, the subjects' level of experience was the most often cited factor [163]. This has led some researchers to actually hire professional programmers to participate in extended experimental evaluations of various techniques and tools (e.g. [168, 8, 73, 30, 171, 191]). But is this expenditure really necessary? And under what conditions?

Conversely, the obsession with "students" being a problem can be criticized as reflecting a simplistic belief that graduation is a pivotal event of great significance. A more reasonable view is that there exists a wide range of capabilities in both students and professionals, with large overlap, but perhaps also focused improvements due to on-the-job training. Tichy, for example, in a list of suggestions to potential reviewers of empirical software engineering research, writes that "don't dismiss a paper merely for using students as subjects" [179]. Among the reasons he cites are the

observation that even if this is not the optimal setting much can still be learned from student studies, and that students are in fact quite close to the target population of developers [109]. Likewise, Soh et al. discuss the classification of experimental subjects based on professional status (students vs. professionals) as opposed to a classification based on expertise, and conclude that expertise does not necessarily correlate with professional status [172]. Consequently excluding students just because they are students would be detrimental both to research and to staffing decisions. And a literature study from 2000–2010 found that large numbers of professional developers are in fact not needed for effective user evaluations, and moderate numbers of students can also be used [45]. More generally, Hannay and Jørgensen explain how absolute realism is not always the best approach in experimental research, and that artificial constructs may be preferable in certain cases [90]. Finally, one should note that the goal of software engineering experiments using human subjects is usually the evaluation of tools, procedures, or methodologies. Such evaluations are typically relative, trying to assess whether one approach is better than another. Therefore the requirement that subjects be representative may be an overstatement: it is actually enough that the relative results be representative.

Our goal in this review is to discuss the basic question:

Should students be used as experimental subjects in software engineering research, and if so, under what conditions?

To do so we organize and present the cumulative experience with issues related to using students in software engineering research, as it is reflected in the scientific literature. This is structured according to the following sub-questions:

- *Why are students perceived to be a problem?* Section 2 presents a list of the potential problems that may arise when using students as subjects, such as the differences between students and professionals in tool use and experience.
- *What is the evidence?* Section 3 reviews studies that contrast students and professionals head-on. It reports evidence which shows both that the distinction is not important, so students can indeed be used in lieu of professionals, but also evidence that experience actually does matter, and thus that student novices and professional veterans may have widely different skills and require (or at least benefit from) different tools and procedures.
- *What does it all mean?* Given the wide diversity of results in the literature, Section 4 provides a discussion of their implications, couching it in the more general treatment of experience and expertise. The main conclusion is that the focus on the dichotomy between students and professionals is a dangerous over-simplification. First, this is the wrong distinction, and it is much more important to classify experimental subjects according to their abilities and suitability to the specific study at hand. At the same time, one must realize that there is an extremely wide spectrum of software developers, so there is no single "right" way to select experimental subjects who will be generally representative.
- *How should students be used?* Finally, Section 5 lists some recommendations concerning the use of students in empirical studies, in case you decide to do so.

	students	*professionals*
toy problem	academic experiment	industrial experiment
real problem	project/internship	real-life situation

Figure 1: *Differences between controlled studies and real-life situations.*

There have been many papers that purport to address the question of using students, but in the end this is always done in the context of a single experimental setup. This review attempts to pool all these previous studies, which are reviewed in detail in Section 3.

It should be noted that experimental subjects (and in particular, using students) are not the only potential problem with controlled experiments in software engineering. Actually, there are two main contentious dimensions in such experiments: the experimental subjects and the tasks being tackled by these subjects (Figure 1) [42, 169, 175, 52]. In many cases the tasks are just toy problems in order to make running the experiments manageable. Such toy problems cannot coax out deep knowledge on how real experts approach complex situations. Moreover, the environment is also important. Perry et al. point out that professional programmers spend only about half of their time coding, so studies of various methodologies and tools conducted under sterile laboratory conditions may not represent what happens in real life situations [138]. It therefore seems that there are significant issues that are simply out of the reach of controlled laboratory experiments. While this is mentioned in the discussion, most of the review focuses on the issue of subjects in controlled experiments, as it is already wide enough.

We note too that using students is but one aspect of potential threats to external validity. And the whole issue of the relative importance of external validity as opposed to internal validity is also hotly debated [164]. Of course, if one believes that internal validity trumps external validity the concerns about using students are moot. But given that many researchers believe that external validity is more important, to the point of expressing complete disbelief in academic studies based on students, a review of this topic is believed to be useful.

The methodology followed is a classical literature review. This is not a "systematic review" [107] as the goal is not to amass evidence for or against a certain effect. Rather, the goal is to collect all the differing considerations and opinions that have been expressed on this topic. To do this we started with several sources, including keyword searches such as "students" and "experimental subjects" and snowballing (following the references in papers that had already been read). This enabled the identification of other important publications across more venues and years than would be possible to scan systematically, as well as the inclusion of literature from related fields, such as general studies of experts vs. novices. The price paid is that others who perform a similar review might end up with a somewhat different list of papers. The following sections present what we learned from the papers that were found.

2 Problems with Students

The main concern regarding using students as experimental subjects is that they may not represent professionals faithfully. Thus if we are interested in software engineering as it is practiced by professionals in the field, data obtained from student experiments may be misleading.

There are several reasons why professional experience is important. One is that experience can be expected to hone the knowledge acquired during studies, making it more directly usable in performing software engineering tasks and eradicating misconceptions. Another is that experience leads to both improved and new capabilities, even changing the way professionals approach a task and not only the proficiency with which they attack it [124, 155]. In addition, students (and experiments based on students) may also suffer from the characteristics of the academic setting. In this section we review studies that have demonstrated and discussed such effects.

2.1 Learning Misconceptions

As noted above the concern about using students in empirical software engineering research is that they "lack experience". One aspect of this is that they may not have ingested all that they have learned, or worse, have not learned something at all or have learned it wrong [88].

A major issue that has been identified in the literature is that novice programmers may hold invalid or inappropriate mental models of what programming constructs actually mean. For example, Bayman and Mayer studied how well 30 undergraduate college students understood nine basic BASIC statements after a 6-hour self-study course widely used in the microcomputer lab [21]. The results showed that between 3% and 80% (median 27%) achieved a fully correct conceptual model of the statements, with up to 56% having incomplete models and up to 60% having incorrect models. Importantly, all subjects successfully completed the course, indicating that the used tests of how well students master the material were ineffective in discerning these problems. Ebrahimi conducted a similar study, classifying language construct misunderstanding by novice programmers learning Pascal, C, Fortran, and Lisp [74]. In addition, he also noted errors in composing these constructs into problem solution plans. And Kaczmarczyk et al. found basic misconceptions regarding the relationship between language elements and underlying memory use, how while loops operate, and the object concept [104].

Similar results have been obtained also when studying more advanced concepts. For example, Kahney performed a study of how novice programmers differ from experienced ones in their understanding of recursion [105]. The results from 30 novice students and 9 more experienced ones (TAs) was that over half of the novices appear to have thought of recursion as a loop, and only 3 really understood it, as opposed to 8 of the more experienced ones.

More recently, Ma et al. looked at how students understood reference assignment in Java at the end of their first year [127]. They found that only 17% had a consistently viable model of what is going on. The remaining 83% were divided evenly between those who were consistently wrong and those who were inconsistent, both groups displaying a wide variety of inappropriate models. In this case a strong correlation was found between the viability of the model held by the students and their performance in the final exam. Nevertheless, many students with non-viable models

also managed to perform well in the finals, and some even held non-viable models of the simpler value assignment statement. Likewise, Madison and Gifford found that two students managed to construct correct programs and answer questions despite having a misconceived notion of how parameters are passed to functions [128].

At a more general level, novice students may not internalize correctly what they have learned. As a result they may misapply it. For example, Jeffries et al. report on students who used large arrays instead of linked lists to store page numbers of an index, and one who decided to store the text to be indexed in a binary tree, thus losing all its structural information, instead of storing the desired index terms in a tree [99].

The conclusion from studies such as these can be summarized as follows:

> Novice programmers, especially beginning undergraduate students, may be ill-equipped for participation in empirical software engineering studies in fields such as program comprehension or design. At the very least, more advanced students (e.g. from the third year of study or beyond) should be preferred.

2.2 Differences in Technology Use

Another difference between students and more experienced developers is how they use available technology and tools. Several studies have documented such effects, and interestingly, they work in both directions.

Inexperienced users may not use available technology that could help them to perform a task, either because they don't know about the tools or they do not feel sure enough to use them. As an example, inexperienced students faced with the task of comprehending Java code that had been obfuscated by identifier renaming did not use renaming facilities to change the obfuscated names back to meaningful names once they figured out what they represented; instead they continued to work with the original obfuscated code [55]. Thus they did not use an available tool that could help them.

On the other hand, students also stand to gain more from using tools, because such tools can make up for the shortcomings in their experience. For example, Ricca et al. demonstrated that inexperienced students benefited more from using web-application-related annotations on UML diagrams than more experienced subjects [144]: the experienced ones could understand the diagrams to a large degree even without the added annotations, but for inexperienced students the added annotations proved invaluable. In other words, the tool — in this case, the added annotations on the UML diagrams — helped the inexperienced users to close the gap between them and the more experienced users.

But this can also work the other way around. Tool usage takes training, and the learning curve itself may depend on previous knowledge. Thus advanced tools may be accessible only to advanced users, and by using them the advanced users *widen* the gap between them and the inexperienced users. An example is provided by Abrahão et al. [1]. The context was again the use of UML diagrams, in this case sequence diagrams used to portray and understand system dynamics. Experiments showed that using these diagrams helped experienced and proficient users

comprehend the modeled functional requirements, but had no significant effect for inexperienced users. Conversely, students may actually be better trained in tool usage than professionals who have not been formally taught the basics [7].

Briand et al. found that adding formal OCL (Object Constraint Language) annotations to UML diagrams helped students perform comprehension and maintenance tasks, but only provided ample training was provided first [40]. Such training was also needed for the effective use of a graphical representation of requirements [157]. Another example was described by Basili et al. [17]. In reviewing five replications of experiments on the effectiveness of defect-based code reading (DBR) as a means to uncover faults, three studies based on undergraduates showed no evidence for increased effectiveness, while two based on graduate students and practitioners showed evidence for a positive effect. This was attributed to the fact that DBR is based on expressing requirements in a state-machine notation, which may be too sophisticated for undergraduates.

An especially interesting experiment shows that differences in tool usage may represent differences in strategy. Bednarik and Tukiainen used eye tracking to see how study participants performed program comprehension tasks, based on code reading and animations of the program execution using the Jeliot 3 visualization tool [24]. The result was that some participants first read the code to form a hypothesis of what it does, and then used one or a few animations to verify this. These participants turned out to be the more experienced ones, with 73.5 ± 69.9 months of programming experience. Other participants, on the other hand, approached the comprehension task by running animations to see what the programs do. It turned out that these participants could easily be characterized as less experienced, having only 18.0 ± 19.4 months of experience.

The conclusion from these examples is

> Study subjects should be selected to fit the task. Studies involving sophisticated tasks, which require mastery of specific tools or methodologies, should not employ student subjects who do not have the necessary skills. At the very least, appropriate training should be provided, and the level of proficiency should be assessed.

We note in passing that experience may not only affect tool usage, but may also inform tool design. In a series of studies LaToza and Myers have surveyed professional developers, and recorded the most vexing problems that troubled them during their work [118, 117]. They then used this to design tools that specifically target these problems. Importantly, the uncovered issues were hard to anticipate, as they reflected problems and work practices of professionals in a real-life setting. For example, the developers often faced problems such as understanding why a certain piece of code was implemented in a certain way, what code does under certain conditions, why it was changed, and so on. Students typically do not face such questions, and can not be expected to identify them as important.

2.3 Lack of Experience

Students who are new to the field by definition lack experience, meaning that they have not done much yet. The interactions of experience with tool usage cited above are not the only effect of

experience. Experience has been shown to have an effect on myriad other software engineering activities as well.

At a rather basic level, experience may affect how one performs even a simple task. For example, Burkhardt et al. used a documentation task to track the mental models of expert and novice programmers [44]. One of the findings was that experts put three times as many comments in header (.h) files as in code (.cc) files; for novices it was exactly the other way around. Moreover, within code files experts emphasized comments on functions as opposed to inline comments (51% vs. 29%, respectively), whereas for novices the inline comments outnumbered the function comments by a 2:1 ratio. It can be conjectured that such behavioral differences stem from experience in trying to understand others' code, and the realization that general explanations and interfaces are more important than code minutiae. In a related vein, several studies have found that students place more emphasis on comments than professionals do [130, 154].

Several additional recent experiments have demonstrated situations in which experience has a considerable effect on outcome. Binkley et al. conducted a few studies of the effect of multi-word identifier styles (that is, using camelCase or under_scores) on their correct and speedy recognition. Initially they found that in general it took longer to identify camelCase and it was faster to identify under_scores [33]. However, computer science training led to a reduction in the difference: more training led to reduced time with camelCase and more time for under_scores. All the subjects were undergraduate students, and the level of training reflected years of studying computer science; importantly, they also included non-computer-science majors as a control group.

These results were largely replicated by Sharif and Maletic [159], using eye-tracking equipment and subjects trained mainly in the under_score style. The subjects were both undergraduate and graduate students and two faculty members. In another experiment, this time measuring the ability to recall identifiers, beginners did badly when the identifiers had under_scores [32]. Thus the experts did better than beginners with under_scores, and beginners did better with camelCase than with under_scores. But these studies also showed many other effects, that in some cases were much more significant than the effect of experience.

Considering more complicated tasks, a study of 5 professional developers by Adelson and Soloway showed that experienced developers may use different approaches depending on whether they have previous domain experience that is specifically relevant to the problem at hand [4]. Novices don't have such resources available to them, and need to build up their knowledge of the domain from scratch. But gaining experience may be quick. Falkner et al. document how students change their software development strategies from their first year to the end of their degree [78].

It is not surprising that experienced developers perform better than novice programmers on various programming-related tasks. But what is the root cause of this advantage? Soloway and Ehrlich suggested that expert advantage stems from two reasons: they know solution patterns, and they follow and exploit programming conventions [173]. Experiments to demonstrate this were conducted with 145 students, with a 2:1 ratio of novices to advanced ones (in their case, students with at least 3 programming courses under their belt), using tasks that were meant to specifically depend on these two traits. The results were that indeed the advanced students performed better. But when program fragments did not conform to the patterns and conventions, the performance

of the advanced programmers was reduced to near that of novices. More results on the internal representations used by novices and experts are described below in Section 3.

In summary, the above results support and extend those of the previous section:

> Experience may certainly contribute to capability and change how problems are tackled. Therefore experimental subjects should have a suitable level of experience to undertake the tasks being asked of them.

2.4 Academic Overqualification

As mentioned already one of the justifications for using students in empirical software engineering research is the perception that they are representative of the next generation of software engineers [109, 179]. his notion is challenged by the Stackoverflow 2015 Developer Survey[1]. Slightly more than 20,000 developers worldwide answered the survey. Among them, 37.7% had a BSc in computer science or a related field, 18.4% had an MSc, and only 2.2% had a PhD. These numbers are rather similar to those reported in a survey of developers using Microsoft technology reported in VisualStudio Magazine in 2013[2]; The numbers there were 33.9% who graduated from a 4-year college, 28.0% with an MSc, and 3.0% with a PhD. However, an additional 10.6% graduated from a 2-year college, and an additional 8.1% took some post-graduate studies without obtaining a degree.

Returning to the Stackoverflow survey, 48% reported that they did not have a university degree in computer science, and 33% reported that they did not attend even one university computer science course. In fact, 41.8% claimed to be self taught. Thus computer science students (and the university education they receive) represent at best about half of developers in the field, and graduate students are especially non-representative. In other words, the notion that students are at least representative of the next generation of software engineers (if not of the current generation) may be optimistic.

A different view on this issue is presented by Lethbridge, who conducted a survey of computer science and software engineering graduates in 1998 [120]. He found that respondents generally considered their education to be moderately relevant to their work as software developers, but with a wide range of opinions, including some who thought it was completely irrelevant. More specifically, of those who studied computer science or software engineering 70% considered their education relevant, but of those who had studied computer or electrical engineering only 30% found it relevant. More interestingly, respondents reported having learned significantly more mathematics than software, but software outranked mathematics by about the same margin for usefulness, current knowledge, and desire to learn more. This implies that academic studies are not perfectly aligned with the needs of industry professionals. Thus,

> A little-appreciated problem with students is that their perspective is academically oriented, and perhaps not aligned with industry practice.

[1] http://stackoverflow.com/research/developer-survey-2015, visited 29 Jul 2015.

[2] https://visualstudiomagazine.com/articles/salary-surveys/salary-survey.aspx, visited 29 Jul 2015.

2.5 Contextual Effects

The previous sections dealt with technical aspects of how students may perform in software engineering experiments, and how this may differ from the performance of professionals. But there are also concerns that stem from the fact that students are just students and that the experiments are taking place in an academic setting.

One possible problem is uniformity and lack of perspective. In an academic setting, it is highly probable that all the student subjects had learned the relevant material from the same professor at the same time. Furthermore, they most probably did not accrue any on-job experience that could challenge what they had learned and confront it with day-to-day situations. Thus their collective perspective on the issue being studied may be limited.

Concern has also been expressed about the effect of the relationship between students and (faculty) experimenters [112]. This may take either of two forms: the students may anticipate the desired outcome and unconsciously favor it, or they may sense their professor's intent and unconsciously try to support it. While such effects have apparently not been reported in the context of software engineering, they are known from other domains. Another potential danger occurs when students are required to participate in an experiment and are not allowed to withdraw at will. In such a situation there is a risk of contaminating results by disgruntled subjects [167]. But this is not necessarily limited to students, and may also happen when professionals are drafted to participate in a study by their boss.

Another concern is that students may be less committed than professionals [95, 27]. This may be especially significant if the experiments are done as part of a class, as in this setting students (at least the smart ones) may be expected to invest effort only to the degree that it contributes to their grades. Volunteer students or those participating in a project may therefore make better experimental subjects. But on the other hand, it is not clear that professional participants are necessarily more committed, especially if they are not paid specifically to participate in the experiment. For example, one study describes a situation where over 100 employees in a firm agreed to participate in a study, only 60 eventually submitted their results, and of those only 33 were really usable [54]. Another study also noted lack of cooperation as a problem, and specifically cited the inclination of industry participants not to follow experimental guidelines, sometimes voiding the results simply because they did not actually use the prescribed methodology that was being examined [183].

> Conducting experiments in an academic setting may be suspect. However, there are no specific reports of such problems in software engineering research, and it is not clear that an industrial setting is better.

3 Direct Comparisons of Students and Professionals

Many of the experiments described in the previous section considered experience as a confounding factor and not as a main effect. But because of concerns about using students as subjects, some papers have also addressed the issue of comparing students to professionals explicitly. Such papers are listed in Table 1, and those that have not been mentioned yet are discussed further next. In this

we expand the discussion to include not only studies pitting students against professionals, but more generally, any study concerned with programmers with different levels of expertise. This can include comparing the performance of novice students with more advanced ones (e.g. [55, 144, 1]), or novice professionals with more experienced ones. It can also involve a sort of meta-study, where a study with one group of subjects is later followed by another study using another group, and the results are compared (e.g. [41, 54, 55]).

3.1 Studies of Cognitive Processes

In the 1980s there was some interest in the field of psychology in understanding the differences between expert and novice programmers. These studies were focused on cognitive processes and internal representations. The common theme was that experts employ deeper knowledge and semantics, while novices tend to use surface attributes and syntax [160]. For example, Weiser and Shertz asked programming novices and experts (undergraduates and graduate students) to classify 27 problems. The novices tended to do so based on application area, whereas the experts tended to classify based on the algorithm that would be used to solve the problem [187].

McKeithen et al. demonstrated that expert programmers are better able to recall semantically meaningful program code, and that their programming knowledge is better organized to reflect programming concepts [133]. To quantify this they asked subjects to recall 21 ALGOL keywords, each time prompting them to start with a different one and continue with others "that go with it". They then analyzed subsequences that tended to appear in the same order. Wiedenbeck showed that experts are both faster and more accurate on simple programming related tasks (e.g. to identify syntactically incorrect lines), implying that experience leads to automation that replaces conscious cognitive effort [188]. These results hark back to the work of Simon on chess masters, who were shown to possess a large "vocabulary" of chess positions [166].

Adelson compared novice programmers with more advanced ones (graduates of an introductory programming course and TAs in that course), using materials and questions at two different cognitive levels [3]. The abstract level represented a program using a flow-chart with high-level blocks, and the corresponding questions were also at this level (e.g. what is the shape of a matrix being used). The concrete level represented the program using a detailed flow-chart of individual operations, and the questions were about details (e.g. which border of the matrix is handled first). The results were that novices tend to operate at the concrete level and experts at the abstract level — so much so, that in certain cases presenting a concrete question regarding an abstractly-represented program caused experts difficulties, and allowed novices to out-perform them.

In a related vein, Burkhardt et al. studied the internal representations of novices and experts for object-oriented programs [43]. The experts were 30 professional programmers, and the novices were 21 advanced students who were relatively new to object technology. Using documentation and code reuse tasks, they found that the experts had a better mental model of the objects that constitute the program and the relationships between them. However, novices could improve their representation if the task demanded it [44].

Table 1: *Papers comparing experimental subjects with different levels of education or experience. L1, L2, and L3 are the levels used in the experiments. Numbers in parentheses are the number of subjects at each level. "stud." is students of unspecified degree, and ↑ indicates students near the end of their degree. "ind." or "prof." means professionals from industry. If degrees and affiliation are not mentioned assignment was made by some assessment of proficiency and classification as "beginner", "novice", "intermediate", "advanced", "experienced", etc.*

	Subjects				
Ref.	*L1*	*L2*	*L3*	*Study*	*Results*
Sackman [151] 1968	trainee (9)	exp. (12)	–	verify the effect of an interactive environment on debugging	for both groups the interactive setting was beneficial, and for both individual differences were high
Jeffries [99] 1981	nov. BSc (5)	exp. (4)	–	compare solution strategies to book indexing problem	all subjects used the same decomposition strategy, but novices were less effective in looking for subproblem solutions and knowledge representation
McKeithen [133] 1981	nov. stud. (29)	int. stud. (29)	exp. (8)	identify how programming knowledge is organized depending on skill	more advanced subjects' knowledge was better organized to reflect programming concepts
Weiser [187] 1983	nov. (6)	exp. (9)	mgr. (4)	classify problems by domains, algorithms, or data structures	novices tended to classify by domain, experts by algorithm, and managers were different from both
Adelson [3] 1984	nov. stud. (42)	TAs (42)	–	identifying the cognitive level at which novices and experts work	novices use concrete representations and experts use abstract representations
Soloway [173] 1984	nov. stud. (94)	adv. stud. (45)	ind. (41)	characterizing the knowledge of advanced programmers	advanced programmers know and use design patterns and programming conventions
Wiedenbeck [188] 1985	nov. (20)	exp. (20)	–	test performance on trivial programming-related tasks	experts are faster and more accurate, implying automation

(Continued on next page)

Table 1: Continued.

Ref.	Subjects *L1*	*L2*	*L3*	*Study*	*Results*
Gugerty [87] 1986	nov. (10)	exp. (10)	–	finding a bug in a program	skilled programmers are faster and better
Basili [15] 1987	stud. (42)	ind. (32)	–	compare software testing and inspections	code reading worked better for professionals
Porter [140, 139] 1998	grad. (48)	ind. (18)	–	systematic strategies for requirements inspections	comparisons of approaches gave the same results despite differences in absolute performance
Schenk [155] 1998	nov. (7)	exp. low (9)	exp. high (9)	detailed study of how system analysts do requirements analysis	experts approach requirements analysis differently from novices
Höst [95] 2000	MSc↑ (25)	ind. (17)	–	subjective assessment of project delay factors	similar results for both groups
Briand [41] 2001	stud.	ind.	–	use defect data to evaluate OO quality metrics	results from student projects largely confirmed by industrial followup
Crosby [59] 2002	nov. (9)	adv. (10)	–	how experts and novices work to comprehend code	experts focus more on the complex parts
Burkhardt [43, 44] 2002	stud. (21)	ind. (30)	–	mental models of object-oriented programs	complex interactions between factors, including some effect of expertise
Runeson [150] 2003	BSc↓ (31)	grad. (131)	–	improvement due to PSP training	freshmen and graduate students showed similar improvement, but graduates were faster
Berander [27] 2004	MSc↑ (20) PhD (15)	proj. (16) ind.	–	requirements prioritization	classroom students divide requirements more evenly between priorities; those in industrial projects make most requirements high priority
Arisholm [9] 2004	stud. jun. (90)	int. sen. (68)	–	maintenance of either a (bad) centralized or (better) delegated OO design	inexperienced subjects had a hard time with the delegated style, students did better on the centralized design

(Continued on next page)

Table 1: Continued.

	Subjects				
Ref.	*L1*	*L2*	*L3*	*Study*	*Results*
Bednarik [23] 2005	nov. (8)	int. (8)	–	gaze analysis when using animations for comprehension	advanced participants exhibited same behavior but were faster
Lange [116] 2006	stud. (111)	ind. (48)	–	effect of defects in UML diagrams on their use	similar effect on both students and professionals
Arisholm [8] 2007	jun. (81)	int. (102)	sen. (112)	compare individual programming with pair programming	pairs of novices got better results on a complex system; experienced pairs were slightly faster on a simple system
Bishop [34] 2008	stud. (34)	ind. (13)	–	debugging spreadsheets	experts were more methodical and found more bugs
Bannerman [11] 2002–2008	stud.	ind.	–	metastudy about test-first vs. test-last development	with students test-first was typically faster but with no quality effect, with professionals it was slower and improved quality
McMeekin [134] 2009	BSc↑ (36)	ind. (26)	–	defect finding using 3 inspection techniques	same relative performance of techniques; different speed depending on experience
Mäntylä [130] 2009	stud. (87)	ind.	–	defect types found in code reviews	similar high-level distribution of defect types were found
Ricca [144] 2010	BSc	grad.	RA	effect of using web application annotations on UML diagrams	added annotations were helpful for inexperienced students
Bergersen [30] 2012	stud.[3] (266)	ind. (65)	–	debugging iterative vs. recursive code	low-skill subjects did better on the recursive version, skilled ones did the same on both

(Continued on next page)

[3]Cited previous work.

Table 1: Continued.

	Subjects				
Ref.	*L1*	*L2*	*L3*	*Study*	*Results*
Soh [172] 2012	stud. (12) nov. (9)	ind. (9) exp. (12)	–	understanding and maintaining UML diagrams	practitioners were more accurate, but for a given accuracy students were faster
Čaušević [54] 2013	MSc (14)	ind. (33)	–	test cases in test-driven development	similar behavior in both groups
Abrahão [1] 2013	low	high	–	effect of UML sequence diagrams on comprehension	sequence diagrams helped high-ability participants
Jung [100] 2013	PhD (7)	ind. (11)	–	compare 2 safety analysis tools	replication in industry gave essentially same result of no difference between tools
Ceccato [55] 2014	BSc (13)	MSc (39)	PhD (22)	the effect of code obfuscation on comprehension	more advanced programmers are better but affected in a similar way
Salviulo [154] 2014	BSc↑ (18)	prof.↓ (12)	–	the use of comments and variable names for comprehension and maintenance	students used comments more, all agreed that good identifier names are important
Daun [65] 2015	BSc (125)	MSc (21)	–	review specifications of an avionics system in either of two ways	graduate students were more effective leading to larger effect size, but results for undergrads were also significant
Salman [153] 2015	grad. (17)	ind. (24)	–	effect of using TDD on code quality	both groups produced code of similar quality when using TDD for the first time
Busjahn [46] 2015	nov. (14)	ind. (6)	–	use eye tracking to study the order of code reading	code reading is less linear than story reading, and experts are less linear than novices

At an even higher level, Jeffries et al. compared five undergraduate students to four experts (an EE professor, two experienced graduate students, and a professional) on how they actually designed a program to create an index for a book. All subjects used the same general strategy

of decomposing the problem into subproblems. However, the experts were much more effective. Thus the novices tended to make do with the first solution they found for a problem and did not consider alternatives, they missed or decided to ignore subtle end cases, and they did not rely on using known algorithmic solutions.

Putting all of this together,

> It appears that expert programmers think differently from novices. This may hinder the use of students, especially at the start of their studies, for tasks that are conceptual or presented at a high level of abstraction.

3.2 Studies Showing Similarity of Students and Professionals

Quite a few studies have compared the performance of students and professionals on concrete programming-related tasks. An especially interesting experiment was conducted by McMeekin et al. [134]. The goal was to compare three software inspection techniques: checklist-based reading (CBR), which was developed in the context of procedural programming, and use-case reading (UCR) and usage-based reading (UBR), which were developed for object-oriented inspection. The reported results were that there was no significant difference between the inspection techniques, but there was indeed a significant difference between the 36 students and 26 professional developers applying them.

However, the paper also included a relatively detailed rendition of the raw results, in the form of box plots of the distributions of number of defects found by students or industry professionals using each of the 3 methods. Except for the scale (professionals found about twice as many defects) these two graphs are extraordinarily similar, including details such as the following:

- The distribution for CBR is the least disperse, and for UBR the most disperse.
- The distribution for UBR subsumes the distribution for CBR: the plotted percentiles above the median were higher for UBR, and those below the median were lower for UBR.
- The distribution for UCR seems to be shifted slightly toward lower values relative to the other two.
- For CBR the median is closer to the 25th percentile, and for UCR it is closer to the 75th percentile.

Thus a reinterpretation of the results is that the experiment with students actually provides an excellent approximation of the results obtained with professionals, as far as the comparison between the three techniques is concerned.

Several other studies also found that the behavior of subjects with different levels of experience was very similar in everything but speed. The earliest is the study by Sackman et al. from 1968 regarding interactive debugging, which showed that both trainees and experienced professionals benefited from using an interactive setting [151]. Moreover, both groups also exhibited similar individual differences between the group members, which was somewhat surprising for the trainees which were expected to be more similar to each other. Bednarik et al. looked at the

gaze behavior of novice and intermediate programmers when using code and animations during program comprehension [23]. The only difference was that the more advanced participants did everything faster, except for the control functions (clicking on buttons to control the animation) where the speed was the same. This exception may actually lend credence to the results, by suggesting that the difference between the subjects was limited to programming experience, and did not extend to button-clicking dexterity. As another example, Porter and Votta studied systematic strategies for requirements inspections, first using graduate students and then with industrial professionals [140, 139]. Again the numbers were different, but the statistical tests comparing different approaches told the same story.

Mäntylä and Lassenius used both students in a course and professionals in actual work to study the types of defects which are found during code inspections [130]. For both groups, the vast majority (77% and 71% respectively) were found to be evolvability defects, and only a minority (13% and 21%) were functional; the remainder were false positives. However, there were some differences in the details. For example, professionals noted more defects concerned with structure and naming, while students noted more defects in code comments.

Similar correspondence between students and professionals was found by Čaušević et al., who initially conducted a study on test-driven development using 14 students. This found that two thirds of the test cases that they produced were positive, but around 60% of the defects were found by the third of the test cases that were negative. This gap was then confirmed in an industrial followup with 33 developers, where only 29% of test cases were negative but they were responsible for finding 71% of the defects [54]. Briand et al. used a similar methodology, where results from student projects were subsequently largely confirmed by an industrial followup [41]. The context was using defect data to evaluate object-oriented quality metrics. Another such replication was done by Jung et al., in which two security analysis tools were shown to support essentially the same level of performance [100].

In the same vein, Svahnberg et al. report on a study conducted in the context of a requirements engineering course, where students were seen to be able to anticipate the values of industrial practitioners [177]. Specifically, the students correctly thought that practitioners will value the business perspective significantly more than the system perspective, despite not making this distinction themselves. Daun et al. also used students in requirements engineering courses [65]. They found that while graduate students were more efficient (faster) and better at identifying stakeholder intention, undergrads produced the same qualitative results. Lange and Chaudron found that students and professionals were affected in a similar manner when they needed to work with defective UML diagrams [116].

A rather different type of study was conducted by Höst et al. [95]. They used 25 students (last year software engineering MSc) and 17 professionals (with an average of 11 years experience). But the task was not a conventional development or maintenance task, but rather a subjective assessment of the relative importance of 10 factors which may affect the time needed to complete a project. It turned out that the assessments of the students and the professional were quite similar, leading to the conclusion that students may be used in lieu of professionals in this case. This is especially interesting because the task involved judgment and not technical skill.

Similarity between students at different levels was also found in how they benefited from novel

training procedures. Runeson checked the improvement of freshmen students and graduate students who received PSP (Personal Software Process) training, and also compared this with professionals who underwent similar training [150]. The results were that all groups exhibited similar improvements in 7 factors that were quantified. The only exceptions were that freshmen improved much more than the others in estimation accuracy and productivity, perhaps because they started at a much lower level. In a related vein, a study of the code quality produced by graduate students and professionals when using a new technique, in this case test-driven development, found that the quality was similar for both groups, despite being different when using traditional methods for which they had experience [153].

The similarity of professionals and students also has an ironic twist. In a study about obfuscation, identifier renaming to obfuscate meaningful names was hard for everyone (both experienced and inexperienced subjects) in similar degrees. This indicated that "implicit documentation" as reflected in meaningful names has a stronger effect on comprehension than experience [55].

To summarize,

> In many studies students were shown to produce the same relative results as professionals, allowing for correct ranking of alternative treatments. However, professionals were typically faster or more effective.

As noted, in practically all the studies cited here the practitioners were found to be faster than the students, sometimes by a wide margin. But the opposite is in principle also possible. For example, Schenk et al. found that expert information systems analysts took more time and verbalized way more considerations than novices when performing a system analysis task [155]. Likewise, Atman et al. found that professional engineers take more time than students to solve a given problem, because they spend more time on problem scoping and on information gathering [10] (albeit this was in the context of designing a playground, not software). Such results are representative of studies which found differences between students and professionals.

3.3 Studies Showing Meaningful Differences

We already saw some examples of studies that demonstrated meaningful differences between students and professionals in Section 2. Here we review a few more which identify specific elements of expertise which may be missing in students.

A relatively early study was conducted by Basili and Selby to compare the relative effectiveness of code inspections and testing schemes [15]. Their results show that code reading leads to better results for professionals, but not for students. An especially noteworthy aspect of this study is that they also classified their subjects according to their level of expertise based on academic performance and years of experience; for professional subjects their manager's opinion was also taken into account. Among the students, 29 were thus classified as "junior" and 13 as "intermediate". Among the professionals, 13 were "junior", 11 were "intermediate", and 8 were "advanced". But the results were that performance differences between professionals and students (irrespective of experience) were more consistent than performance differences between subjects with different expertise levels. This is somewhat tempered by the fact that differences between expertise levels

were rather noisy, possibly because the assignment into expertise levels was not based on a direct evaluation.

Comparable results were obtained by Soh et al., in a study where the understanding and maintenance of UML diagrams was used as the experimental task [172]. This study involved 12 students and 9 practitioners. But in addition to comparing them based on status, they were also compared based on their years of experience, and the result in this case was that years of experience was the more important factor. However, this result is somewhat tainted by the relatively small number of subjects and by the large overlap between the classifications: 8 of the 9 practitioners were considered experienced, and 8 of the 12 students were novices.

Berander reports on student behavior that deviates from the observed behavior in industry in the context of requirements prioritization [27]. This was done by conducting a prioritization game, where clients set priorities, developers provide time estimates, and together they draw a schedule of how the planned features will be distributed across product releases. Observations from industry, and also from students working on a large-scale project with industry partners, shows that there is a clear tendency to pile the vast majority of the requirements into the "most important" category. But when students played the role of clients in a classroom study, they overwhelmingly tended to divide the requirements evenly between the different priorities. This simplifies the scheduling but is unrealistic.

A possible generalization of behavior is that experts are more methodical [99]. In a non-software example, professional engineers took more time than students to design a playground, and the extra time was spent mainly on problem scoping and information gathering [10]. In debugging a spreadsheet, professionals covered more of the cells more consistently [34]. But the professional outlook can also be a disadvantage: in the debugging tasks, professionals were found to be more tuned to logic and concepts, so they tended to miss superficial issues like a typo in a constant.

Lui and Chan give a striking example where the level of expertise may confound the experimental results [125]. Their study was concerned with the effectiveness of pair programming. Many studies have been conducted on this topic, with mixed results. The novelty of Lui and Chan's work was in controlling for the level of the participants. They conclude that pairs of novices, or in general pairs of programmers confronted with new and challenging problems, stand to benefit considerably from pair programming. But for pairs of experts, or generally pairs confronted with problems they have handled before, this is a waste of effort. The experiment showing this was based on repeat-programming: the subjects were asked to solve the same problem time after time, and the benefit of working in pairs was seen to drop with iteration number.

Arisholm and Sjøberg designed an experiment showing how mixing people with different levels of competence can lead to problems in the field [9]. Specifically, they used two different design styles to solve the problem of implementing a simple coffee machine: one in which the front panel object is in control and just uses other object to perform simple tasks, and the other in which the front panel merely initiates activities and delegates the actual execution to other objects which encapsulate the details. The delegated style was considered the better object-oriented design. In the experiment, undergraduate students and junior consultants were found to have problems maintaining the delegated style, whereas graduate students and senior consultants performed better

with this style. Thus if senior designers are employed and use the preferred style to design a product, there is a danger that more junior maintainers will be ill-equipped to maintain it.

In followup work with additional colleagues, Arisholm et al. compared the above results (but excluding the students) with the work of pairs of professional developers [8]. The results were that working in pairs was in general not beneficial, as the strongest effect was an 84% increase in total effort (because two people were involved and it took approximately the same time). However, they did observe an interaction with problem complexity: pairs of junior and intermediate consultants achieved significantly better results than individuals on the complex (delegated) problem, and pairs of intermediate or senior consultants took somewhat less time than individuals on the simple (centralized) version.

It is hard to generalize the above results. But we can state that

> Many factors affect the performance of software engineering tasks, and experience or expertise sometimes figure among them. However, this often interacts with other factors, and pinpointing the precise effect is difficult. Many more experiments on diverse aspects of this issue are desirable.

4 Discussion

As shown in the previous sections, there is a wide range of studies which have considered — or grappled with — the confounding effects of experience on software engineering performance. But can this be limited to a discussion of using student subjects as opposed to professionals? And what other considerations should be applied? In the following we discuss these and related questions based on the evidence reviewed above and additional literature from other fields concerning topics like experience and expertise.

4.1 The Semantics of "Student" and "Professional"

The basic complaint against using students as experimental subjects is that they are perceived as not representative of professional developers in a "real" industrial setting. But real life may be more complicated, leading to situations where this distinction does not correspond with the intent.

First, it should be noted that "students" come in many varieties. Undergraduate students are not the same as graduate students. Within the set of undergraduates, freshmen (first year students) are different from second or third year students. Likewise, within the set of graduate students, a distinction can be made between masters students and doctoral students. In fact, many studies compare undergraduate students to advanced graduate students when they want to assess the effect of experience, or else use students from different levels to control for the effect of experience [1]. In addition, there may be differences between students studying in different degree programs (e.g. software engineering vs. computer science) and at different institutions (e.g. a community college vs. a technical university).

Second, the dichotomy assumes that students lack professional experience. This is not always the case. Many youngsters nowadays learn to program in their teens, and can accumulate quite

a lot of experience in hacking or contributions to open source projects before they graduate from high school. Some may even establish their own startup companies. In Israel, for example, many students come with experience from their army service [156]. In some study programs internships in industry are incorporated into the program, thereby giving the students some professional experience as they learn. Students may also work in parallel to their studies in order to support themselves; thus study participants identified as coming from industry may actually also be students, and even undergraduates (e.g. [129]). In all these cases, the students can therefore actually have significant industrial experience.

And third, the term "professional" is also not well-defined. Different terms are sometimes used, such as "developers", "practitioners", or "engineers", and it is not clear exactly what they mean and whether they are indeed different [69]. Their training is also not necessarily the same; For example, Arisholm and Sjøberg report a case where senior consultants in general held higher degrees than junior and intermediate consultants [9]. Moreover, a professional can be new on the job or have the benefit of many years of experience, just like students [26]. And finally, their jobs may actually be quite different too. For example, are professional developers and testers the same? Are those assigned to develop new functionality the same as those assigned to perform maintenance tasks [38]? And what about having experience in different methodologies and approaches? While it stands to reason that experience in a specific relevant domain may be beneficial to experimental subjects (e.g. [4]), one can also argue for more general capabilities that come with experience [173, 175].

> The terms "student" and "professional" are ill-defined, and not necessarily exclusive. Using them to label experimental subjects may be misleading.

4.2 Novices and Experts

Beyond the labeling of students and professionals, the more meaningful objection is that students are *novices*, and therefore do not represent the work practices of professionals who are *experts*. So it is worth our time to consider these terms as well.

Experience is acquired over time by receiving feedback in real-world situations. Expertise is the result of internalizing this experience to create an understanding of concepts and decision-making procedures, together with an appreciation for potential biases and problems [155] (as also reflected in the Confucian saying "I do and I understand"). Novices by definition lack such expertise, and are therefore prone to making errors in complex decision-making situations.

The distinction between novices and experts is very important in any field, and also in software development. Experts know more and do things better, using different approaches than novices. Specifically, experts rely on deep domain knowledge to get to the crux of the problem. Novices are typically limited to relying on superficial cues. In addition, it has been observed that novices treat complex systems mainly at the structural level, whereas experts understand the behavior and function of the system elements [20, 94, 93]. This distinction may be expected to be particularly relevant to software too. Select attributes of the differences between novices and experts, as reviewed by Lord and Maher, are listed in Table 2.

Table 2: *Characteristics of novices and experts, based on [124, 94].*

Feature	*Novice*	*Expert*
Search	simplified heuristics until satisfactory alternative found	automatic homing in on best alternatives
Selection	based on superficial features	based on meaning
Information	modest amount used	extensive and highly organized
Focus	system structure	behavior and function of system elements
Process	typically serial	often parallel

In the field of technical education more levels are identified. An influential report on this issue was written by Dreyfus and Dreyfus [71]. The main levels they identified, and the characteristics of each level, are

1. Novice: knows to apply learned rules to basic situations
2. Competent: recognizes and uses recurring patterns based on experience
3. Proficient: prioritizes based on holistic view of the situation
4. Expert: experienced enough to do the above intuitively and automatically

Thus the effect of experience is to enable the practitioner to break away from prescribed rules and apply a wider perspective, eventually doing this without conscious effort. A fifth level, that of a master, has been suggested to reflect the ability to innovate and develop new tools and methods to cope with new situations.

While the difference between novices and experts may be quite significant, the question of whether it is important in the context of software engineering experiments is debatable. One view is that expertise is not really important, for any of the following reasons:

- In many cases the tasks being performed by experimental subjects are not challenging enough, and do not require a holistic view or innovative solutions. Thus the differences between novices and experts come into play only during extended work on large-scale and difficult problems, and not in experiments.
- Alternatively, when experiments involve technology or methods that are new to everybody, then skill or previous experience does not necessarily impart an advantage.
- Finally, even if advanced professionals may do everything faster, and maybe even differently, still the qualitative experimental evaluation could well be the same as with students or other inexperienced subjects.

The other view is that the differences come into play even in simple situations, as experts have a significant advantage in grasping the situation and understanding the interactions between the

code and the context. For example, Gugerty and Olson compared novices to skilled programmers in a debugging task, and found that the skilled ones were faster, found the bug more often, and hardly ever introduced new bugs while hunting for the existing one [87]. This was attributed to the ease with which they were able to dissect and encode the program, and thereby to comprehend the code and come up with hypotheses about what was wrong.

Moreover, hardened professionals may acquire various practices with time and experience that affect their approach, and cause them to deviate from what may be expected and what students are taught. For example, Roehm et al. conducted an observational study of how professionals comprehend software when they need to perform a maintenance task [149]. A surprising result was that they may actually try to *avoid* comprehending the code, using various shortcuts instead. For example, one participant identified which functions are relevant by first commenting out all the applicable code, and returning the functions one by one until there were no more compiler error messages. As another example, Marasoiu et al. observed professional software developers who used an environment's code completion feature as a debugging aid — if the code completion did not work, they took it as an indication that something was wrong [135]. So experts can leverage existing technology to their advantage in unorthodox ways.

> There are real differences between novice and expert software developers. Being able to place experimental subjects on this spectrum is generally desirable, if only to see whether there exists an interaction between performance on the experimental task and expertise.

4.3 The Making of an Expert

Using a title to describe experimental subjects (namely, labeling them as "students" or "professionals") is often a stand-in for asserting their level of expertise. A possibly better alternative is to quote their *experience* (that is, years of doing software development), and indeed many studies report that experienced developers perform better (e.g. [75, 56]). For example, Robilard et al. studied how five developers approached maintenance tasks on an open-source project (specifically, changes to autosave functionality in jEdit) [146]. They found that experience makes a big difference in success rates and reflects real differences in approach. Crosby et al. show that advanced programmers are somehow better able to focus on the complex parts of the code [59], and Busjahn et al. show that advanced programmers read code differently from novices [46]. But in fact experience in years is also just an easy-to-measure proxy for different levels of proficiency or capability.

Moreover, experience is not the whole story. Another consideration is that there is evidence for very significant variability among programmers. Perhaps the first study to show this was the one by Sackman et al. from 1968. The goal of this study was to compare online and offline approaches to debugging. But one of its main findings was that the biggest differences in performance were due to personal variability between experimental subjects, which in one case reached a ratio of 28:1 [151]. Large individual differences were also cited in various other studies, e.g. [136, 60, 63, 111, 75, 141, 142]. Personnel/team capability figures prominently on the cover of Boehm's famous book *Software Engineering Economics* as the cost driver with the highest range of possible

values, much more than any other factor[4] [36]. McConnell recounts an anecdote where a single programmer was called in to replace a team of 80 (!) when a critical project was in risk of missing a deadline [132]. In the specific context of empirical research, such variability could swamp out the effects being studied [62].

Evidence for inherent differences in proficiency can also come from computer science education. In particular, a bimodal distribution of grades in CS 101 courses is sometimes observed, and has led to discussions of the mechanisms that create it. One of the options is possession of a "geek gene", namely that some people are predisposed to computer programming and therefore have an edge on others [5]. More precisely, various concrete characteristics have been cited, such as being able to clearly articulate a problem-solving strategy [165] and holding valid mental models of value and reference assignments [127]. However, it is important to note that one of the more famous reports suggesting that good programmers can be identified in advance based on having consistent mental models of program behavior was later retracted when more experimental data was collected [37] and a replication also failed [126]. And other explanations have also been proposed, e.g. that a fluke success early in the course can set a student on the better learning path [147].

Based on the above, proficiency may be thought of as representing the cumulative influence of three separate factors:

- Natural talent. Personal differences between different people are most probably very important. Jackson has a wonderful short story equating "brilliance" with the talent to make things appear simple [96, p. 20]. Sex has also been shown to have some impact [158].
- Formal training and deliberate practice. It may be expected (or at least hoped) that a university or college education improves one's proficiency, and that advanced degrees have some added benefit over that obtained from a first degree. The same goes for vocational training. And copious amounts of deliberate practice have been claimed to be the real factor leading to world-class performance [76, 77, 58].
- Experience on the job. This is easy to measure, at least superficially (that is, length of experience as opposed to quality of experience), but is at best only one of several factors that affect proficiency.

The conclusion is that

> Experimental studies should try to assess the actual proficiency of their subjects, and not just tag them as "students" or count their years of experience.

4.4 Experience and Proficiency

Even if experience is only one factor contributing to proficiency, it could still be that they are highly correlated. It is not at all clear that experience is necessarily correlated with proficiency in

[4]The graphic on the cover (Figure 33-1 from the book) shows the range to be 1 to 4.18, more than double the range of 1 to 2.36 for product complexity which is the next factor. But a factor of 4.18 is actually less than twice a factor of 2.36.

all cases: developers with many years of experience may still be mediocre, while relative novices may be very proficient. Thus years of experience, while easy to measure, may turn out to be a poor predictive attribute [28]. Also, even if more skilled developers may be more efficient in term of time, their work may not necessarily be of higher quality [83].

It is also important to note that "experience" is by definition limited to those domains and methodologies in which an individual has worked. Thus it was found that professionals did not produce higher-quality code than students when the experiment involved using a methodology with which they did not have any prior experience, in this case test-driven development [153]. Likewise, advanced students did not have an edge over novices when confronted with obfuscated code, which was beyond the scope of their previous experiences [55]. But strange counter examples also exist: Carver et al. have found that inspectors with non-computer-science degrees found more requirements defects than those *with* computer science degrees [53]. Another interesting and complicated interaction was discovered by Basili and Selby [15]. They found that for professional developers code reading was more effective than either black-box testing or statement coverage. With students, however, they did not find such clear results, and in one experiment all three methods were indistinguishable. This result is especially noteworthy given that the professional developers' main prior experience had been with functional testing, not with code reading. In other words, the advantage delivered by code reading depended in some *indirect* way on the experience of the professional developers.

Sonnentag et al. provide a deep discussion of the differences between *experienced* developers and *highly-performing* developers [175]. For example they claim that experienced developers tend to spend more time than inexperienced ones on program comprehension and on clarifying requirements, perhaps due to having experienced situations where this was a problem (a similar trait has been observed for engineers in general [10]). Highly performing developers, on the other hand, tend to spend *less* time on program comprehension, possibly because they "get it" faster. An earlier study on high performers identified several differences between them and more mediocre performers, but emphatically years of experience was not one of them [174]. However, it should be noted that these results pertain to the comparison of professionals who were all experienced, and not to the comparison of experienced professionals to novice students.

An interesting observation is that the effect of experience is not linear. Sackman et al. ran an early experiment of debugging time in online vs. offline settings. This included an initial assessment of aptitude before taking part in the experiment itself. The result was that the aptitude test had a good correlation with the experiment results for trainees, but not so for professionals with an average of 7 years experience. In discussing this they speculated "that general programming skill may dominate early training and initial on-the-job experience, but that such skill is progressively transformed and displaced by more specialized skills with increasing experience" [151]. Thirty years later Sonnentag also showed that years of experience were not correlated with differences in performance for professionals, while citing results claiming that experience is indeed important for students [174]. This can be interpreted as suggesting that students are still acquiring knowledge, whereas seasoned professionals have already reached a plateau in their capabilities. Similar observations were reported by Bergersen et al. [28]. These results agree with the "laws of practice", which state that initially, when one lacks experience, every additional bit of experience makes a

significant contribution, but when one is already experienced, the marginal benefit from additional experience is much reduced [137, 91].

> While experience may certainly contribute to capability and change how problems are tackled, it seems that using years of experience or employment are an overly simplistic metric for qualifications. In particular, beyond several years of experience individual differences probably become more important than differences in experience.

At the very highest levels of performance, experience is not enough. For example, the seminal work of Simon on chess masters indicated that the most highly skilled needed to have invested a significant time in obtaining experience, but not that anyone who invests such time automatically becomes a master [166]. Similar findings have been uncovered in many other fields as well [76]. Rather, deliberate practice is needed, meaning extensive repeated practice directed specifically at those areas in which you are not yet proficient, and directed by a mentor who provides candid feedback [77, 57, 58]. While investing time is also required (ten years and 10,000 hours is often cited as a minimum), it is how this time is used that is really important. But in an ironic twist of affairs, this also explains why such top-notch expert performance is largely irrelevant in normal contexts: deliberate practice is not fun [76, 58], so most people do not engage in it. In the specific context of software engineering experiments, we are more often interested in studying "normal" practitioners than those who have the exceptional capacity to push themselves to the limit.

4.5 Assessing the Level of Proficiency

Assessing the proficiency of experimental subjects is important for several reasons [110, 136]:

- First, it may enable us to claim that the results are relevant to a general setting, based on an evaluation showing that the subjects possess a representative level of proficiency.
- Second, in the context of designing the experiment, we can verify that groups of subjects using different treatments are not biased in terms of ability. We may even use this to screen candidates so as to reduce the variance in capabilities, and avoid mixing subjects with different performance levels. Such mixing — which occurs naturally due to the large variability in individual capabilities — increases the dispersion of results for each treatment, and may mask the underlying differences due to the different treatments. This is especially problematic with small numbers of subjects.
- Third, by quantifying the proficiency of different subjects, we can assess whether there exists an interaction between the experimental treatment and the level of proficiency [30]. For example, this will tell us whether using a certain method or tool is more beneficial for proficient developers, for mediocre ones, or for novices.
- Moreover, identifying the specific strengths and weaknesses of individual subjects [2] may uncover additional interactions and help avoid construct validity issues.

A Major issue is of course how to design tests with good discrimination power, which will effectively identify and rank the students (or other experimental subjects) on a scale from the most proficient to the less so. Moreover, software design tasks are typically ill-defined, meaning that the problem specification is incomplete and discovered as part of the solution process [175]. As a result there is no single correct solution. Therefore evaluations of programming and development proficiency must contend with the need to evaluate the quality of the results. Tests therefore sometimes focus on small well-defined tasks rather than the general development process.

In the 1960s IBM devised the programmer aptitude test, which was designed to aid in the evaluation of job candidates[5]. This is actually more of a standard IQ test, with questions about completing series of numbers, series of shapes, and high-school arithmetic word problems; it has nothing to do directly with programming. Nevertheless, this and some other tests became used by a large percentage of the industry, and led to the formation of the ACM special interest group on Computer Personnel Research (SIGCPR) [131]. However, the tests' ability to predict on-job programming performance was debated.

Some ideas about assessing programming ability have been gleaned based on analyses of the Advanced Placement Computer Science A tests administered in the United States by the College Board, a private company tasked with developing and administrating various standardized tests. This is a large-scale test offered to high-school students, meant to be commensurate with the level achieved following a first-semester college course in Java.

The first analysis of such test results was performed by Reges, who analyzed the 1988 test results [143]. Interestingly, he found that five specific questions had non-trivial correlations with all the rest, and therefore provide a good indication for overall success. (The question with the highest correlation was "If b is a Boolean variable, then the statement b := (b = false) has what effect?", and made it to the title of Reges's paper.) It was speculated that these questions thus capture some core elements of computer science understanding. The same data was later re-analyzed by Lam et al., who concluded that questions with legal code snippets in them were those that had the best correlation with other questions (and hence with overall score) [115]. However, this is tainted by the fact that nearly two thirds of the questions were in this category. They also identified questions about arrays, linked lists, compilers, and recursion as having relatively high correlations. Questions about invariants or including type definitions had relatively strong negative correlations.

A more detailed analysis was conducted by Lewis et al. using the results of the 2004 and 2008 tests [121]. Note that the tests had of course changed since 1988. For example, the programming language used was switched to Java. More importantly, more questions included code in them, and by 2009 this applied to fully 95% of the test. Therefore having code could not be considered a discriminatory characteristic of different questions. This study failed to recognize any specific common features in questions that are better at discriminating among test participants, even though such questions were indeed identified.

In a related vein, Ma et al. studied the results of first year students learning to program in Java. They claim that a strong correlation exists between final grades and holding valid mental

[5] J. L. Hughes and W. J. McNamara, IBM Programmer Aptitude Test (Revised), Form Number 120-6762-2. A scan is available at URL http://ed-thelen.org/comp-hist/IBM-ProgApti-120-6762-2.html.

models of value and reference assignments [127]. Likewise, Simon et al. claim that the ability to clearly articulate a problem-solving strategy is a good predictor of success in competence tests [165]. These examples imply that the tests indeed measure deeper understanding and not only superficial technical competence. But Bayman and Mayer report a contrary finding, where many novice programmers had misconceptions about what various BASIC statements do despite being able to pass tests [21]. Also, the old study by Sackman et al. found little correlation between grades in tests and no-job performance [151].

In the context of empirical software engineering research, the idea is to use a separate common pretest task which is done by all subjects to form a baseline for comparison [9]. Then different subjects do different treatments, and their results are compared with their performance on the common initial task. This has the additional beneficial attribute that the pretest measures performance directly, and avoids all the confounding factors related to experience, education, and motivation.

Pretesting has been used in several studies, but surprisingly there has been very little work on devising the tests. A specific test to measure programming skill has recently been proposed and validated by Bergersen et al. [31]. This involves 12 Java programming tasks, and takes several hours, so it is suitable for only large experiments. Importantly, it combines scores for quality of the solution with the time needed to achieve this solution [29].

It has also been suggested that the pretest can focus on specific relevant aspects of proficiency and knowledge [51, 112, 2]. But one should be aware of a dangerous loop. In many cases, proficiency is included as one of the independent variables that may affect and explain performance differences. But if the test used to assess proficiency is similar in any way to the experimental procedure being investigated, this may lead to a meaningless finding that those who are better at performing the test are also better at performing the related experiment. For example, Bateson et al. set out to show that experienced programmers (students who had taken more than 3 programming courses) have better memory and problem solving skills than novices (students who had taken up to 3 programming courses), by using tasks and questions similar to those used in final exams of programming courses [19]. While they found that the experienced subjects did better, this provides little information beyond the observation that students retain at least some of what they had learned. It is therefore necessary to devise general proficiency tests which are independent of the issues being studied [42].

Siegmund et al. studied the use of background questions for assessing programming-related proficiency. To do this they considered the correlation between answers to various potential questions and performance in 10 program comprehension tasks, based on the assumption that more capable subjects should be able to perform better [162]. The result was that the highest correlation was obtained for the question "On a scale from 1 to 10, how do you estimate your programming experience?" (Spearman correlation coefficient of $\rho = 0.539$), and the second highest was for the question "How do you estimate your programming experience compared to your class mates?" ($\rho = 0.403$). The question about years of programming experience led to a somewhat lower yet statistically significant correlation ($\rho = 0.359$). Questions about level of education (e.g. number of courses) led to much lower and statistically insignificant correlation.

Kleinschmager and Hanenberg likewise compared the use of pre-experiment tests, grades in university courses, and self-assessment by the subjects themselves, in two separate experiments

involving 20 and 21 students. In both cases the self-estimates allowed them to divide the students into three groups, such that the performance of the top and bottom groups were significantly different [110]. Course grades and pre-testing did not lead to such results; for example, in one of the experiments students with lower grades in programming courses actually performed better in the experiment. Thus in their admittedly limited sample using grades or tests were not better than relying on the self assessment of the subjects. This may be explained by students studying together having a better notion of how they really compare with each other. But it is not clear that such self assessment would work equally well in a more general setting.

A third related study, by Aranda et al., found that self assessment of expertise was not correlated with effectiveness (ρ between -0.01 and -0.05), and might suffer from over-estimation bias [6]. In this case experience fared slightly better, but again the sample size was small.

The bottom line of this discussion is that

> It is hard to devise a good proficiency test, and more work on this is issue is required. For the time being, self-assessment appears to be a reasonably good option.

4.6 Limitations of Controlled Experiments

In the preceding sections we discussed the meaning of being a student and the need for tests which differentiate experimental subjects according to their abilities. Here we consider another dimension of the problem altogether: what controlled experiments can be used for. The idea is that if controlled experiments are in general of limited scope, then maybe it is acceptable for the experimental subjects to be limited too.

We have already noted above that controlled experiments in software engineering are often chastised for using toy problems (e.g. Figure 1 and the related discussion). The reason for using tasks with very limited scope is to keep the experiments tractable, allowing subjects to complete them in a relatively short time, and enabling the collection of results from multiple subjects in the interest of statistical power. But it is plausible that there are issues which simply cannot be investigated using such controlled experiments, because they cannot be condensed into a suitably limited framework.

One example for this is the effect and possible benefits of test-first development. Test-first development is a component of agile methodology, and in particular of Extreme Programming (XP) [22]. A meta-study conducted by Bannerman and Martin showed that half of the 6 controlled experiments investigating this issue using students found that test-first led to faster progress, and a solid majority found that it had no effect on quality [11]. Conversely, of 11 studies using professionals, which were predominantly case studies, 4 found that test first was slower, and 8 found that it led to higher quality. Thus the results interacted not only with the experimental subjects, but also with the study type, and perhaps the different results were not due to using students but due to using controlled experiments. In a related vein, di Bella et al. argue that studying pair programming — a basic component of XP — should be done by observing industrial developers in action over a prolonged period rather than using small test cases in controlled experiments [67].

More generally, it seems to be agreed that controlled experiments are ill-suited to answer the "big questions", such as the benefits of object-oriented programming and agile development, the

patterns of software evolution, and so on. These issues encompass an extremely wide and complicated set of factors and interactions, and really come into play mostly over extended periods and large projects. This realization has led to the growth of the whole field of mining software repositories as part of performing case study research, often augmented by surveys of professionals, and leading to theory building.

Furthermore, controlled experiments can also not be used when studying professional work practices and the dynamics of large-scale software development, e.g. the communication between different teams taking part in a project [64]. Thus

> Assuming we accept that controlled experiments are limited to relatively small, focused, and well-defined tasks, the expertise of experimental subjects may be of limited importance and students may be suitable.

4.7 Other Issues and Confounding Factors

The obsession with the question of using students as experimental subjects may mask other important factors. First, as we have discussed extensively, it may actually be the wrong question, because what we are really interested in is probably proficiency. Second, there are other potentially important factors that receive much less attention, such as sex, personality, and mood [89, 86].

In describing empirical studies one often includes demographic data such as the number of male and female subjects, but most often this is not used as an independent variable when analyzing the results. One exception is the work on program reading patterns by Sharafi et al., in the context of a study on different identifier styles. Specifically, they write that "male and female subjects follow different comprehension strategies: female subjects seem to carefully weight all options and spend more time to rule out wrong answers while male subjects seem to quickly set their minds on some answers, possibly the wrong ones" [158].

Unlike age and sex, the personality of experimental subjects is typically not assessed in the context of controlled experiments. This does not mean that it is not important. Personality can be characterized in various ways. One of them is the Myers-Briggs personality type notation, which consists of a combination of 4 letters [66]:

E/I denotes being and **e**xtrovert or **i**ntrovert. Extroverts thrive on human interaction, whereas introverts tend to be loners.

S/N denotes relying on the **s**enses to obtain information, or alternatively relying more on i**n**tuition.

T/F denotes the way decisions are reached: either **t**hinking it out logically, or based on **f**eelings and the opinions of others.

J/P denotes using **j**udgment to plan everything in advance, as opposed to using **p**erception and being more spontaneous and flexible.

Such classifications have been used to classify programmers and check whether their personality affects their performance. For example, Capretz studied a sample of 100 software engineers and found that a full 24% of them were of ISTJ type, more than double the rate in the general population

[47]. The most significant dimension was T/F, with 81% being thinking to only 19% feeling. Followup work suggested that different personality types are suitable for different roles and different stages in the software lifecycle [48]. Conversely, Turley and Bieman found that among exceptional software engineers (albeit a small sample) half were INTJ, that is, using intuition rather than sensing information, and 85% overall were thinking types [181]. Bishop-Clark and Wheeler conducted a study showing that sensing (S) students performed better than intuitive (N) students, and judging (J) students were better than perceptive (P) students, but only on programming tasks and not in final grades [35]. Similar studies have implicated personality types in the success of pair programming [25, 186, 152] and in code reviews [66]. Finally, Turley and Bieman claim that when trying to classify software engineers into exceptional and nonexceptional ones, general personality traits such as "helps others" and "willing to confront others" were among the dominant discriminant factors [181].

Personality and sex are attributes of experimental subjects which may be just as important as expertise.

The subjects being used are an important attribute of controlled experiments, but there are others [72]. According to Votta and Porter, empirical research in software engineering must contend with 3 dimensions [184]:

- Individuals vs. groups
- Students vs. professionals
- Lab conditions vs. real life

We have focused on the second of these, and extended the discussion to also include different levels of proficiency. But this may not be the most important dimension. Controlled experiments nearly always use subjects working individually, and in laboratory conditions, namely tackling well-defined problems of modest size. This may be cause for significant threats to external validity.

Working in groups may be different. First there is the issue of division of labor, where each developer needs to focus on only part of the problem. This also facilitates a measure of specialization, allowing different developers to become experts in their individual areas of responsibility. Second, group dynamics and interactions facilitate cross fertilization and the exchange of ideas, ideally leading to a situation where the whole group performs better than the sum of its individual participants. Thus the social context has an effect on how people work [145]. Conversely, personality clashes may come into effect in groups and disrupt their progress, an effect that would not occur when developers work individually. To account for such effects, Basili and Zelkowitz suggest to perform a series of experiments progressing from individual programmers to groups, and from students to industrial professionals [18].

Likewise, there are possible interactions with problem size. The toy systems used in experiments may be unrepresentative of real problems encountered in professional work. In particular, it has been observed that real complex problems are needed to bring out the advantage of expertise [155]. Putting these two aspects together, Damian et al. report on research showing that team

members with different roles and different domain knowledge hold the key to effective functional communication in a project [64]. This study naturally relied on observations, interviews, and surveys of professional practitioners involved in multi-team projects, and could not be done with students in a lab.

However, one should also not exaggerate the importance of working in groups or on large and complex problems. The importance of these factors may depend on the study. When studying code comprehension, for example, individuals and limited code may well suffice. When studying tool usage individuals again may suffice, but the amount of code needed may be larger in order to bring out the benefits of using the tool. Then there is the question of whether we are seeking relative results (a comparison of two methodologies to see which is better) or more precise absolute ones.

> The study context, e.g. working in a group or in real-life conditions, may perhaps be more important than attributes of experimental subjects.

4.8 Compensation by Experimental Procedures

The main perceived problem with using students, as articulated in many of the works cited above, is that they are inexperienced and therefore not representative. Another problem is the danger that there will be a very wide range of results, and that this variability would swamp out the experimental effects [62]. Therefore such variability needs to be controlled or at least factored out, by using appropriate experimental methodology.

An interesting approach to reduce the adverse effects of conducting experiments with inexperienced students was suggested by Carver et al. [50]. The concern was that when evaluating a new technology the students are too low on the learning curve, so their achievements will not be representative of professionals who spend more time to learn the new approach. The suggested solution was to divide the students into pairs, and conduct the study twice: first one performs the task and the other observes, and then they switch roles and do it again. The measurement is done only on the second time. The first is used only to accelerate the learning by providing hands-on experience.

Another interesting alternative for assessing the effect of experience is to do so in reverse. Bednarik and Tukiainen identified two strategies of performing comprehension tasks in their eye-tracking study [24]. So they retroactively divided the study participants according to the strategy (in particular, how many program animation runs they used), and analyzed the background data on the two groups that were produced. The results were that participants in one group had significantly more previous programming experience than participants in the other group, and also included the two professional programmers that had participated in the study.

Regarding the effect of the large variability of results, using students may actually reduce the variability because they all have about the same level of education, leading to better statistical characteristics [17, 112]. Nevertheless, one should always apply an independent test of general proficiency and filter outliers to reduce variability. Alternatively, it may be possible to create teams with the same mix of proficiencies, e.g. by identifying the top-performing students and distributing them among the teams [13].

Brooks suggests the use of within-subject experimental designs to compensate for differences in ability [42]. Variance is reduced and the analysis is indeed improved by analyzing the distribution of within-subject differences between the treatments, rather than trying to first characterize the performance for each treatment individually [119, chap. 11]. Within subject designs are a special case of block designs, where levels of the uninteresting factor (in our case, the experience or proficiency of experimental subjects) are randomly balanced across experimental treatments. This and other approaches to statistical analysis are discussed at length by Juristo and Moreno [101]. A problem with such analyses is that traditionally the analysis was based on assumptions relating to the normality of the data. However, robust alternatives that apply to any distribution are also available [189, 190].

The advantage of within-subject designs stems from the fact that each subject is exposed to all the different levels of the treatment. Thus, if a certain subject's abilities are either above or below the norm, this will apply to all the treatments in the same way. However, if the task being performed is the same, there is a significant risk of a learning effect [108]. In the first instance the subject needs to contend both with a new task and with the specific treatment being used, but in the second instance the task is already known, so only effects of the treatment remain. Using different tasks eliminates the learning effect, but introduces the task as a new (and possibly no better) confounding factor.

> It is desirable to focus on within-subject differences when mixed populations of subjects are used, provided learning effects can be ruled out.

Finally, an important experimental tool is replication. In particular, external replication (that is, replication by other researchers) is considered an especially effective technique to increase confidence in a result. This increase in confidence is contingent on the unavoidable variations between the original and the replication, which show that the observed effect is indeed robust. In the context of software engineering experiments, a major element of this variability is the use of different experimental subjects [161, 103, 81]. And in particular, replications using professionals can increase the confidence in student-based results.

4.9 Students Explicitly Desired

Nearing the end of this discussion, we consider a special case of software engineering experimentation which explicitly targets novices, e.g. to see how certain tools may help them to perform certain tasks despite lack of prior knowledge or experience [109]. In such situations is may actually be appropriate to focus on undergraduate students and exclude more mature students in order to reduce the variability in the subjects' level of experience.

For example, Liu et al. describe a tool called InsRefactor which is designed to help novice programmers refactor their code and resolve code smells [123]. The idea is to proactively alert them to code smells as they are created, rather than leaving it up to them to request information about code smells retroactively. To investigate the effectiveness of this tool a controlled experiment with two groups of students was used. Similarly, Fernandez et al. used students to test a usability

inspection tool integrated into a web development process, and explicitly justify this by citing "the intention ... to provide a Web usability evaluation method which enables inexperienced evaluators to perform their own usability evaluations" [82]. A third example is the work of van Heesch et al., who studied the degree to which documentation helps junior designers [182]. And Busjahn et al. use novice students specifically as a contrast to experienced professionals, to show how experience affects code reading patterns [46].

Briand et al. conducted a study of how quality guidelines affect maintainability, using students as subjects [39, 38]. In particular, they identify the cognitive complexity of object-oriented designs as a potential problem. While they frankly note that this causes a threat to external validity as it is not clear that the results generalize beyond inexperienced students, they argue that inexperienced programmers are often assigned to maintenance tasks, and therefore student subjects are actually appropriate in this case.

Another situation where students are explicitly needed is in the evaluation of educational tools and procedures. One example is the study by Runeson concerning the effectiveness of PSP (Personal Software Process) training [150]. He found that the improvements from PSP level 0 to PSP level 2 was similar for freshmen and for graduate students. Another example is described by Janzen et al. [97]. This is a study of an educational platform called WebIDE which was used to teach introductory Java and Android programming. A controlled experiment was used to compare a group of students who used this platform with another group who used a more conventional lab setting. A third example is a study on the effect of internship on on-job performance [70]. In this case two groups of interns were assessed for five months as they used agile methods on projects in the telecom industry.

Finally, Kuzniarz et al. suggest that the worst-case experiment for a new methodology or tool is when subjects know about the (established) experimental alternative but not about the new treatment [113]. Under this scenario a positive effect regarding the experimental treatment is especially convincing. And students are especially suitable for such experiments, as their lack of experience increases the chance that they do not know of the new treatment.

5 Recommendations Regarding Using Students

As expanded in Section 6 below, our main conclusion is that the issue of using students as experimental subjects should not be the factor which determines whether an empirical study is considered worthwhile. Students may be used beneficially in controlled experiments in many cases, and as far as their availability leads to conducting more such experiments their use may catalyze significant contributions to software engineering research. But there are indeed cases where students would be inappropriate.

The main consideration regarding the use of students is that their level should be matched to the requirements of the study being performed. This leads to the following more specific recommendations:

1. Studies of problems in beginning to program, programming education, or non-programmer

end-user assessment can use novice students (in their first year, after one programming course).

2. Studies that do not require an extensive learning curve can use intermediate students (toward the end of their BSc, but with no industrial experience). In effect such students can be expected to have similar capabilities as beginning professional developers, leading to valid relative results when comparing straightforward tools or methodologies.
3. More precise quantitative studies or those requiring more experience may use advanced students (graduate students in a programming-related program and/or with industrial experience). However, it is always advisable to assess them individually and not count on their schooling and experience alone — and this applies also to non-student subjects. Training should be provided as appropriate.

Students should generally *not* be used in studies that depend on specific expertise which requires significant experience and a long learning curve to achieve, or in studies of professional practices. Such studies are best performed by observing and interviewing professionals, not by controlled experiments.

All the above is from the scientific and experimental point of view. But using students also has ethical aspects. Ethical concerns usually relate to avoiding inflicting any harm on the experimental subject or on society at large, and on informed and voluntary consent to participate in the experiment. This is regulated by Institutional Review Boards (IRBs) [106].

In the context of student participation in software engineering experiments there is no real danger of causing actual harm to the students, but some have voiced concern regarding harm to their academic progress. Therefore, especially when experiments are carried out as part of compulsory classes, they should have educational goals [49, 27]. Examples include the opportunity to learn or exercise some technique or methodology, being exposed to cutting-edge ideas and procedures, creating awareness of difficulties and trade-offs, providing industrial-like experience, and first-hand learning about empirical methods. Evidence that participation in experiments indeed contributes to students' education has been provided by Staron [176].

Still, using students in experiments should be done subject to ethical considerations [167, 49, 52]. For example, could the students have learned the same things more efficiently in some other way? Is it fair and reasonable to grade them on their performance in an experiment, especially if they were divided into groups that used different treatments? Is it reasonable to give academic credit for participation in experiments? Some of these concerns can be mitigated by giving students feedback after the experiment is completed, so they see how their participation added to the knowledge in the field.

The issue of informed consent is also problematic in a classroom setting, as students may refrain from opposing suggestions or requests from their professors [167]. Thus at a minimum one needs to uphold anonymity, and allow the option to opt out, thereby negating the fear of influence on grades. Thus the ethical point of view in this matter coincides with the methodological view that coercion to participate in an experiment may lead to unreliable results — a problem that can occur also in an industrial setting, and is not unique to academia. Indeed, it is in general necessary

not to mix experimental observations with evaluations of performance. Gathered data should not be used to evaluate subjects outside of the study context.

6 Conclusions

Using students in software engineering experiments is often cited as a problem, because students constitute a convenience sample: they are selected for the study because they are easily available to the researcher, and because they are cheap, regardless of whether they are representative of the target population in general. This is a far cry from the ideal of using random sampling, where study participants can be argued to truly represent the target population. As a result many researchers (and reviewers) have reservations about the external validity of student-based experiments, claiming there is no reason to believe that the results generalize beyond the original study population.

Conversely, it has been claimed that in many contexts using student subjects is actually valid. While the students are not a valid representative statistical sample of software professionals, they can be viewed as the next generation of professionals [109, 179]. So students are perfectly suitable when the study does not require a steep learning curve for using new technology [17, 153]. Moreover, using industry professionals or web-based volunteers is usually not any better, because these techniques too cannot guarantee a valid random sample of the general software practitioner population. And in any case there are also wide differences in background, experience, and capabilities among practitioners.

The main conclusions of this review are summarized in Table 3. Taken together, the overarching message is that "can students be used as experimental subjects?" is not the right question. First and foremost, the goal should be that *the observed effect be representative of the real effect*, rather than that the experimental subjects be representative of real developers. But even this is an over-simplification, because it assumes that there is a single real effect. As a research community, we need to embrace variability and collect much more data from diverse conditions.

Reviewing the literature on the subject indicates that "students vs. professionals" is actually a misrepresentation of the confounding effect of proficiency, and in fact differences in performance are much more important than differences in status [175, 172]. It is reasonable to assume that there indeed exists a big difference between complete novices and graduating students, but after three or four semesters students are already reasonable experimental subjects for general studies. It is possible that upon starting employment there is another large increase in capabilities, but this may be more focused on the specific technologies used in the individual place of work. Indeed, a major problem in experimental software engineering is the differences between individual experimental subjects. Such differences suggest the need for a basic proficiency test as part of the experimental setup [79, 163]. But it is not easy to come up with a simple and discriminating test.

Given the difficulty in measuring proficiency, experience is often used as a proxy, under the assumption that more experience is equivalent to higher proficiency. This may be true for relative beginners (students and new professionals), but more senior professionals may reach a plateau where additional experience does not lead to significant additional improvements [174]. Still, it is important to acknowledge that expertise (typically emanating from experience) does in fact lead

Table 3: *Summary of main observations and recommendations.*

1. The level of the experimental subjects (students or otherwise) should be commensurate with the tasks they are expected to perform
2. Graduate students and even students at the end of their BSc have similar proficiency to industrial professionals for general programming tasks
3. In many cases experiments with student subjects lead to the correct relative results, even if they are generally slower or less proficient than professionals
4. Students are naturally suitable when studying novices
5. Subjects with experience with the tools or techniques being studied are required when there is a long learning curve to use them effectively; in some cases adequate training sessions should be provided
6. Studies of how experts tackle complex problems require real experts, but also real problems; performing them is especially difficult, and alternatives such as case studies should be considered
7. When the effect of proficiency is the focus of study, subjects should be assessed individually rather than being assigned by degree or affiliation
8. Academic studies with students can be profitably used in initial steps of larger industrial collaboration research efforts
9. One can compensate for different levels of proficiency by using within-subject experimental designs, provided there is no danger of learning effects
10. Subjects should want to participate in the experiment. Students or professionals who are told to participate may be problematic.
11. Experience is just one factor that may affect proficiency, and it may be important to consider others as well (e.g. sex or personality)
12. When using students as experimental subjects,
    - Student subjects should not be identified and their performance in the experiment should not affect their grades
    - Participation in experiments should not be compulsory for students unless the experiment has clear educational benefits

to different behaviors in some cases. Thus professionals (or experienced developers) are needed when maturity such as in tool usage is potentially part of the setup. Inexperienced students don't use tools as much, even if they could benefit from them more [55].

Special care should be taken when claims about interactions with experience or proficiency are made. One possible problem is confounding proficiency-based filtering with the results, such that subjects identified as less proficient will necessarily perform worse. Another is that claimed results about the effect of experience, e.g. that a tool helps inexperienced developers more, may not be convincing because not enough experienced developers were available to compare with, and it may be the case that the tool helps for everyone [148].

The alternative to using students is to recruit industrial professionals. This may be hard to do: one either has to pay them a competitive fee, or at least schedule experiments so as not to interfere with their normal work schedule [168, 69, 183]. Therefore it is important to make the best use of this scarce resource. One repeated suggestion is to conduct a pilot study first, based on students, and only then move to the "real" study with professionals [169, 49, 85, 18]. Using students first allows to both test and debug the experimental procedure, and to justify the extra effort of using professionals [17, 52, 179].

Another issue is where the experiment is run: maybe conducting it in an industrial setting is even more important than using professionals [168, 72, 69]. But companies are often reluctant to support this for lack of immediate tangible benefits. This is short sighted, as industry can also benefit from participating in studies [49]. Specifically, by allowing employees to participate in academic experiments they obtain early evidence to confirm or refute hypotheses about methodologies and technologies; they learn about new ones; they obtain knowledge of the procedures, costs, and benefits of empirical software engineering; and they learn about their own process and people (in observational/survey studies). One can also view experiments as training [114] or technology transfer [122, 12, 18, 85], and possibly use internship in the company as a vehicle [54], which may later aid in recruitment.

It is important to note that the issue of using students is but one aspect of empirical software engineering studies. There are many other methodological issues that are no less important. One issue that is often problematic is using appropriate statistical tools. This starts with the most basic, for example the need to show full data distributions, or at least box plots, and not make do with averages of widely dispersed data points with skewed or bimodal distributions. It continues with using modern and non-parametric techniques that are not compromised by data which does not fit model assumptions like normality [189, 190].

There is also significant need for more experimentation in general [180, 178]. A literature study of the period 1993–2002 found that controlled experiments papers represent only 1.9% of all 4543 papers that were examined [170]. Indeed, too many results in the literature, which are then accepted as "facts", are actually based on a single experiment in a specific setting with few subjects conducted many years ago. A selection of interesting examples is provided by Glass [84], including the often-quoted 28:1 ratio in performance between the best and worse programmers, which is actually based on a study of debugging performance from 1968 using a grand total of 21 subjects [151].

Discussions and arguments for external validity of individual studies cannot replace actual

experimental evidence. In particular, it is ludicrous to expect any single study to reveal the full picture or even a significant part of the picture of a research topic. Rather, each individual study should be regarded as a pixel. Moreover, at least some of them should be targeted at basic science issues with no immediate practical relevance. The full picture can then emerge when we have enough replications of enough diverse studies — orders of magnitude more than we have now — to step back and observe the underlying currents and revealed patterns.

To obtain the required evidence many more replications are needed, as opposed to branching into innovative uncharted territory. Moreover, different levels of replications should be employed [81, 103, 161]. This includes variations in the experimental artifacts and the approach in addition to using different experimental subjects. Experimental validity can only be obtained by using all of these in tandem.

And finally returning to the issue of students as experimental subjects, our understanding of using different experimental subjects will improve with more empirical work on the effects of experience and expertise, and specifically on what makes highly-performing developers different and how expertise develops or can be promoted [76, 175, 71, 155]. This line of work is more meaningful than comparing students to professionals.

Acknowledgments

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13). Many thanks to Lutz Prechelt for his very useful comments on an earlier draft of this paper, and to anonymous reviewers of another draft.

References

[1] S. Abrahão, C. Gravino, E. Insfran, G. Scanniello, and G. Tortora, "*Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments*". *IEEE Trans. Softw. Eng.* **39(3)**, pp. 327–342, Mar 2013, DOI: 10.1109/TSE.2012.27.

[2] S. T. Acuña, N. Juristo, and A. M. Moreno, "*Emphasizing human capabilities in software development*". *IEEE Softw.* **23(2)**, pp. 94–101, Mar-Apr 2006, DOI: 10.1109/MS.2006.47.

[3] B. Adelson, "*When novices surpass experts: The difficulty of a task may increase with expertise*". *J. Exp. Psych.: Learning, Memory, and Cognition* **10(3)**, pp. 483–495, Jul 1984, DOI: 10.1037/0278-7393.10.3.483.

[4] B. Adelson and E. Soloway, "*The role of domain experience in software design*". *IEEE Trans. Softw. Eng.* **SE-11(11)**, pp. 1351–1360, Nov 1985, DOI: 10.1109/TSE.1985.231883.

[5] A. Ahadi and R. Lister, "*Geek genes, prior knowledge, stumbling points and learning edge momentum: Parts of one elephant?*" In 9th *Intl. Computing Education Research*, pp. 123–128, Aug 2013, DOI: 10.1145/2493394.2493416.

[6] A. Aranda, O. Dieste, and N. Juristo, "*Evidence of the presence of bias in subjective metrics: Analysis within a family of experiments*". In 18th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, art. no. 24, May 2014, DOI: 10.1145/2601248.2601291.

[7] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, "*The impact of UML documentation on software maintenance: An experimental evaluation*". *IEEE Trans. Softw. Eng.* **32(6)**, pp. 365–381, Jun 2006, DOI: 10.1109/TSE.2006.59.

[8] E. Arisholm, H. Gallis, T. Dybå, and D. I. K. Sjøberg, "*Evaluating pair programming with respect to system complexity and programmer expertise*". *IEEE Trans. Softw. Eng.* **33(2)**, pp. 65–86, Feb 2007, DOI: 10.1109/TSE.2007.17.

[9] E. Arisholm and D. I. K. Sjøberg, "*Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software*". *IEEE Trans. Softw. Eng.* **30(8)**, pp. 521–534, Aug 2004, DOI: 10.1109/TSE.2004.43.

[10] C. J. Atman, R. S. Adams, M. E. Cardella, J. Turns, S. Mosborg, and J. Saleem, "*Engineering design processes: A comparison of students and expert practitioners*". *J. Eng. Educ.* **96(4)**, pp. 359–379, Oct 2007, DOI: 10.1002/j.2168-9830.2007.tb00945.x.

[11] S. Bannerman and A. Martin, "*A multiple comparative study of test-with development product changes and their effects on team speed and product quality*". *Empirical Softw. Eng.* **16(2)**, pp. 177–210, Apr 2011, DOI: 10.1007/s10664-010-9137-5.

[12] V. R. Basili, "*The experimental paradigm in software engineering*". In *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pp. 3–12, Springer-Verlag, 1993, DOI: 10.1007/3-540-57092-6_91. Lect. Notes Comput. Sci. vol. 706.

[13] V. R. Basili, L. C. Briand, and W. L. Melo, "*How reuse influences productivity in object-oriented systems*". *Comm. ACM* **39(10)**, pp. 104–116, Oct 1996, DOI: 10.1145/236156.236184.

[14] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz, "*The empirical investigation of perspective-based reading*". *Empirical Softw. Eng.* **1(2)**, pp. 133–164, Jan 1996, DOI: 10.1007/BF00368702.

[15] V. R. Basili and R. W. Selby, "*Comparing the effectiveness of software testing strategies*". *IEEE Trans. Softw. Eng.* **SE-13(12)**, pp. 1278–1296, Dec 1987, DOI: 10.1109/TSE.1987.232881.

[16] V. R. Basili, R. W. Selby, and D. H. Hutchens, "*Experimentation in software engineering*". *IEEE Trans. Softw. Eng.* **SE-12(7)**, pp. 733–743, Jul 1986, DOI: 10.1109/TSE.1986.6312975.

[17] V. R. Basili, F. Shull, and F. Lanubile, "*Building knowledge through families of experiments*". *IEEE Trans. Softw. Eng.* **25(4)**, pp. 456–473, Jul/Aug 1999, DOI: 10.1109/32.799939.

[18] V. R. Basili and M. V. Zelkowitz, "*Empirical studies to build a science of computer science*". *Comm. ACM* **50(11)**, pp. 33–37, Nov 2007, DOI: 10.1145/1297797.1297819.

[19] A. G. Bateson, R. A. Alexander, and M. D. Murphy, "*Cognitive processing differences between novice and expert computer programmers*". *Intl. J. Man-Machine Studies* **26(6)**, pp. 649–660, Jun 1987, DOI: 10.1016/S0020-7373(87)80058-5.

[20] D. Batra and J. G. Davis, "*Conceptual data modelling in database design: Similarities and differences between expert and novice designers*". *Intl. J. Man-Machine Studies* **37(1)**, pp. 83–101, Jul 1992, DOI: 10.1016/0020-7373(92)90092-Y.

[21] P. Bayman and R. E. Mayer, "*A diagnosis of beginning programmers' misconceptions of BASIC programming statements*". *Comm. ACM* **26(9)**, pp. 677–679, Sep 1983, DOI: 10.1145/358172.358408.

[22] K. Beck, "*Embracing change with extreme programming*". *Computer* **32(10)**, pp. 70–77, Oct 1999, DOI: 10.1109/2.796139.

[23] R. Bednarik, N. Myller, E. Sutinen, and M. Tukiainen, "*Effects of experience on gaze behavior during program animation*". In 17th *Workshop of Psychology of Programming Interest Group*, pp. 49–61, Jun 2005.

[24] R. Bednarik and M. Tukiainen, "*An eye-tracking methodology for characterizing porgram comprehension processes*". In 4th *Symp. Eye Tracking Res. & App.*, pp. 125–132, Mar 2006, DOI: 10.1145/1117309.1117356.

[25] A. Begel and N. Nagappan, "*Pair programming: What's in it for me?*" In 2nd *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 120–128, Oct 2008, DOI: 10.1145/1414004.1414026.

[26] A. Begel and B. Simon, "*Novice professionals: Recent graduates in a first software engineering job*". In *Making Software*, A. Oram and G. Wilson (eds.), chap. 26, pp. 495–516, O'Reilly, 2011.

[27] P. Berander, "*Using students as subjects in requirements prioritization*". In *Intl. Symp. Empirical Softw. Eng.*, pp. 167–176, Aug 2004, DOI: 10.1109/ISESE.2004.1334904.

[28] G. R. Bergersen and J.-E. Gustafsson, "*Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective*". *J. Individual Differences* **32(4)**, pp. 201–209, Nov 2011, DOI: 10.1027/1614-0001/a000052.

[29] G. R. Bergersen, J. E. Hannay, D. I. K. Sjøberg, T. Dybå, and A. Karahasanović, "*Inferring skill from tests of programming performance: Combining time and quality*". In 5th *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 305–314, Sep 2011, DOI: 10.1109/ESEM.2011.39.

[30] G. R. Bergersen and D. I. K. Sjøberg, "*Evaluating methods and technologies in software engineering with respect to developer's skill level*". In 16th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, pp. 101–110, May 2012, DOI: 10.1049/ic.2012.0013.

[31] G. R. Bergersen, D. I. K. Sjøberg, and T. Dybå, "*Construction and validation of an instrument for measuring programming skill*". *IEEE Trans. Softw. Eng.* **40(12)**, pp. 1163–1184, Dec 2014, DOI: 10.1109/TSE.2014.2348997.

[32] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "*The impact of identifier style on effort and comprehension*". *Empirical Softw. Eng.* **18(2)**, pp. 219–276, Apr 2013, DOI: 10.1007/s10664-012-9201-4.

[33] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "*To CamelCase or under_score*". In 17th *Intl. Conf. Program Comprehension*, pp. 158–167, May 2009, DOI: 10.1109/ICPC.2009.5090039.

[34] B. Bishop and K. McDaid, "*Spreadsheet debugging behaviour of expert and novice end-users*". In 4th *Intl. Workshop End-User Software Engineering*, pp. 56–60, May 2008, DOI: 10.1145/1370847.1370860.

[35] C. Bishop-Clark and D. D. Wheeler, "*The Myers-Briggs personality type and its relationship to computer programming*". *J. Res. Computing in Education* **26(3)**, pp. 358–370, Spring 1994, DOI: 10.1080/08886504.1994.10782096.

[36] B. W. Boehm, *Software Engineering Economics*. Prentice-Hall, 1981.

[37] R. Bornat, S. Dehnadi, and Simon, "*Mental models, consistency and programming aptitude*". In 10th *Australasian Comput. Education Conf.*, pp. 53–61, Jan 2008.

[38] L. C. Briand, C. Bunse, and J. W. Daly, "*A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs*". *IEEE Trans. Softw. Eng.* **27(6)**, pp. 513–530, Jun 2001, DOI: 10.1109/32.926174.

[39] L. C. Briand, C. Bunse, J. W. Daly, and C. Differding, "*An experimental comparison of the maintainability of object-oriented and structured design documents*". *Empirical Softw. Eng.* **2(3)**, pp. 291–312, Sep 1997, DOI: 10.1023/A:1009720117601.

[40] L. C. Briand, Y. Labiche, M. Di Penta, and H. D. Yan-Bondoc, "*An experimental investigation of formality in UML-based development*". *IEEE Trans. Softw. Eng.* **31(10)**, pp. 833–849, Oct 2005, DOI: 10.1109/TSE.2005.105.

[41] L. C. Briand, J. Wüst, and H. Lounis, "*Replicated case studies for investigating quality factors in object-oriented designs*". *Empirical Softw. Eng.* **6(1)**, pp. 11–58, Mar 2001, DOI: 10.1023/A:1009815306478.

[42] R. E. Brooks, "*Studying programmer behavior experimentally: The problems of proper methodology*". *Comm. ACM* **23(4)**, pp. 207–213, Apr 1980, DOI: 10.1145/358841.358847.

[43] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "*Mental representations constructed by experts and novices in object-oriented program comprehension*". In *Human-Computer Interaction*, pp. 339–346, Jul 1997, DOI: 10.1007/978-0-387-35175-9_55.

[44] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "*Object-oriented program comprehension: Effect of expertise, task and phase*". *Empirical Softw. Eng.* **7(2)**, pp. 115–156, Jun 2002, DOI: 10.1023/A:1015297914742.

[45] R. P. L. Buse, C. Sadowsky, and W. Weimer, "*Benefits and barriers of user evaluation in software engineering research*". In *Object-Oriented Prog. Syst., Lang., & Appl. Conf. Proc.*, pp. 643–656, Oct 2011, DOI: 10.1145/2076021.2048117.

[46] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "*Eye movements in code reading: Relaxing the linear order*". In 23rd *Intl. Conf. Program Comprehension*, May 2015.

[47] L. F. Capretz, "*Personality types in software engineering*". *Intl. J. Human-Comput.* **58(2)**, pp. 207–214, Feb 2003, DOI: 10.1016/S1071-5819(02)00137-4.

[48] L. F. Capretz and F. Ahmed, "*Making sense of software development and personality types*". *IT Professional* **12(1)**, pp. 6–13, Jan-Feb 2010, DOI: 10.1109/MITP.2010.33.

[49] J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "*Issues in using students in empirical studies in software engineering education*". In 9th *Softw. Metrics Symp.*, pp. 239–249, Sep 2003, DOI: 10.1109/METRIC.2003.1232471.

[50] J. Carver, F. Shull, and V. Basili, "*Observational studies to accelerate process experience in classroom studies: An evaluation*". In *Intl. Symp. Empirical Softw. Eng.*, pp. 72–79, Sep 2003, DOI: 10.1109/ISESE.2003.1237966.

[51] J. Carver, J. VanVoorhis, and V. Basili, "*Understanding the impact of assumptions on experimental validity*". In *Intl. Symp. Empirical Softw. Eng.*, pp. 251–260, Aug 2004, DOI: 10.1109/ISESE.2004.1334912.

[52] J. C. Carver, L. Jaccheri, S. Morasca, and F. Shull, "*A checklist for integrating student empirical studies with research and teaching goals*". *Empirical Softw. Eng.* **15(1)**, pp. 35–59, Feb 2010, DOI: 10.1007/s10664-009-9109-9.

[53] J. C. Carver, N. Nagappan, and A. Page, "*The impact of educational background on the effectiveness of requirements inspections: An empirical study*". *IEEE Trans. Softw. Eng.* **34(6)**, pp. 800–812, Nov/Dec 2008, DOI: 10.1109/TSE.2008.49.

[54] A. Čaušević, R. Shukla, and S. Punnekkat, "*Industrial study on test driven development: Challenges and experience*". In 1st *Intl. Workshop Conducting Empirical Studies in Industry*, pp. 15–20, May 2013, DOI: 10.1109/CESI.2013.6618464.

[55] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "*A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques*". *Empirical Softw. Eng.* **19(4)**, pp. 1040–1074, Aug 2014, DOI: 10.1007/s10664-013-9248-x.

[56] P. H. Cheney, "*Effects of individual characteristics, organizational factors and task characteristics on computer programmer productivity and job satisfaction*". *Inf. & Management* **7(4)**, pp. 209–214, Aug 1984, DOI: 10.1016/0378-7206(84)90020-X.

[57] G. Colvin, "*What it takes to be great*". *Fortune* **154(9)**, 30 Oct 2006.

[58] G. Colvin, *Talent Is Overrated: What* Really *Separates World-Class Performers from Everybody Else*. Portfolio, 2008.

[59] M. E. Crosby, J. Scholtz, and S. Wiedenbeck, "*The roles beacons play in comprehension for novice and expert programmers*". In 14th *Workshop Psychology of Programming Interest Group*, pp. 58–73, Jun 2002.

[60] B. Curtis, "*Substantiating programmer variability*". *Proc. IEEE* **69(7)**, p. 846, Jul 1981, DOI: 10.1109/PROC.1981.12088.

[61] B. Curtis, "*By the way, did anyone study any real programmers?*" In 1st *Workshop Empirical Studies of Programmers*, pp. 256–262, 1986.

[62] B. Curtis, "*A career spent wading through industry's empirical ooze*". In 2nd *Intl. Workshop Conducting Empirical Studies in Industry*, pp. 1–2, Jun 2014, DOI: 10.1145/2593690.2593699.

[63] B. Curtis, S. B. Sheppard, E. Kruesi-Bailey, J. Bailey, and D. A. Boehm-Davis, "*Experimental evaluation of software documentation formats*". *J. Syst. & Softw.* **9(2)**, pp. 167–207, Feb 1989, DOI: 10.1016/0164-1212(89)90019-8.

[64] D. Damian, R. Helms, I. Kwan, S. Marczak, and B. Koelewijn, "*The role of domain knowledge and cross-functional communication in socio-technical coordination*". In 35th *Intl. Conf. Softw. Eng.*, pp. 442–451, May 2013, DOI: 10.1109/ICSE.2013.6606590.

[65] M. Daun, A. Salmon, T. Weyer, and K. Pohl, "*The impact of students' skills and experiences on empirical results: A controlled experiment with undergraduate and graduate students*". In 19th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, art. no. 29, Apr 2015, DOI: 10.1145/2745802.2745829.

[66] A. Devito Da Cunha and D. Greathead, "*Does personality matter? an analysis of code-review ability*". *Comm. ACM* **50(5)**, pp. 109–112, May 2007, DOI: 10.1145/1230819.1241672.

[67] E. di Bella, I. Fronza, N. Phaphoom, A. Sillitti, G. Succi, and J. Vlasenko, "*Pair programming and software defects—a large, industrial case study*". *IEEE Trans. Softw. Eng.* **39(7)**, pp. 930–953, Jul 2013, DOI: 10.1109/TSE.2012.68.

[68] M. Di Penta, R. E. K. Stirewalt, and E. Kraemer, "*Designing your next empirical study on program comprehension*". In 15th *Intl. Conf. Program Comprehension*, pp. 281–285, Jun 2007, DOI: 10.1109/ICPC.2007.17.

[69] O. Dieste, N. Juristo, and M. D. Martínez, "*Software industry experiments: A systematic literature review*". In 1st *Intl. Workshop Conducting Empirical Studies in Industry*, pp. 2–8, May 2013, DOI: 10.1109/CESI.2013.6618462.

[70] S. C. dos Santos and F. S. Soares, "*Authentic assessment in software engineering education based on PBL principles*". In 35th *Intl. Conf. Softw. Eng.*, pp. 1055–1062, May 2013, DOI: 10.1109/ICSE.2013.6606655.

[71] S. E. Dreyfus and H. L. Dreyfus, *A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition*. Tech. Rep. ORC-80-2, Operations Research Center, University of California, Berkeley, Feb 1980.

[72] T. Dybå, D. I. K. Sjøberg, and D. S. Cruzes, "*What works for whome, where, when, and why? on the role of context in empirical software engineering*". In *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 19–28, Sep 2012, DOI: 10.1145/2372251.2372256.

[73] W. J. Dzidek, E. Arisholm, and L. C. Briand, "*A realistic empirical evaluation of the costs and benefits of UML in software maintenance*". *IEEE Trans. Softw. Eng.* **34(3)**, pp. 407–432, May/Jun 2008, DOI: 10.1109/TSE.2008.15.

[74] A. Ebrahimi, "*Novice programmer errors: Language constructs and plan composition*". *Intl. J. Human-Computer Studies* **41(4)**, pp. 457–480, Oct 1994, DOI: 10.1006/ijhc.1994.1069.

[75] D. E. Egan, "*Individual differences in human-computer interaction*". In *Handbook of Human-Computer Interaction*, M. Helander (ed.), chap. 24, pp. 543–568, Elsevier Science Publishers B. V. (North-Holland), 1988.

[76] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer, "*The role of deliberate practice in the acquisition of expert performance*". *Psychological Rev.* **100(3)**, pp. 363–406, Jul 1993, DOI: 10.1037/0033-295X.100.3.363.

[77] K. A. Ericsson, M. J. Prietula, and E. T. Cokely, "*The making of an expert*". *Harvard Business Rev.* Jul-Aug 2007.

[78] K. Falkner, C. Szabo, R. Vivian, and N. Falkner, "*Evolution of software development strategies*". In 37th *Intl. Conf. Softw. Eng.*, pp. 243–252, May 2015, DOI: 10.1109/ICSE.2015.153.

[79] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "*Measuring programming experience*". In 20th *Intl. Conf. Program Comprehension*, pp. 73–82, Jun 2012, DOI: 10.1109/ICPC.2012.6240511.

[80] D. G. Feitelson, "*Experimental computer science: The need for a cultural change*". URL http://www.cs.huji.ac.il/~feit/papers/exp05.pdf, 2005.

[81] D. G. Feitelson, "*From repeatability to reproducibility and corroboration*". *Operating Syst. Rev.* **49(1)**, pp. 3–11, Jan 2015, DOI: 10.1145/2723872.2723875.

[82] A. Fernandez, S. Abrahão, and E. Insfran, "*Empirical validation of a usability inspection method for model-driven web development*". *J. Syst. & Softw.* **86(1)**, pp. 161–186, Jan 2013, DOI: 10.1016/j.jss.2012.07.043.

[83] D. Fucci, B. Turhan, and M. Oivo, "*On the effects of programming and testing skills on external quality and productivity in a test-driven development context*". In 19th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, art. no. 25, Apr 2015, DOI: 10.1145/2745802.2745826.

[84] R. L. Glass, *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003.

[85] T. Gorschek, C. Wohlin, P. Garre, and S. Larsson, "*A model for technology transfer in practice*". *IEEE Softw.* **23(6)**, pp. 88–95, Nov/Dec 2006, DOI: 10.1109/MS.2006.147.

[86] D. Graziotin, X. Wang, and P. Abrahamsson, "*Happy software developers solve problems better: Psychological measurements in empirical software engineering*". *PeerJ* **2**, art. no. e289, Mar 2014, DOI: 10.7717/peerj.289.

[87] L. Gugerty and G. M. Olson, "*Comprehension differences in debugging by skilled and novice programmers*". In 1st *Workshop Empirical Studies of Programmers*, pp. 13–27, 1986.

[88] M. Guzdial, "*Why is it so hard to learn to program?*" In *Making Software*, A. Oram and G. Wilson (eds.), pp. 111–124, O'Reilly Media Inc., 2011.

[89] J. E. Hannay, "*Personality, intelligence, and expertise: Impacts on software development*". In *Making Software*, A. Oram and G. Wilson (eds.), pp. 79–110, O'Reilly Media Inc., 2011.

[90] J. E. Hannay and M. Jørgensen, "*The role of deliberate artificial design elements in software engineering experiments*". *IEEE Trans. Softw. Eng.* **34(2)**, pp. 242–259, Mar/Apr 2008, DOI: 10.1109/TSE.2008.13.

[91] A. Heathcote, S. Brown, and D. J. K. Mewhort, "*The power law repealed: The case for an exponential law of practice*". *Psychonomic Bulletin & Review* **7(2)**, pp. 185–207, Jun 2000, DOI: 10.3758/BF03212979.

[92] J. Henrich, S. J. Heine, and A. Norenzayan, "*The weirdest people in the world?*" *Behavioral and Brain Sciences* **33(2-3)**, pp. 61–83, Jun 2010, DOI: 10.1017/S0140525X0999152X.

[93] C. E. Hmelo-Silver, S. Marathe, and L. Liu, "*Fish swim, rocks sit, and lungs breathe: Expert–novice understanding of complex systems*". *J. Learning Sci.* **16(3)**, pp. 307–331, 2007, DOI: 10.1080/10508400701413401.

[94] C. E. Hmelo-Silver and M. G. Pfeffer, "*Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions*". *Cognitive Sci.* **28(1)**, pp. 127–138, Jan-Feb 2004, DOI: 10.1016/S0364-0213(03)00065-X.

[95] 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)**, pp. 201–214, Nov 2000, DOI: 10.1023/A:1026586415054.

[96] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press and Addison-Wesley, 1995.

[97] D. S. Janzen, J. Clements, and M. Hilton, "*An evaluation of interactive test-driven labs with WebIDE in CS0*". In 35th *Intl. Conf. Softw. Eng.*, pp. 1090–1098, May 2013, DOI: 10.1109/ICSE.2013.6606659.

[98] A. Jedlitschka and D. Pfahl, "*Reporting guidelines for controlled experiments in software engineering*". In *Intl. Symp. Empirical Softw. Eng.*, pp. 95–104, Nov 2005, DOI: 10.1109/ISESE.2005.1541818.

[99] R. Jeffries, A. A. Turner, P. G. Polson, and M. E. Atwood, "*The processes involved in designing software*". In *Cognitive Skills and Their Acquisition*, J. R. Anderson (ed.), pp. 255–283, Lawrence Erlbaum Assoc., 1981.

[100] J. Jung, K. Hoefig, D. Domis, A. Jedlitschka, and M. Hiller, "*Experimental comparison of two safety analysis methods and its replication*". In *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 223–232, Oct 2013, DOI: 10.1109/ESEM.2013.59.

[101] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*. Kluwer, 2001.

[102] N. Juristo and S. Vegas, "*Functional testing, structural testing and code reading: What fault type do they each detect?*" In *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*, R. Conradi and A. I. Wang (eds.), pp. 208–232, Springer-Verlag, 2003, DOI: 10.1007/978-3-540-45143-3_12. Lect. Notes Comput. Sci. vol. 2765.

[103] N. Juristo and S. Vegas, "*The role of non-exact replications in software engineering experiments*". *Empirical Softw. Eng.* **16(3)**, pp. 295–324, Jun 2011, DOI: 10.1007/s10664-010-9141-9.

[104] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman, "*Identifying student misconceptions of programming*". In 41st *SIGCSE Tech. Symp. Comput. Sci. Ed.*, pp. 107–111, Mar 2010, DOI: 10.1145/1734263.1734299.

[105] H. Kahney, "*What do novice programmers know about recursion*". In *SIGCHI Conf. Human Factors in Comput. Syst.*, pp. 235–239, Dec 1983, DOI: 10.1145/800045.801618.

[106] J. L. King, "*Humans in computing: Growing responsibilities for researchers*". *Comm. ACM* **58(3)**, pp. 31–33, Mar 2015, DOI: 10.1145/2723675.

[107] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "*Systematic literature reviews in software engineering — a systematic literature review*". *Inf. & Softw. Tech.* **51(1)**, pp. 7–15, Jan 2009, DOI: 10.1016/j.infsof.2008.09.009.

[108] B. Kitchenham, J. Fry, and S. Linkman, "*The case against cross-over design in software engineering*". In 11th *Softw. Technology & Engineering Practice*, pp. 65–67, Sep 2003, DOI: 10.1109/STEP.2003.32.

[109] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "*Preliminary guidelines for empirical research in software engineering*". *IEEE Trans. Softw. Eng.* **28(8)**, pp. 721–734, Aug 2002, DOI: 10.1109/TSE.2002.1027796.

[110] 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 3rd *Workshop Evaluation & Usability of Prog. Lang. & Tools*, pp. 15–24, Oct 2011, DOI: 10.1145/2089155.2089161.

[111] M. Klerer, "*Experimental study of a two-dimensional language vs Fortran for first-course programmers*". *Int. J. Man-Machine Stud.* **20(5)**, pp. 445–467, May 1984, DOI: 10.1016/S0020-7373(84)80021-8.

[112] A. J. Ko, T. D. LaToza, and M. M. Burnett, "*A practical guide to controlled experiments of software engineering tools with human participants*". *Empirical Softw. Eng.* **20(1)**, pp. 110–141, Feb 2015, DOI: 10.1007/s10664-013-9279-3.

[113] L. Kuzniarz, M. Staron, and C. Wohlin, "*Students as study subjects in software engineering experimentation*". In 3rd *Conf. Softw. Eng. Res. & Pract. Sweden*, pp. 19–24, 2003.

[114] O. Laitenberger and J.-M. DeBaud, "*Perspective-based reading of code documents at Robert Bosch GmbH*". *Inf. & Softw. Tech.* **39(11)**, pp. 781–791, 1997, DOI: 10.1016/S0950-5849(97)00030-X.

[115] A. J. Lam, C. M. Lewis, C. L. Lu, I. B. Ornstein, and D. Wang, "*Classifying problems to explain patterns of correlation on the 1988 advanced placement computer science exam*". *J. Computing Sciences in Colleges* **27(4)**, pp. 112–119, Apr 2012.

[116] C. F. J. Lange and M. R. V. Chaudron, "*Effects of defects in UML models – an experimental investigation*". In 28th *Intl. Conf. Softw. Eng.*, pp. 401–411, May 2006, DOI: 10.1145/1134285.1134341.

[117] T. D. LaToza and B. A. Myers, "*Developers ask reachability questions*". In 32nd *Intl. Conf. Softw. Eng.*, vol. 1, pp. 185–194, May 2010, DOI: 10.1145/1806799.1806829.

[118] T. D. LaToza and B. A. Myers, "*Hard-to-answer questions about code*". In *Evaluation & Usability of Prog. Lang. & Tools*, art. no. 8, Oct 2010, DOI: 10.1145/1937117.1937125.

[119] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*. McGraw-Hill, 3rd ed., 2000.

[120] T. C. Lethbridge, "*A survey of the relevance of computer science and software engineering education*". In 11th *Conf. Softw. Eng. Education*, pp. 56–66, Feb 1998, DOI: 10.1109/CSEE.1998.658300.

[121] C. M. Lewis, H. Khayrallah, and A. Tsai, "*Mining data from the AP CS A exam: Patterns, non-patterns, and replication failure*". In 9th *Intl. Computer Education Research Conf.*, pp. 115–122, Aug 2013, DOI: 10.1145/2493394.2493415.

[122] S. Linkman and H. D. Rombach, "*Experimentation as a vehicle for software technology transfer–a family of software reading techniques*". *Inf. & Softw. Tech.* **39(11)**, pp. 777–780, 1997, DOI: 10.1016/S0950-5849(97)00029-3.

[123] H. Liu, X. Guo, and W. Shao, "*Monitor-based instant software refactoring*". *IEEE Trans. Softw. Eng.* **39(8)**, pp. 1112–1126, Aug 2013, DOI: 10.1109/TSE.2013.4.

[124] R. G. Lord and K. J. Maher, "*Alternative information-processing models and their implications for theory, research, and practice*". *Acad. Mgmt. Rev.* **15(1)**, pp. 9–28, Jan 1990, DOI: 10.5465/AMR.1990.4308219.

[125] K. M. Lui and K. C. C. Chan, "*Pair programming productivity: Novice-novice vs. expert-expert*". *Intl. J. Human-Computer Studies* **64(9)**, pp. 915–925, Sep 2006, DOI: 10.1016/j.ijhcs.2006.04.010.

[126] J. Lung, J. Aranda, S. Easterbrook, and G. Wilson, "*On the difficulty of replicating human subjects studies in software engineering*". In 30th *Intl. Conf. Softw. Eng.*, pp. 191–200, May 2008, DOI: 10.1145/1368088.1368115.

[127] L. Ma, J. Ferguson, M. Roper, and M. Wood, "*Investigating the viability of mental models held by novice programmers*". In 38th *SIGCSE Symp. Comput. Sci. Education*, pp. 499–503, Mar 2007, DOI: 10.1145/1227504.1227481.

[128] S. Madison and J. Gifford, "*Modular programming: Novice misconceptions*". *J. Res. Tech. Ed.* **34(3)**, pp. 217–229, 2002, DOI: 10.1080/15391523.2002.10782346.

[129] M. d. A. Maia and R. F. Lafetá, "*On the impact of trace-based feature location in the performance of software maintainers*". *J. Syst. & Softw.* **86(4)**, pp. 1023–1037, Apr 2013, DOI: 10.1016/j.jss.2012.12.032.

[130] M. V. Mäntylä and C. Lassenius, "*What types of defects are really discovered in code reviews?*" *IEEE Trans. Softw. Eng.* **35(3)**, pp. 430–448, May-Jun 2009, DOI: 10.1109/TSE.2008.71.

[131] D. B. Mayer and A. W. Stalnaker, "*Selection and evaluation of computer personnel – the research history of SIG/CPR*". In 23rd *ACM Natl. Conf.*, pp. 657–670, Aug 1968, DOI: 10.1145/800186.810630.

[132] S. McConnell, "*What does 10x mean? measuring variations in programmer productivity*". In *Making Software*, A. Oram and G. Wilson (eds.), pp. 567–573, O'Reilly Media Inc., 2011.

[133] K. B. McKeithen, J. S. Reitman, H. H. Reuter, and S. C. Hirtle, "*Knowledge organization and skill differences in computer programmers*". *Cognitive Pshchol.* **13(3)**, pp. 307–325, Jul 1981, DOI: 10.1016/0010-0285(81)90012-8.

[134] D. A. McMeekin, B. R. von Konsky, M. Robey, and D. J. A. Cooper, "*The significance of participant experience when evaluating software inspection techniques*". In *Australian Softw. Eng. Conf.*, pp. 200–209, Apr 2009, DOI: 10.1109/ASWEC.2009.13.

[135] M. Mărăşoiu, L. Church, and A. Blackwell, "*An empirical investigation of code completion usage by professional software developers*". In 26th *Workshop Psycology of Programming Interest Group*, pp. 71–82, Jul 2015.

[136] G. J. Myers, "*A controlled experiment in program testing and code walkthroughs/inspections*". *Comm. ACM* **21(9)**, pp. 760–768, Sep 1978, DOI: 10.1145/359588.359602.

[137] A. Newell and P. S. Rosenbloom, "*Mechanisms of skill acquisition and the law of practice*". In *Cognitive Skills and Their Acquisition*, J. R. Anderson (ed.), pp. 1–55, Lawrence Erlbaum Assoc., 1981.

[138] D. E. Perry, N. A. Staudenmayer, and L. G. Votta, "*People, organizations, and process improvement*". *IEEE Softw.* **11(4)**, pp. 36–45, Jul 1994, DOI: 10.1109/52.300082.

[139] A. Porter and L. Votta, "*Comparing detection methods for software requirements inspections: A replication using professional subjects*". *Empirical Softw. Eng.* **3(4)**, pp. 355–379, Dec 1998, DOI: 10.1023/A:1009776104355.

[140] A. A. Porter, L. G. Votta, Jr., and V. R. Basili, "*Comparing detection methods for software requirements inspections: A replicated experiment*". *IEEE Trans. Softw. Eng.* **21(6)**, pp. 563–575, Jun 1995, DOI: 10.1109/32.391380.

[141] L. Prechelt, "*Comparing Java vs. C/C++ efficiency differences to interpersonal differences*". *Comm. ACM* **42(10)**, pp. 109–112, Oct 1999, DOI: 10.1145/317665.317683.

[142] L. Prechelt and B. Unger, "*An experiment measuring the effects of Personal Software Process (PSP) training*". *IEEE Trans. Softw. Eng.* **27(5)**, pp. 465–472, May 2001, DOI: 10.1109/32.922716.

[143] S. Reges, "*The mystery of "b := (b = false)"*". In 39th *SIGCSE Tech. Symp. Computer Science Education*, pp. 21–25, Mar 2008, DOI: 10.1145/1352322.1352147.

[144] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, "*How developers' experience and ability influence web application comprehension tasks supported by UML stereotypes: A series of four experiments*". *IEEE Trans. Softw. Eng.* **36(1)**, pp. 96–118, Jan/Feb 2010, DOI: 10.1109/TSE.2009.69.

[145] T. R. Riedl, J. S. Weitzenfeld, J. T. Freeman, G. A. Klein, and J. Musa, "*What we have learned about software engineering expertise*". In *Software Engineering Education*, J. E. Tomayko (ed.), pp. 261–270, Springer-Verlag, 1991, DOI: 10.1007/BFb0024298. Lect. Notes Comput. Sci. vol. 536.

[146] M. P. Robilard, W. Coelho, and G. C. Murphy, "*How effective developers investigate source code: An exploratory study*". *IEEE Trans. Softw. Eng.* **30(12)**, pp. 889–903, Dec 2004, DOI: 10.1109/TSE.2004.101.

[147] A. Robins, "*Learning edge momentum: A new account of outcomes in CS1*". *Comput. Sci. Education* **20(1)**, pp. 37–71, Mar 2010, DOI: 10.1080/08993401003612167.

[148] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej, "*Monitoring user interactions for supporting failure reproduction*". In 21st *Intl. Conf. Program Comprehension*, pp. 73–82, May 2013, DOI: 10.1109/ICPC.2013.6613835.

[149] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "*How do professional developers comprehend software?*" In 34th *Intl. Conf. Softw. Eng.*, pp. 255–265, Jun 2012, DOI: 10.1109/ICSE.2012.6227188.

[150] P. Runeson, "*Using students as experiment subjects – an analysis on graduate and freshmen student data*". In 7th *Intl. Conf. Empirical Assessment in Softw. Eng.*, pp. 95–102, Apr 2003.

[151] H. Sackman, W. J. Erikson, and E. E. Grant, "*Exploratory experimental studies comparing online and offline programming performance*". *Comm. ACM* **11(1)**, pp. 3–11, Jan 1968, DOI: 10.1145/362851.362858.

[152] N. Salleh, E. Mendes, J. Grundy, and G. S. J. Burch, "*An empirical study of the effects of personality in pair programming using the five-factor model*". In 2nd *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 214–225, Oct 2009, DOI: 10.1109/ESEM.2009.5315997.

[153] I. Salman, A. Tosun Misirli, and N. Juristo, "*Are students representative of professionals in software engineering experiments?*" In 37th *Intl. Conf. Softw. Eng.*, pp. 666–676, May 2015, DOI: 10.1109/ICSE.2015.82.

[154] F. Salviulo and G. Scanniello, "*Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals*". In 18th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, art. no. 48, May 2014, DOI: 10.1145/2601248.2601251.

[155] K. D. Schenk, N. P. Vitalari, and K. S. Davis, "*Differences between novice and expert systems analysts: What do we know and what do we do?*" *J. Mgmt. Inf. Syst.* **15(1)**, pp. 9–50, Summer 1998.

[156] D. Senor and S. Singer, *Start-Up Nation*. Twelve, 2009.

[157] Z. Sharafi, A. Marchetto, A. Susi, G. Antoniol, and Y.-G. Guéhéneuc, "*An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension*". In 21st *Intl. Conf. Program Comprehension*, pp. 33–42, May 2013, DOI: 10.1109/ICPC.2013.6613831.

[158] Z. Sharafi, Z. Soh, Y.-G. Guéhéneuc, and G. Antoniol, "*Women and men — different but equal: On the impact of identifier style on source code reading*". In 20th *Intl. Conf. Program Comprehension*, pp. 27–36, Jun 2012, DOI: 10.1109/ICPC.2012.6240505.

[159] B. Sharif and J. I. Maletic, "*An eye tracking study on camelCase and under_score identifier styles*". In 18th *Intl. Conf. Program Comprehension*, pp. 196–205, Jun 2010, DOI: 10.1109/ICPC.2010.41.

[160] B. Shneiderman and R. Mayer, "*Syntactic/semantic interactions in programmer behavior: A model and experimental results*". *Intl. J. Comput. & Inf. Syst.* **8(3)**, pp. 219–238, Jun 1979, DOI: 10.1007/BF00977789.

[161] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "*The role of replications in empirical software engineering*". *Empirical Softw. Eng.* **13(2)**, pp. 211–218, Apr 2008, DOI: 10.1007/s10664-008-9060-1.

[162] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "*Measuring and modeling programming experience*". *Empirical Softw. Eng.* **19(5)**, pp. 1299–1334, Oct 2014, DOI: 10.1007/s10664-013-9286-4.

[163] J. Siegmund and J. Schumann, "*Confounding parameters on program comprehension: A literature survey*". *Empirical Softw. Eng.* **20(4)**, pp. 1159–1192, Aug 2015, DOI: 10.1007/s10664-014-9318-8.

[164] J. Siegmund, N. Siegmund, and S. Apel, "*Views on internal and external validity in empirical software engineering*". In 37th *Intl. Conf. Softw. Eng.*, pp. 9–19, May 2015, DOI: 10.1109/ICSE.2015.24.

[165] Simon et al., "*The ability to articulate strategy as a predictor of programming skill*". In 8th *Australasian Comput. Education Conf.*, pp. 181–188, Jan 2006.

[166] H. A. Simon and W. G. Chase, "*Skill in chess*". *American Scientist* **61(4)**, pp. 394–403, Jul-Aug 1973.

[167] J. Singer and N. G. Vinson, "*Ethical issues in empirical studies of software engineering*". *IEEE Trans. Softw. Eng.* **28(12)**, pp. 1171–1180, Dec 2002, DOI: 10.1109/TSE.2002.1158289.

[168] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanovic, E. F. Koren, and M. Vokác, "*Conducting realistic experiments in software engineering*". In *Intl. Symp. Empirical Softw. Eng.*, pp. 17–26, Oct 2002, DOI: 10.1109/ISESE.2002.1166921.

[169] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, and M. Vokáč, "*Challenges and recommendations when increasing the realism of controlled software engineering experiments*". In *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*, R. Conradi and A. I. Wang (eds.), pp. 24–38, Springer-Verlag, 2003, DOI: 10.1007/978-3-540-45143-3_3. Lect. Notes Comput. Sci. vol. 2765.

[170] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanović, N.-K. Liborg, and A. C. Rekdal, "*A survey of controlled experiments in software engineering*". *IEEE Trans. Softw. Eng.* **31(9)**, pp. 733–753, Sep 2005, DOI: 10.1109/TSE.2005.97.

[171] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "*Quantifying the effect of code smells on maintenance effort*". *IEEE Trans. Softw. Eng.* **39(8)**, pp. 1144–1156, Aug 2013, DOI: 10.1109/TSE.2012.89.

[172] Z. Soh, Z. Sharafi, B. Van den Plas, G. Cepeda Porras, Y.-G. Guéhéneuc, and G. Antoniol, "*Professional status and expertise for UML class diagram comprehension: An empirical study*". In 20th *Intl. Conf. Program Comprehension*, pp. 163–172, Jun 2012, DOI: 10.1109/ICPC.2012.6240484.

[173] E. Soloway and K. Ehrlich, "*Empirical studies of programming knowledge*". *IEEE Trans. Softw. Eng.* **SE-10(5)**, pp. 595–609, Sep 1984, DOI: 10.1109/TSE.1984.5010283.

[174] S. Sonnentag, "*Expertise in professional software design: A process study*". *J. App. Psychol.* **83(5)**, pp. 703–715, Oct 1998, DOI: 10.1037/0021-9010.83.5.703.

[175] S. Sonnentag, C. Niessen, and J. Volmer, "*Expertise in software design*". In *The Cambridge Handbook of Expertise and Expert Performance*, K. A. Ericsson, N. Charness, P. J. Feltovich, and R. R. Hoffman (eds.), pp. 373–387, Cambridge University Press, 2006.

[176] M. Staron, "*Using students as subjects in experiments – a quantitative analysis of the influence of experimentation on students' learning process*". In 20th *Conf. Softw. Eng. Education & Training*, pp. 221–228, Jul 2007, DOI: 10.1109/CSEET.2007.56.

[177] M. Svahnberg, A. Aurum, and C. Wohlin, "*Using students as subjects — an empirical evaluation*". In 2nd *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 288–290, Oct 2008, DOI: 10.1145/1414004.1414055.

[178] W. F. Tichy, "*Should computer scientists experiment more?*" *Computer* **31(5)**, pp. 32–40, May 1998.

[179] W. F. Tichy, "*Hints for reviewing empirical work in software engineering*". *Empirical Softw. Eng.* **5(4)**, pp. 309–312, Dec 2000, DOI: 10.1023/A:1009844119158.

[180] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz, "*Experimental evaluation in computer science: A quantitative study*". *J. Syst. & Softw.* **28(1)**, pp. 9–18, Jan 1995, DOI: 10.1016/0164-1212(94)00111-Y.

[181] R. T. Turley and J. M. Bieman, "*Competencies of exceptional and nonexceptional software engineers*". *J. Syst. & Softw.* **28(1)**, pp. 19–38, Jan 1995, DOI: 10.1016/0164-1212(94)00078-2.

[182] U. van Heesch, P. Avgeriou, and A. Tang, "*Does decision documentation help junior designers rationalize their decisions? a comparative multiple-case study*". *J. Syst. & Softw.* **86(6)**, pp. 1545–1565, Jun 2013, DOI: 10.1016/j.jss.2013.01.057.

[183] S. Vegas, O. Dieste, and N. Juristo, "*Difficulties in running experiments in the software industry: Experiences from the trenches*". In 3rd *Intl. Workshop Conducting Empirical Studies in Industry*, pp. 3–9, May 2015, DOI: 10.1109/CESI.2015.8.

[184] L. G. Votta and A. Porter, "*Experimental software engineering: A report on the state of the art*". In 17th *Intl. Conf. Softw. Eng.*, pp. 277–279, Apr 1995, DOI: 10.1145/225014.225040.

[185] G. S. Walia and J. C. Carver, "*Using error abstraction and classification to improve requirements quality: Conclusions from a family of four empirical studies*". *Empirical Softw. Eng.* **18(4)**, pp. 625–658, Aug 2013, DOI: 10.1007/s10664-012-9202-3.

[186] T. Walle and J. E. Hannay, "*Personality and the nature of collaboration in pair programming*". In 3rd *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 203–213, Oct 2009, DOI: 10.1109/ESEM.2009.5315996.

[187] M. Weiser and J. Shertz, "*Programming problem representation in novice and expert programmers*". *Intl. J. Man-Machine Studies* **19(4)**, pp. 391–398, Oct 1983, DOI: 10.1016/S0020-7373(83)80061-3.

[188] S. Wiedenbeck, "*Novice/expert differences in programming skills*". *Intl. J. Man-Machine Studies* **23(4)**, pp. 383–390, Oct 1985, DOI: 10.1016/S0020-7373(85)80041-9.

[189] R. Wilcox, *Introduction to Robust Estimation & Hypothesis Testing*. Academic Press, 3rd ed., 2012.

[190] R. R. Wilcox, *Fundamentals of Modern Statistical Methods: Substantially Improving Power and Accuracy*. Springer, 2nd ed., 2010.

[191] A. Yamashita and L. Moonen, "*Exploring the impact of inter-smell relations on software maintainability: An empirical study*". In 35th *Intl. Conf. Softw. Eng.*, pp. 682–691, May 2013, DOI: 10.1109/ICSE.2013.6606614.