Empir Software Eng (2015) 20:110–141
DOI 10.1007/s10664-013-9279-3

A practical guide to controlled experiments of software engineering tools with human participants

Andrew J. Ko · Thomas D. LaToza · Margaret M. Burnett

Published online: 27 September 2013
© Springer Science+Business Media New York 2013

Abstract Empirical studies, often in the form of controlled experiments, have been widely adopted in software engineering research as a way to evaluate the merits of new software engineering tools. However, controlled experiments involving *human participants* actually *using* new tools are still rare, and when they are conducted, some have serious validity concerns. Recent research has also shown that many software engineering researchers view this form of tool evaluation as too risky and too difficult to conduct, as they might ultimately lead to inconclusive or negative results. In this paper, we aim both to help researchers minimize the risks of this form of tool evaluation, and to increase their quality, by offering practical methodological guidance on designing and running controlled experiments with developers. Our guidance fills gaps in the empirical literature by explaining, from a practical perspective, options in the recruitment and selection of human participants, informed consent, experimental procedures, demographic measurements, group assignment, training, the selecting and design of tasks, the measurement of common outcome variables such as success and time on task, and study debriefing. Throughout, we situate this guidance in the results of a new systematic review of the tool evaluations that were published in over 1,700 software engineering papers published from 2001 to 2011.

Keywords Research methodology · Tools · Human participants · Human subjects · Experiments

Communicated by: Premkumar Thomas Devanbu

A. J. Ko (✉)
University of Washington, Seattle, WA, USA
e-mail: ajko@uw.edu

T. D. LaToza
University of California, Irvine, Irvine, CA, USA
e-mail: tlatoza@ics.uci.edu

M. M. Burnett
Oregon State University, Corvallis, OR, USA
e-mail: burnett@eecs.oregonstate.edu

1 Introduction

Over the past three decades, empirical studies have become widely accepted as a way to evaluate the strengths and weaknesses of software engineering tools (Zannier et al. 2006; Basili et al. 1986; Basili 1993, 2007; Rombach et al. 1992; Fenton 1993; Tichy et al. 1995; Basili 1996; Lott and Rombach 1996; Tichy 1998, Sjøberg et al. 2007). Software engineering researchers now perform an impressively diverse array of evaluations, including applications of tools against large software repositories, observations of tools in use in the field, surveys and interviews with developers who use the tools in practice, detailed experiences of applying the tools to real programs, and of course, controlled experiments of tools' effects on human performance such as speed, correctness, and usefulness.

However, this last kind of evaluation, the controlled (quantitative) experiment[1] with human participants using the tools, is still rare. In a study of 5,453 articles published between 1993 and 2002, Sjøberg et al. (2005) found only 103 such studies; similarly, in a review of 628 papers, Ramesh et al. (2004) found that only 1.8 % were experiments with human participants. Other reviews have found that many of these studies contain omissions that make their results difficult to interpret and apply to practice, such as the involvement of non-representative participants and the omission of key methodological details about the experimental setting and the population sampled (Zannier et al. 2006; Dieste et al. 2011; Kampenes et al. 2007; Glass et al. 2002; Hannay et al. 2007; Sjøberg et al. 2005). This suggests that there is still a need for more high quality experiments that rigorously evaluate the use of new tools, to supplement other forms of evidence available through non-experimental methods.

Recent work has investigated the scarcity of these experiments with human participants, finding that many software engineering researchers view them as too difficult to design, too time consuming to conduct, too difficult to recruit for, and with too high a risk of inconclusive results (Buse et al. 2011). In our own discussions with software engineering researchers about these issues, many have also pointed to the lack of practical guidance on how to design experiments in ways that minimize these problems. Methodological literature instead focuses on other important but more general issues, providing conceptual background on experiments, measurement, and research design (e.g., Wohlin et al. 2000; Easterbrook et al. 2008; Shull et al. 2006; Juristo and Moreno 2001; Kitchenham et al. 2002). Thus, although these resources are important, more guidance is necessary to help overcome the complexities of designing experiments that involve developers using a tool.

To address this gap, this paper provides the reader with a compendium of alternatives for many of the challenges that arise in designing quantitative controlled experiments with software developers, from recruiting and informed consent, to task design and the measurement of common outcome variables such as time and success on task. Our discussion of each of these topics focuses not on the "best" choices—every experiment has limitations—but on the range of choices and how each choice affects the risk of obtaining inconclusive results. Throughout our guide, we draw upon experiments published in the past decade, which we identified through a systematic review of 345 tool evaluations with human participants reported in 1,701 articles published from 2001 to 2011. We discuss the norms in this literature not to suggest that current practices in evaluation are best practices, but to surface the inherent tensions that any study design must make, how researchers currently balance

[1] Strictly speaking, an experiment is by definition quantitative [Basili 2007]. Other kinds of empirical studies are not technically experiments. Thus, in this paper, when we refer to experiments, we mean quantitative experiments.

them, and in many cases, how researchers are still struggling to find an appropriate balance. Our hope is that this guide, supplemented by this discussion of current evaluation practices, provides practical and actionable methodological guidance that, combined with more general resources on empirical software engineering methods, can prepare the reader to conduct useful, valid, and successful experimental evaluations of tools with developers.

The outline of this paper is as follows. First, we provide an overview of the design of a canonical tool evaluation experiment with human participants and discuss factors that can contribute to inconclusive results. We then discuss each of the major components of this design, the range of choices for each component, and their tradeoffs. We end with a short discussion situating controlled experiments with developers in the broader context of empirical evaluations, and with an appendix, detailing our systematic review of the past decade of tool evaluations.

2 Designing Tool Evaluation Experiments

Before we begin our discussion of quantitative controlled experiments, it is first important to note that before choosing a research method (experimental or otherwise), it is necessary to first clearly define the research question that is driving the tool evaluation. After all, there are many research questions that cannot be easily answered with quantitative experiments. For example, consider the evaluation of a new debugging tool. How do developers use it? What concepts must they understand to use it successfully? How long does it take to learn? What kinds of defects is it most useful for? What kinds of defects is it not useful for? Experiments, as inherently comparative methods, cannot easily answer such questions. Instead, these questions are more easily investigated with non-experimental quantitative or qualitative methods. Moreover, answers to these formative questions are often necessary to obtain before one knows which comparisons, which tasks, which developers, and which measures even make sense for an experiment (LaToza and Myers 2011a, b). Some researchers in other fields such as HCI have even argued that the expectation of experimental evaluation before publishing a new tool can be a hindrance to innovation, disincentivizing researchers from gathering richer and more useful experiences with a technology through the use of qualitative methods (Greenberg and Buxton 2008).

That said, there are research questions that an experiment is best suited to answer. Does a new debugging tool increase the speed with which a developer can localize a fault? Does it lead to improved understanding of a program's implementation? Does it better facilitate collaborative debugging in software teams? If one is ready to answer these questions, there are many methods that can answer these questions (case studies, field deployments, experiments, etc.). Quantitative controlled experiments allow isolation of variables through control: more than other methods, experiments provide the most rigorous evidence that it is the isolated variable(s), such as the particular tool, and not some other factor such as a developer's experience or real-world circumstances, that led to a difference in some software engineering outcome. The tradeoff for this control, as we discuss throughout the rest of this article, is artificiality. In essence, every empirical method has strengths and weaknesses; that is why researchers often use a variety of empirical methods, both experimental and non-experimental qualitative evaluation (Murphy et al. 1999; Holmes and Walker 2013).

Given that a researcher has chosen a controlled experiment to answer a research question, let us consider Fig. 1, which portrays a common design for a tool evaluation experiment with human participants. In the figure, the circles represent human participants, with the white circles representing those who eventually participate in the study. This experiment design

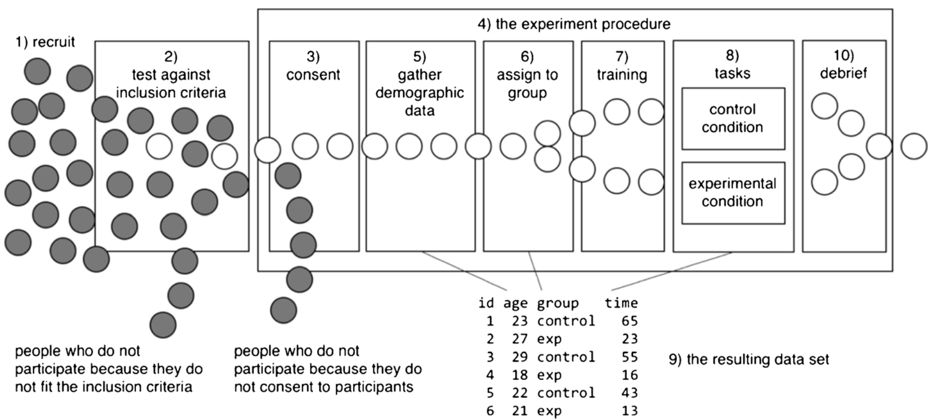

Fig. 1 A canonical design for a tool evaluation experiment with two conditions and a set of tasks. The *circles* represent human participants; the *white circles* are those that satisfy the inclusion criteria. This design includes one independent variable and two conditions. The resulting data set is listed at the bottom

includes one independent variable with two conditions: a *control* group that uses some baseline tool and an *experimental* group that uses a research tool. A key property of this experimental design is that every participant receives the *exact* same materials, instruction, tasks, and environment, *except* for which tool they receive. If there is a difference in the outcomes between the two groups, one can then make a strong claim that this was due to the research tool and nothing else. (Of course, providing the exact same materials is often not possible without harming validity: for example, comparing a new IDE to an existing IDE by stripping all of the features from the existing IDE that are not present in the new one would be disorienting to users of the existing IDE and an unrealistic comparison, as occurred in Ko and Myers (2009). An experimental design is therefore partly a compromise between realism and control).

Designing a tool evaluation experiment involves several key components (each discussed in later sections):

1. **Recruitment** (Figure 1.1, Section 3). Marketing materials, such as an e-mail, poster, or advertisement, are commonly used to entice people to participate.
2. **Selection** (Figure 1.2, Section 3). *Inclusion criteria* are used to filter potential participants and determine if they are part of the intended population. For example, a tool intended for web developers might set inclusion criteria to be people with at least a year of experience in both JavaScript and PHP.
3. **Consent** (Figure 1.3, Section 4.1). If a researcher is at an academic institution, they are likely required to first obtain some form of human subjects approval that evaluates the ethical aspects of the experiment design. Researchers must complete and submit forms describing the experiment in detail, including a consent form that gives participants information about the study to help them decide whether to participate.
4. **Procedure** (Figure 1.4, Section 4). The procedure for the experiment determines what a participant will do from the beginning of an experiment to its end. The design of the procedure should include a script detailing what the experimenter will (and will not) say throughout the experiment, or alternatively, written materials that a participant will read.
5. **Demographic measurements** (Figure 1.5, Section 4.2). Surveys and interviews are common ways of collecting and measuring demographic variables. This data can be

gathered before or after a task, or even as part of testing a potential participant against inclusion criteria.

6. **Group assignment** (Figure 1.6, Section 4.3). Assignment of participants to groups is often done randomly, to distribute variation in participant behavior across conditions evenly, but there are alternatives.
7. **Training** (Figure 1.7, Section 4.4). It is often necessary to teach participants how to use a tool before they use it in the tasks, so that they can use it at a level appropriate for the study's research questions. This can come before or after assigning a participant to a group, depending on whether the training materials differ by condition.
8. **Tasks** (Figure 1.8, Section 5). The researcher needs to choose one or more tasks for participants to perform with the tool they receive.
9. **Outcome measurements** (Figure 1.9, Section 6). Examples of outcome variables include task success, task completion time, defects found, etc. Each outcome variable chosen will need a method for measuring it.
10. **Debrief and compensate** (Figure 1.10, Section 7). Studies typically (but optionally) end with a debriefing, helping participants better understand the purpose of the study and telling participants the answers to the tasks they attempted. The end of the study is also an appropriate time to compensate participants for their time.

While this may seem like a long list, most must first be described to receive human subjects approval (as explained in Section 4.1) and will ultimately need to be described in the "method" section of a research paper. In fact, one strategy for designing an experiment is to draft the human subjects approval form and method section *first*, as a draft of an experimental design. This can help to identify flaws and uncover details that have yet to be defined.

The ultimate, primary goal of conducting an experiment is to determine if a change in an independent variable (e.g., condition) causes a change in a dependent variable (e.g., outcome measurement). One goal of experimental design is thus to maximize the likelihood that an experiment will be able to measure such changes. There are several approaches to achieving this goal:

- *Increase the effectiveness of the tool*. Obviously, the more profound the effect of the tool, the less risk the experiment has of masking the true benefits of the tool because of noise and variation, the fewer observations one needs, and the less one will have to perfect the experimental materials. For particularly effective tools, it can be possible to see significant differences with a small number of participants in each experimental condition (e.g., LaToza and Myers (2011a, b) found significant differences with just 6 participants per condition). Clearly, this is the ideal way to reduce the costs and risks of an experiment, but not always one that is easiest to achieve.
- *Increase the number of participants*. With enough observations, nearly any legitimate difference between experimental conditions can be observed statistically (although not all such differences are meaningful (Kaptein and Robertson 2012). In Section 3, we discuss strategies for recruiting developers, to give the experiment the best chance of observing a difference, if one exists.
- *Decrease extraneous variation in the outcome variables*. Variation in the measurement of outcome variables is the primary reason for negative experimental results: the noisier the signal, the more difficult it is to observe underlying differences in the signal. However, some of this variation is natural and should be preserved; developers, for example, have different knowledge, skills, and ability to learn to a new tool. The "unnatural" variation, or *extraneous* variation, is the variation to eliminate. This includes

anything that would not normally occur in the world and is instead due to the artificiality of the experiment. For example, the developers need to understand the tasks and instructions given them; they should, as a group, have a comparable understanding of the tools being tested; and the study's measurements must be well defined and consistently measured. When participants have varying understanding, they behave inconsistently, making it more difficult to know if their performance varied because of the tool or something else. We discuss sources of extraneous variation throughout the rest of this paper.

In the remaining sections of this paper, we discuss each of the experiment design elements in Fig. 1 in detail, identifying various design choices and strategies, with a particular emphasis on minimizing the risk of undue negative results.

3 Recruiting

One perceived barrier to tool evaluation experiments is recruiting developers (Buse et al. 2011). Here we discuss three challenges in recruiting: (1) finding people who actually represent the intended users of the tool, (2) convincing enough to participate that the statistical comparison will have enough power to reveal any differences that occur, and (3) dealing with the large variation in skill among developers. In this section, we provide guidance for each.

3.1 Deciding Who to Recruit

Recruiting rests on the *inclusion criteria* for the study (Figure 1.2), which are the attributes that participants must have to participate in a study. More generally, good inclusion criteria reflect the characteristics of the people that a researcher believes would benefit from using the research tool, and the work contexts in which the researcher believes the tool would be useful (Murphy-Hill et al. 2010), such as the type of software being developed, the type of organization developing it, and the types of processes being used.

Ideally, inclusion criteria capture the most important attributes that one expects a tool's users to have, such as their skills, motivations, and work practices. For example, when Nimmer and Ernst (2002) wanted to compare a static and dynamic checker, they explicitly focused on supporting experienced Java developers that had no experience with static and dynamic checkers. Their hypothesis was that these developers would quickly learn how to use these checkers to succeed on a set of annotation tasks, more so than developers without the tools. Therefore, they restricted their participants to people with no prior experience with ESC/Java, Houdini, and Daikon (the tools being studied), but with substantial Java programming experience. They also decided to allow graduate students at the researchers' universities to participate, since they fit these criteria. Nimmer and Ernst also had to develop reliable ways of checking whether a participant fit these criteria: to check tool experience, they simply asked a participant if they were familiar with each tool; for Java programming experience, they asked how many years they had been writing Java code. They then informally used these metrics to exclude students without the desired level of experience.

There are several common inclusion criteria for tool evaluation experiments:

- **Programming language experience.** A tool might expect developers to have expertise with a particular programming language. Recent studies have found that self-estimation of language experience on a scale of 1 to 10 correlates moderately with performance on

programming tasks (Feigenspan et al. 2012), at least for undergraduate students. (There has yet to be research on how to measure the expertise of more experienced programmers).

- **Experience with technologies related to the tool.** Whether participants can learn to use a tool might depend a lot on their prior experience with similar tools. This experience can be measured with simple ordinal ratings of experience levels (e.g., “no experience”, “some experience”, “significant experience”).
- **Industry experience.** This is typically an indicator for multiple kinds of skills and knowledge, such as knowledge of version control systems, experience working on a team, and expertise with languages. It may be more instructive to just ask about the specific competencies that are relevant to the tool being evaluated, rather than asking about overall industry experience.
- **(Natural) language proficiency**. Non-native speakers often have a difficult time understanding experiment instructions and tool features; they may also be reluctant to admit confusion. These sources of variation may be worth eliminating from a study by recruiting only participants with many years experience in the natural language used by a tool.

Other common demographic variables such as age and gender might be useful to help characterize the recruited participants in a publication, but they are often less relevant for deciding whether someone is an intended user of a tool, since these attributes often have little to do with a person’s level of skill.

The criteria must also be valid. For example, in our systematic review of software engineering tool evaluation studies from the past decade (detailed in the Appendix), we found that 62 % of studies evaluating a tool actually involved the tool’s inventors themselves using the tool and reporting on their personal experiences (see Fig. 2 for these results by year). These inclusion criteria are flawed from an evaluation perspective, as the tool designers themselves would be far more likely to use their own tool successfully than someone new to the tool. In studies like these, *confirmation bias* (Nickerson 1998) is at play, as the inventors might give undue attention to tasks and results that confirm their beliefs about the benefits of a tool, while unconsciously overlooking evidence about the tool’s weaknesses. In general, inventors who evaluate a tool by using it themselves might provide useful insights about the *applicability* of a tool, but little about whether someone else would benefit from it in practice.

A long-standing debate on inclusion criteria is whether computer science students should be eligible to participate in tool evaluation studies (Carver et al. 2003; Beringer 2004). In a review of 113 experiments published from 1993 to 2002, 72 % of experiments exclusively involved students (Sjøberg et al. 2005). In our review of the most recent decade of studies, we only found 23 % of studies reporting the use of students. (Of the remaining studies, 56 % of studies recruited one or more participants who were software professionals of some kind

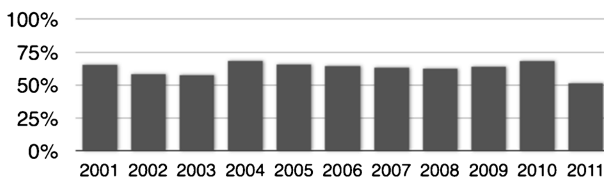

Fig. 2 Proportion of evaluations involving human participants in which the authors themselves were the study participants

and 23 % provided no detail about who the participants were at all, only describing them as "experienced programmers" of some kind.) Therefore, the field is clearly shifting away from the use of students in studies.

The rationale for this shift has primarily to do with programming expertise, as undergraduates, and many graduate students, are typically too inexperienced to be representative of a tool's intended user. Students may also pose other validity concerns, such as the *subject-expectancy effect*, in which participants who expect a certain outcome behave in a way that ensures it. For example, Nimmer and Ernst (2002) recruited graduate students from their own institution. Even though they were careful to state that these students were not from their research group, these students may still have been more motivated to learn how to use the dynamic checkers because of the inherent power imbalance between faculty and students (there is no evidence of this point in software engineering research, but it has been widely documented in behavioral sciences involving faculty student dynamics (Rosenthal and Rubin 1978)). The *observer-expectancy* effect (Rosenthal 1966) may also be at play when involving students, where a researcher's interest in a positive outcome may cause them to unconsciously influence the behavior of the students in an experiment, because the students may view the researchers as having a better understanding of how the tool works than they do. Professional software developers may be less subject to these effects, because they may view the researchers as having less authority or a similar level of expertise.

That said, students can be appropriate participants when their knowledge, skills, and experiences fit within a tool's intended user population. There are also many ways to minimize the biases that students may introduce. For example, many students that have returned for master's degrees in business, computer science, and other fields have many years of software development experience. These students may even be more consistent in their behavior, since they may have a common educational background, helping to minimize variation in their performance on tasks. One can also mitigate subject-expectancy effects by presenting the experiment in a way that suggests that the researcher has no stake in the outcome. For example, if the experiment compares Tool A against Tool B, but the experimenter presents the two tools in a way that makes them appear to be *other* researchers' tools, the participants will be less subject to biases. This, of course, requires that the graduate students do not already know about the tool being developed by that researcher.

Another challenge with recruiting is the variability in software developer skill (Boehm and Papaccio 1988). Such variability is often a challenge in experiments with small sample sizes, as it can require a large number of samples to overcome the resulting variation in developer performance. Of course, this variation is not necessarily something to avoid: if a tool is really designed for developers of all skill levels in a particular programming language, then it is appropriate to evaluate the tool with developers of varying skill. In this case, skill can simply be a factor that is used to interpret the results of a study, providing qualitative insights into how well the tool works for developers with different skill levels. On the other hand, if a tool is intended for developers with a particular level of skill, one can attempt to recruit only developers with that level of skill. As mentioned earlier, Feigenspan et al. 2012 have found that self-estimation of language experience on a scale of 1 to 10 correlates moderately with performance on programming tasks for undergraduate students. The same measure may also work for more experienced developers, but there has yet to be research to validate this. Other approaches to measuring skill include having developers take skills assessment, such as the programming tests often employed in software development job interviews.

Although all of our recruiting discussion up to this point has concerned individuals, many tools are meant to be used in the context of a team. There is unfortunately very little

guidance on how to perform experiments in team contexts, let alone how to recruit for them. In fact, in our systematic literature review, we did not find a single paper that attempted an experiment in the context of a team, let alone an experiment that actually recruited multiple teams to use a new software engineering tool. How to conduct such experiments is an open question, deserving of new research methodologies. Until these exist, the most feasible methods for studying tools used by teams are likely qualitative field evaluations that have teams temporarily adopt a new tool and involve direct observation, logging, and other data gathering techniques to attempt to characterize the effect of the tool's adoption on the team.

3.2 Deciding How Many Participants to Recruit

From a statistical perspective, the more participants one recruits, the lower the chances that legitimate differences will be masked by variation. Studies in our systematic review had a large range of participants, from 1 to 2,600 (the latter was a field deployment) with a median of 10. Experiments recruited a median of 36 participants, ranging from 2 to 222, with a median of 18 participants per condition. Therefore, previously published experiments typically have much smaller numbers of participants than other kinds of studies. While approximately 20 participants per condition are not guaranteed to be a large enough sample to overcome variation in measurements, it is a reasonable number to start with, provided that appropriate statistical tests are used.

A more principled way to decide how many participants to recruit is to do a prospective *power analysis* (e.g., Dybå et al. 2006), which gives an estimate of the sample size required to detect a difference between experimental conditions. The exact calculation of this estimate depends on the specific statistical test used to compare the groups, but it generally requires three inputs: (1) the expected *effect size* (the difference in the outcome variable between groups), (2) the expected variation in this outcome measurement, and (3) the Type I error rate α (typically .05 in software engineering). The first two must be estimated. One source of these estimates is to use data from previous experiments on the tool or even pilot studies (discussed later in Section 8). There are also standard approaches for estimating sample size and effect size such as Cohen's *d*, odds ratio, and Abelson's Causal Efficacy Ratio. Breaugh (2003) provides an accessible introduction to these topics.

3.3 Doing the Recruiting

Once inclusion criteria have been defined, recruiting participants is the next challenge. Recruiting is essentially a marketing problem of getting the attention of a set of people with certain characteristics. The goal is therefore to (1) find out where they put their attention and (2) entice them to participate.

If students have been deemed a valid and representative population to recruit from, recruitment can be straightforward if a researcher is at a university, as there are many ways to reach students (e.g., announcing something in class, sending an announcement to an e-mail list, placing flyers in buildings frequented by students, and so on). Enticing students to participate is also straightforward: small amounts of compensation are often motivating and students may have an intrinsic interest in learning about a research project.

In contrast to students, software professionals can be challenging to recruit, partly because they are so busy. If a researcher is part of a software company, this is easier, as participants may be more invested in helping someone part of the same organization and the industry researcher may have a better understanding of developers concerns and interests. If a researcher is in academia, there are many ways to recruit professionals that have been

successful in the past (Table 1). For example, most of the studies from our systematic review leveraged existing relationships with industry researchers or professional software developers. Some studies had graduate students who were temporary interns at companies, who could recruit at the companies while they were interns. Other authors mentioned having a personal relationship to a person in a company who could help them recruit. Other researchers established formal partnerships with companies to evaluate tools and techniques, such as contract software development work, allowing them to work with established teams.

Recruiting from software companies is not the only way to find participants with industry experience. For academics, many universities staff software teams to develop in-house software solutions for university administration or to help faculty and staff manage courses and students. Not only are these teams nearby, but they also often have an explicit responsibility to support the university, possibly by participating in experiments. The developers on these teams may also have connections into the larger community of local developers.

Many cities also have vibrant communities related to software engineering, many of which maintain a web presence via Meetup.com, public mailing lists, or Facebook groups. Attending these groups' face-to-face meetings can help develop social inroads into the local software industry. They may also be a place to advertise and recruit for experiment participation.

Another source of prospective participants is an alumni database of recent graduates. For example, the first author's school maintains a LinkedIn group with nearly all of the alumni of his school's undergraduate program. Not only does this allow him to message all previous alumni who are still in the local community, but it also allows him to filter his search by developers working in specific software industries, at specific companies, or whom have specific software development skills listed on their résumés. These features allow him to control for variation in participants' backgrounds. The fact that alumni are former students provides an additional incentive them to participate, as many are interested in supporting their former institution.

One more source of potential participants is online labor markets. In other disciplines (Kittur et al. 2008), a popular example is Amazon's Mechanical Turk (mTurk), which is a site that allows "requesters" to post web-based tasks and "workers" to complete them for pay. Studies have shown that the workers on mTurk are primarily North American and Indian, that most are young, educated males and that up to a third even have graduate degrees (Ross et al. 2010); most, however, do not have programming experience. Other options are freelance developers who look for jobs on sites such as oDesk.com, Freelancer.com, Topcoder.com, Craigslist, and other sites that host consulting job posting. At sites like these, local developers can be hired at typical market rates for most kinds of

Table 1 Ways to recruit software professionals

Ways to recruit software professionals
Student interns in companies recruit within a company
Personal contact within a company recruits on behalf of a researcher
In-house university software teams
Meetup.com social groups related to software development
Computer science student alumni (found in alumni databases)
Online labor markets (MTurk, ODesk, Freelancer.com, etc.)

programming jobs, enabling one to perform longitudinal experiments lasting weeks or months (for example, at the time of this writing, there are hundreds of developers with Java experience on oDesk.com, who charge a median of $30 per hour).

When writing recruitment materials or interviewing a potential participant, it is important to ask about inclusion criteria in a neutral manner, so that it is not clear to recruits what the "right" answers are. Otherwise, the risk arises that people eager to participate may provide misinformation, threatening the study validity and potentially introducing extraneous variation.

3.4 Recruiting Remote Participants

Designing a study so that it can be performed remotely can lower barriers to participation (Bruun et al. 2009), making possible larger sample sizes, and thereby increasing statistical power. For example, (Ko and Wobbrock 2010) recruited hundreds of web developers online in just 1 h, avoiding the logistics of scheduling a lab-based, in-person study. Remote participation, depending on the study, can also allow participants to use their own workstations, their own space, their own tools, and even work at times of their choosing, potentially making the experiment more ecologically valid by accounting for participants natural work environments.

Examples of disadvantages of remote participants are that developers may masquerade with levels of experience they do not really have, they may randomly check off boxes on questionnaires to quickly complete tasks, or they may leave a task temporarily, requiring the experimenter to monitor task activity. These can add new sources of extraneous variation. One way to detect such variation is to first run the study in a lab environment with a small number of users, establishing a baseline (Kittur et al. 2008). Another factor is the level of compensation; payments that are uncharacteristically high may attract participants that are excessively motivated by money, which may lead to unrealistic behavior in study tasks. Analyzing the market prices for tasks of comparable scope is one way to neutralize the effect of compensation.

If the tasks are of the kind where the correctness of the work performed cannot be judged automatically (for instance, if developers are asked to explain how much they trust the information produced by a tool), remote participants may have less of an incentive to provide a thoughtful response than those who are providing a response face-to-face (Bruun et al. 2009). To address this source of variation, one can introduce additional measures that indicate signs of hurried work, such as comprehension tests and time-to-respond measurements. Anomalies in these scores may indicate invalid data. Amazon's mTurk site details a number of other best practices for mTurk, and Kittur et al. (2008) detail several kinds of crosschecks.

4 Pre-Task Study Procedures

The canonical pre-task activities for an experiment include consenting participants, gathering demographic data, assigning participants to groups, and providing training. Each of these pre-task activities has subtle complexities that require some planning.

4.1 Informed Consent

Before participants begin the study, it is typical to first describe the purpose of the study, what they will be asked to do, and what risks and benefits participating might pose. This

process, which is typically called *informed consent*, allows a participant to decide whether they want to participate. Informed consent is typically administered through a document, a few pages in length, which details the study, its procedures, and its risks and benefits (most universities have templates for these). If a researcher is an industry employee, this is an ethical, but ultimately optional step in the process (unless required by an employer). If the researcher is a member of academia, however, it is likely that they are *required* to obtain informed consent and that the study design itself must first be approved by an ethics committee. In the U.S., these are called Institutional Review Boards (IRBs); all organizations that receive support from the federal government, directly or indirectly, are obligated to conduct these reviews in order to protect human participants from undue harm.

Obtaining approval from an IRB or similar ethics board can be time-consuming. In a recent survey of software engineering researchers (Buse et al. 2011), almost all of the respondents who had never conducted a user evaluation perceived the IRB as a barrier, whereas more than 20 % of respondents who had conducted a user evaluation said it was a barrier. While it can take at least a few weeks to obtain approval (and more, depending on the institution and study), software engineering experiments are often reviewed less rigorously than other types of experiments. This is because most tool evaluation experiments are eligible for expedited review because they (1) pose minimal risk, (2) they typically do not involve intentional deception, (3) they do not concern sensitive populations or topics, and (4) they include informed consent procedures. Minimal risk in this setting typically means that the probability of harm is not greater than that encountered in daily life. Harm can be physical, financial, or emotional. Therefore, some IRBs will inquire about the potential for significant stress, embarrassment, or shame in a task. These forms of harm can occur in software engineering experiments that ask participants to complete particularly difficult tasks, especially if the participants do not have adequate experience to be successful. For example, in one study run by the first author, a participant complained to the IRB that the task made her feel like even more of a failure at computer tasks than she already believed herself to be, harming her computer self-efficacy. (As we discuss later, it can be advantageous to *ensure* that not all participants complete a task, to avoid ceiling effects in success measurements, which may further increase the risk of such harm.)

To obtain IRB approval, one typically needs to complete a form detailing the design of the experiment, much like in the “method” section of a research paper. These forms will ask about the risks and benefits to participants, recruitment and compensation, and the informed consent document that will be provided to participants. More cautious IRBs will ask to see the exact study materials that a researcher will use, such as surveys, interview questions, and recruitment text. While preparing these materials can be time consuming, one has to prepare them anyway: the need for IRB approval will just require preparation farther in advance of the experiment.

4.2 Gathering Demographic Data

If a participant agrees to participate, often the next step is to gather some demographic information from them, to better understand their level of expertise, their background, their experience with certain tools, and so on. This can be done in an interview style, which has the advantage of possibly making the participant more comfortable, or with an online or paper survey, which has the advantage of ensuring more consistent responses. One way to ensure accurate and reliable responses to each survey question is to iteratively test the survey: ask someone to fill it out, and to think aloud while they attempt to answer the questions. If they have trouble interpreting a question, the question will probably lead to

noisy and unreliable answers. This iterative testing can be part of pilot testing (discussed in Section 8).

Some researchers gather demographic data during recruiting, especially if the demographic data is necessary for evaluating potential participants against inclusion criteria. For example, suppose a researcher wants to only study developers who have worked on the server side of enterprise web applications; it probably makes sense to ask about this and other background details before they arrive to participate. One can do this by merging the inclusion criteria assessment and demographic data into a single online survey that potential participants complete to determine if they are eligible for the study. The other advantage of gathering demographic data during recruiting is that one will have data on who did *not* participate, either because they did not satisfy the inclusion criteria, or because they never showed up to the study. This will help a researcher characterize the *selection bias* of the sampling procedures, which can limit the generalizability of a study's results.

Alternatively, researchers can gather demographic data *after* the participants perform the tasks in the experiment. This choice has several advantages: it ensures that there will be enough time to go through all of the tasks; it may also be better for participants to be fatigued while they fill out a survey than while they are trying to complete the final task in a study. There is also the possibility that by asking about participants' experience just before a task, the less experienced participants may feel less confident in their ability to succeed at it because they will be reminded of their inexperience. This effect, known as *stereotype threat*, can have a significant effect on skilled performance (Steele and Aronson 1995).

Some IRBs in the United States do not approve gathering demographic data before obtaining consent. If this is the case, or the study design gathers demographic data after study tasks, it may only become apparent that a participant did not satisfy the inclusion criteria until after they have completed a task. For example, it may become evident after observing a participant attempt to program in a particular language that they do not really know the language. Data from participants that do not satisfy the inclusion criteria should be excluded from analyses.

4.3 Assigning Participants to an Experimental Condition

There are many ways to assign participants to experimental conditions (Rosenthal and Rosnow 2007). Most studies use simple random assignment, distributing random variation in participants' skills and behaviors across all conditions, minimizing the chance that some observed difference in the groups' performance is due to participant differences, rather than differences in the tools being compared. There are alternatives to random assignment, however, that can reduce the number of participants necessary to recruit.

One alternative is the *within-subjects* experiment (Rosenthal and Rosnow 2007), where all of the participants in a study use all of the tools being compared, one at a time, across several tasks. For example, suppose a researcher had a new debugging tool and wanted to show that developers were able to more quickly fix failed tests than without the tool. A within-subjects design could involve two tasks, where a participant would use the new debugging tool for one task, and then *not* use the tool for the other task. Because there is significant potential for participants to learn something from the new tool that would give them an advantage in the other task—known as a *learning effect* (Rosenthal and Rosnow 2007)—one must randomize both the order of tasks and on which tasks a participant gets to use the experimental tool. This randomization is known as *counterbalancing* (Rosenthal and Rosnow 2007). When there are more than two tools being compared, counterbalancing can be achieved by mapping conditions onto a Latin Square (Martin 1996). There are, of course,

downsides to within-subjects designs, including fatigue or boredom: giving a developer multiple tasks to complete with multiple different tools could be exhausting and require an impractical amount of their time.

Another alternative to random assignment is an *interrupted time-series* design (McDowall et al. 1980). The idea behind this design is to make three measurements of the outcome variable: before the tool is introduced, after the tool is introduced, and, finally, after removing the tool from the environment. This allows one to see the potential causal effects of the tool on a task over time while testing on only a single group of participants. For example, a study evaluating a new debugging tool might give participants 9 small debugging tasks, randomizing their order, and only allow the participants to use the experimental tool in the middle three tasks. If the tool had an effect on debugging, one would expect the participants' debugging speed to go up after the tool was introduced and then back down after it was taken away. This would provide quasi-experimental evidence that any improvements in debugging speed were due to the introduction and removal of the tool, rather than learning effects from success with prior tasks. This design also has the advantage of being more natural in industry settings. For example, an interrupted time-series design can be framed as a "trial run" of a new tool in a company setting: the tool is introduced, improving outcomes, and then the tool is taken away, to see whether performance returns to its original baseline. Participants in the study can then better articulate what benefit the tool provided to their work, because they had the experience of both gaining and losing the tool's benefit over a period of time. A disadvantage of an interrupted time-series design is that it provides less confidence about causality, because it is harder to rule out the effect of confounding variables that co-occurred during the introduction and removal of the tool.

4.4 Training Participants

Before participants begin work on the experiment's tasks, one must first ensure that the participants know everything necessary to succeed. This includes knowledge such as:

- *How to use the tools in the environment provided*. The tools might include the research tool, the tool used in the control group, and the other developer tools that the participants will need. Note, however, that in studies evaluating whether participants can learn to use the tool *independently*, the participants will need training on everything *except* the research tool.
- *Terminology*. If participants do not understand complex concepts or terminology used to describe the task, they may struggle with a task because of a confusing word, making it difficult to observe the benefits of the tool.
- *The design of the programs they will work with during tasks*. There may be aspects of a program's design that would only be known after significant experience with a code base. If this knowledge is necessary for success in the experiment's tasks, but not something that the tool is intended to help them acquire, then providing the knowledge before they begin the task may reduce extraneous variation. For example, a researcher might give them documentation on the implementation and a brief lesson on the key components in the implementation. Large, complex software applications can often take weeks to learn in full, and so the knowledge provided in a study needs to be tailored to the specific information needs of the tasks.

The decision of what to teach and what to have participants learn during the tasks is a key experimental design consideration: whatever information is provided during training becomes an assumption about the conditions in which the tool will be used. For example, Ko

and Myers (2009) were evaluating a new debugging tool's effect on debugging speed and determined that participants would need to learn how to use the tool and also something about the 150,000-line implementation of the program the participants would debug. One design that Ko and Myers considered was to provide no training upfront, having the participants learn how to use the tool and learn the application's implementation during the tasks themselves. This design would have tested whether the tool could be quickly learned and applied to unfamiliar code with no training. The alternative design they considered was to first spend 30 min teaching the participants how to use the debugging tool, and then begin the tasks. This design would have tested whether the tool could be used to debug unfamiliar code, *under the assumption* of a systematic introduction to the tool's features and concepts. After several pilot experiments, the authors determined that it took too much time for the participants to learn how to use the tool independently, and so they chose the second design. The tradeoff was that the results of the experiment assumed prior training on the tool, which may be an unrealistic expectation in practice. As with any controlled experiment, the challenge was in deciding whether this was *too* unrealistic to be a useful evaluation. The authors decided it was not.

The design of the training materials is similar to any other kind of instructional design: the study should provide a way to teach the concepts and skills quickly and effectively and devise a way to ensure that the participants have successfully learned the material. To teach the material, researchers can create videos or paper instructions, do a live walkthrough, provide example tasks, give computer-based or tool-embedded tutorials (Kelleher and Pausch 2005), provide command cheat-sheets, or offer any other way of delivering instruction. To test whether the participants have learned everything necessary in the training, a study might ask them to explain the materials back to the experimenter, provide them with an assessment of what they learned, or have a variable-length question and answer session that ends when the researcher is confident that they understand the material.

To be more certain that the participants understand everything in the training, one effective approach for both teaching and assessing knowledge is a *mastery-learning* approach (Keller 1968) (Anderson and Reiser 1985). This involves devising a series of fixed and testable learning objectives, and only allowing the participant to move to the next learning objective when the previous objective is met, typically through testing of some sort. Because this may require a variable amount of time, this can affect the time remaining in the study to complete the tasks. To avoid this variation, one option is to have participants complete training in a separate session beforehand; for example, participants might complete a tutorial online on their own.

There is no one right approach to training. But generally, the more formal the instruction and the more careful the assessment, the more comparable participants' knowledge, and the less variation there will be in participants' performance on the tasks (and therefore the lower the chance of a undue negative results). These issues do, however, highlight one challenge with software engineering tool evaluation in general: many tools may not have a significant positive effect until a software professional knows them quite well (after months or perhaps years of use), making it difficult to recruit or train anyone in a manner that would allow for a meaningful evaluation. This suggests that there is an inherent bias in controlled experiments toward evaluating tools that can be quickly learned, and against tools that require significant practice. Future work on software engineering evaluation methods may need to explore new forms of longitudinal evaluation that exploit the for-hire services such as TopCoder and oDesk mentioned in Section 3, to allow developers time to gain expertise with a tool before conducting an experimental evaluation.

5 Task Design

One of the challenges in controlled experimental design is the tradeoff between realism and control (Sjøberg et al. 2003): realistic study designs provide for greater generalizability, but involve fewer controls, making it more difficult to know with certainty what causes any differences observed in an experiment. Task design is at the heart of this tradeoff, as tasks represent a distillation of realistic and messy software engineering work into brief, accessible, and actionable activities. In this section, we discuss several aspects of task design and their relationship to generalizability.

5.1 Feature Coverage

One aspect of tasks to consider is what features of the tool the task will exploit. If the tool only takes one action or provides one kind of output, this might be a simple decision. If the tool provides a wide range of functionality and has multiple commands, the researcher must decide on what range of features the tasks will focus. The more features that participants must learn and experiment with, the more diverse their approaches to the task may be. This variation in strategy can increase variation in performance, reducing the study's ability to detect the tool's benefits. However, tasks that encourage developers to use only part of a tool will limit ecological validity, as in any realistic setting, developers would have all of the features available to them and have to decide which features to use for a task. The more coverage one chooses, the more realistic the study, but the higher the risk of undue negative results.

5.2 Experimental Setting

The physical or virtual location in which participants complete a task is another aspect of task design that can affect the generalizability of a controlled experiment's results. For example, a study that is performed in a developer's actual workspace or using a developer's familiar toolset is likely to be more realistic than a setting that a developer is entirely unfamiliar with. On the other hand, a developer's workspace can be difficult to control. Note, however, that even lab settings raise some control difficulties: the lab may have distractions that affect participant behavior, such as people walking into the lab and interrupting the study; some of the lab's chairs could be uncomfortable; or computers may use different computing platforms at different seats. If an experiment involves multiple participants in the same room (as in (Rothermel et al. 2000)), participants might "follow the crowd" and finish early simply because other participants are leaving. If the study allows remote or asynchronous participation, these problems can multiply, because the experimenter might not be aware of interruptions or if the tool even worked correctly. The tradeoff here is the same as in the previous section: the more realistic the setting, the less control, and the higher the risk of undue negative results.

For controlled experiments, software engineering researchers generally favor control in such tradeoffs. In our systematic review, we found that only 14 % of the 345 experiments with human participants were performed outside of a lab environment. This trend has not changed in the past decade ($\chi^2(10,N=345) = 3.89$, $p=.952$). Experiments were also significantly less likely to be performed outside of the lab when compared to non-experiments ($\chi^2(1,N=345) = 3.70$, $p<.05$). In fact, in the literature we surveyed, only two experiments were performed outside of a lab environment (Kersten and Murphy 2006; Dig et al. 2008). Both compensated for this lack of control by recruiting a large sample of observations. Of

course, whether favoring control is appropriate depends on the nature of the research question being asked and what is already known about an approach: if a tool is entirely new, it may be more valuable to observe a tool being used in more realistic conditions with fewer controls in order to better understand the benefits of the tool in practice.

5.3 Task Origin

Tasks can vary in their origin. Some researchers select “found” tasks from real software development projects and then adapt them to suit a study’s needs, while others design tasks from scratch.

The benefit of a “found” task is that the study’s results may be more ecologically valid, illustrating a tool’s benefits (or lack thereof) on a real software engineering activity taken from practice. For example, many of the studies in our systematic review searched for bug reports in open source projects, as they define some problem that must be solved and are closely linked with a real code base. This practice is not yet dominant, however: in the past decade, only 15 % of studies have used this approach (and in our survey, the evidence does not suggest that this has not changed over time ($\chi^2(10,N=345) = 9.61$, $p=.476$)).

The benefit of designing a task from scratch is that it can be tailored specifically to exercise the features of the tool. The risk in doing this is that the task will not adequately reflect the types of tasks that occur in practice. There is also a risk that the new task gives the tool an unrealistic advantage over the baseline tool in the study. Therefore, creating a custom task may be necessary when a tool supports tasks that do not yet occur in real software engineering projects or there is evidence that a *type* of task occurs in practice, but not suitable actual tasks. For example, LaToza and Myers (2011a, b) wanted tasks that involved participants answering code reachability questions, but it was almost impossible to determine, simply from bug report descriptions in open source projects, which bugs might have the right characteristics. They did have evidence from previous studies that the tasks occurred in practice, however, and so they created their own tasks based on the properties of previously observed tasks (LaToza and Myers 2010). Ultimately, judgments about whether a task is “typical” are difficult to make. Recent studies of tasks in various software development contexts are one basis, however, for determining how frequently certain types of tasks occur in practice (e.g., Sillito et al. 2006; Ko et al. 2007; LaToza and Myers 2010).

It is also possible to allow participants to choose their own tasks for a study. This would have strong ecological validity, since all of the tasks would be real, but it would also introduce so much variation that it might be difficult to statistically detect differences between one tool and another. Some researchers perform these studies, but rarely make quantitative comparisons of the results (see Atkins et al. 2002 for a rare example of a study that does).

5.4 Programs and Data

Another dimension of task design is the data and source code used. Like the tasks themselves, real programs and data from open source projects ensure that the study results reflect real benefits. The downside to using realistic programs is that they can be messy and complex. This can lead to additional engineering work to scale the tool to supporting such programs. At the same time, it can also make it more challenging for study participants to understand the system, requiring either more time in the study design to teach the system or

introducing the risk that the system will be too complex for participants to complete any task. To work around this complexity, one can choose a real system, but focus participants attention (and tutorial materials) on a piece of the system that is less complex. Most software engineering studies in the past decade have used this approach, finding real programs from open source projects or other sources: 75 % of all studies used artifacts drawn from practice. This trend does not appear to have changed over the past decade ($\chi^2(10,N=345) = 8.84$, $p=.547$).

5.5 Task Duration

Deciding on a task duration is essentially a choice between two options: unlimited time to work on a task (allowing either the participant or the experimenter to decide when the task is complete) or a time limit. A disadvantage of long or unlimited work times is that this requires recruiting participants willing to work for such time periods. This challenge might be avoided if one wants to allow participants to complete the task remotely; they can then work on the task when they have time, completing it on their own schedule. For example, Hanenberg (2010) used a variable task duration in a comparison of static and dynamic typing, having participants work for up to 27 h across 4 days. The advantage of these long or unlimited task durations is that they allow larger and more realistic tasks. A disadvantage is loss of control over how participants allocate their time and what quality and quantity of data the experiment can collect: for example, they may rush to complete the task as quickly as possible, or they may spend so long on one task that they do not have much time to consider the others.

Of the 51 studies in our survey that reported task duration, most were lab studies, with a median length of nearly 2 h (see Table 2). These 2 h, however, included *every* part of the experiment, not just the tasks; thus so these likely included only 60–90 min of work time, to leave time for consent, surveys, debriefing, and other experiment activities.

These times are similar to the 1–2 h task durations reported for lab studies published from 1993 to 2002 (Sjøberg et al. 2005).

Time limits can also introduce a variation issue: tasks may be so easy that almost everyone completes them before the time expires. This can lead to *ceiling effect*s (Rosenthal and Rosnow 2007) on outcome measures, making it difficult to statistically discriminate between the outcomes of two groups because everyone does so well. Tasks may also be so difficult that no one can complete them in the allotted time no matter which tool they use (called a *floor effect* (Rosenthal and Rosnow 2007)). Neither type of task shows the true differences between tools because of too *little* variation within and between conditions.

Table 2 For the studies that reported task durations, the minimum, median, and maximum task duration reported across the five types of studies, in minutes

Type	*N*	Min	Median	Max
Field	6	107 min	45 days	173 days
Interview	1	–	–	–
Lab	39	18 min	110 min	10 days
Survey	0	–	–	–
Application	5	240 min	3.5 days	10 days
Overall	51	18 min	145 min	173 days

The number of tasks a participant completes is obviously constrained by how much of their time an experimenter has. However, the more tasks a participant completes, the more the study will be able to say about which types of tasks the tool is most effective in supporting and the more data can be collected to substantiate these benefits. Clearly, there is a complex interplay between the number of tasks, study duration, task difficulty, and recruiting. One way to inform these tradeoffs is through data: piloting the tasks (discussed in Section 8) can help determine how long they take participants to complete.

5.6 Task Difficulty

One challenging aspect of task design is predicting how difficult a task will be. One solution is to test potential tasks, observing their difficulty before committing to one. For example, one might select a variety of tasks and have several participants attempt all of them as part of piloting (Section 8). This will provide a sense of which tasks are least and most difficult. Alternatively, if a task is described as a bug report, it may be possible to infer the task's difficulty by looking at how long the bug took to be closed and what information was required to resolve it.

6 Measuring Outcomes

A central choice in designing a tool evaluation experiment is selecting and measuring the outcomes of developers' work on a task. Other work has addressed the basics of measurement (e.g., in software engineering, the goal-question-metric approach advocated by Basili et al. (1994) is a good place to start). In this paper, we focus on practical advice for measuring specific outcomes in clean and valid ways.

In our systematic review, we found a wide range of outcomes being measured, including task completion, time on task, failure detection, search effort, accuracy, precision, correctness, solution quality, program comprehension, confidence, usability, utility, mistakes, and many other measures specific to the tools being evaluated. In this section, we focus specifically on the three most common outcome variables: success on task, time on task, and tool usefulness.

6.1 Measuring Success on Task

For software engineering tasks, defining success involves determining three things: (1) the goal state or states that a participant must reach to be counted as successful, (2) a method for determining when a goal state has been reached, and (3) a method of communicating the goal to participants that all participants will interpret similarly. If not defined clearly or not followed consistently, each of these are potential sources of extraneous variation which can lead to undue negative results.

6.1.1 Identifying Goal States

A challenge in identifying goal states is that there are often many ways for a participant to be successful. To illustrate, let us consider (Ko and Myers 2009), in which participants were given 30 min per bug report and asked to use a new debugging tool to diagnose problems reported in two bug reports, each taken from an open source project. The conceptual definition of success used in this study was the elimination of the undesirable behavior described in the bug report. This conceptual definition was not sufficient to *measure* success, however. For example, one of the bug reports required a checkbox to be removed from a dialog box. There were multiple changes

that could have achieved this, including hiding the checkbox, removing the code that added the checkbox but keeping the checkbox construction, or commenting all code out that was related to the checkbox. The patch that was actually committed removed all traces of the checkbox from the implementation. The most ecologically valid definition of success would have been to require participants to write the patch exactly as submitted to the open source project. This would also have been quite restrictive, as most participants did not make exactly this change.

Even after preliminary analysis, pilot studies revealed that there were in fact *many* more possible changes than those the authors had anticipated. Therefore, rather than try to enumerate them all, the authors chose a more relaxed notion of success: the developers would write an explanation of the failure in terms of a sequence of events leading to failure and then propose a modification to the source code. To judge these explanations and proposed changes, the authors developed a grading rubric for comparing the proposed change to the patch submitted in the actual project. This approach introduced variation in the judges' ability to consistently score the responses, but allowed for variability in participants' solutions. It was also the only goal state that was both easy to explain to participants and feasible to achieve in a 30-min task. To ensure that the scoring rubric was reliable, the authors evaluated the inter-rater reliability of the judges' scores (Gwet 2010),.

Whether participants fix a bug, find a part of a program, make a decision, collaborate, or perform any other kind of software engineering activity, identifying goal states that count as success requires a careful analysis of the possible choices that participants might make. Again, a standard way to identify possible outcomes is through piloting (Section 8), allowing the experimenter to observe choices participants may make in the real study.

6.1.2 Determining When a Goal State is Reached

The second part of defining success on task is to define how the goal states will be observed. One decision here is whether the experimenter will make the determination (either by watching the participant work or via automated detection) or whether the participant will be responsible for telling the experimenter when they believe they have succeeded.

If the experimenter will monitor for success, it is helpful to know in advance the possible states that count as success. Detecting goal states through observation, such as by watching over the shoulder of a developer, can be error prone, and so it can be helpful for experimenters to practice. Experimenters may practice during piloting (Section 8); to train multiple experimenters to detect success, the pilots can be screen captured, so that the participants' paths to success can be observed repeatedly. Another approach to detecting success is to automate it, gathering data from a tool or integrated development environment that can reliably indicate when someone has succeeded. For example, one might use an automated test suite that can be run by the participant and define success as when all of the test cases pass, as was done in Ko and Wobbrock's study (2010) and in Holmes and Walker (2013). These automated approaches can be more precise by avoiding noise in manual detection, but require a researcher to identify *all* of the goal states in advance. This can be difficult, as developers often behave unexpectedly and achieve success in many ways.

Alternatively, the participants may determine for themselves when they believe they have succeeded. This is the most ecologically valid choice, since, in practice, developers do not have an oracle that notifies them when they have succeeded. This also ensures that the goal state will be easily observable, because it requires participants to explicitly communicate their success to the experimenter. This choice has the tradeoff, however, of introducing variation in how long a participant might take to complete a task, as individual participants may vary in the confidence they feel to be necessary to determine that they have succeeded.

One can try to influence this by providing guidance, but this can be difficult to convey unambiguously. Another approach is to give participants a fixed time period in which to work and ask them to use any remaining time to increase their confidence in their answer. However, this does not allow capturing time on task as an outcome measure.

6.1.3 Defining Success to Participants

The third aspect of defining success on task is unambiguously communicating the goal to participants. When participants themselves determine when they are successful, participants confused about what counts as success will tend to ask the experimenter if they are done. In this case, it is helpful to repeat the definition of success to them in a way that does not artificially influence their ability to succeed (which would confound the results of the study). When the experimenter is monitoring for success, participants may still ask if they have succeeded with the task simply because they want to be done. In this case, the experimenter can say, "I am unable to answer that question" and ask them to continue until they are done. Both of these scenarios can be frustrating for participants, and so it can be valuable to invest time in effectively and unambiguously communicating the task goal to participants.

6.2 Measuring Time on Task

Another popular outcome measure is time on task. To define it, one must precisely detail: (1) when a task *starts* and *ends* and (2) who will determine when the participant has finished.

The choice of a start and end point is really a choice about what constitutes "the task." For example, a task could start immediately after the experimenter hands the participant a written description of the task. This would include the time required to read and comprehend the description, introducing the variability in participants' reading speed and the uncertainty as to whether they read all the information, skimmed it, or glanced at just the first few lines. One study mitigated this variation by giving participants a packet of task instructions to read silently while the experimenter read the same instructions aloud (Golden et al. 2005); the participant was encouraged to ask questions when something was unclear. In this study, time on task did not start until after the experimenter had finished reading the packet. This ensured that every participant spent the time required to at least *hear* the instructions, but it could not guarantee that they understood them. Another way to provide similar assurance is to present task instructions in a wizard-like interface, prohibiting the participant from advancing until an appropriate minimum reading time has elapsed or comprehension questions are successfully answered. If the study gives developers unlimited time to read a task before time begins, developers may be able to start planning a solution, artificially decreasing the amount of time on task.

One must also define the *end* of a task; this decision is complicated by definitions of task success. If the task has an easily identifiable goal state, one could define the end of the task to be when the participant reaches it. However, if the participant succeeds but does not know it, one must decide whether to tell them that they have; similarly, if they fail but believe they have succeeded, one must decide whether to tell them that they have not succeeded. The decision here is ultimately on whether the time on task should include the time it takes for developers to both be successful and know they are successful. To claim that a tool helps developers succeed in a real-world context, the definition should probably also include the time it takes for a developer to decide they are done. This definition, however, threatens ecological validity, since the participant may say they have succeeded, when in fact they have not (in real life, the developer would only find out through later testing). All of these

options introduce variation into time measurements; the definition one chooses should therefore account for other variables being measured.

Clearly, time on task and success on task trade off with each other (Wickelgren 1977). Participants can often work faster and less carefully, reducing time, but also lowering the quality of their work; they may also work more slowly and carefully, increasing time and success. Because of this tradeoff, it is helpful to measure both, so that one can see *how* each developer managed speed and quality. If variation in developer performance is a concern, it can be helpful to communicate to participants whether speed or quality is more important, so they behave more consistently as a group, minimizing variation in measurements. To communicate this tradeoff,, the study can manipulate the incentive structure in the experiment. For example, a study can encourage participants to work more quickly by providing bonus pay for lower time on task. Or, alternatively, a study could encourage higher quality work by providing a bonus for a correct answer. Of course, all of these measures can threaten the realism of the study, since developers in practice likely make their own individual choices about speed and quality, independent of their work environment. These challenges again highlight the importance of considering other sources of evaluation insight, such as qualitative data, to help interpret the quantitative results of an experiment.

6.3 Measuring Usefulness

Many of the papers in our systematic review attempted to measure participants' opinions of the *usefulness* of a tool, which generally means that the tool provides functionality that satisfies a need or provides a benefit. Many experiments measured this by directly asking developers whether they found the tool "useful", whether they would "consider using it in the future", and of course, asking more qualitative questions about what was useful and why. These and related measures attempt to assess the likelihood of the experimental tool being used in a hypothetical future in which a tool was widely available.

Unfortunately, attempting to quantify usefulness is difficult and these self-report measurements often fail to capture the multidimensional nature of usefulness. Whether a tool is useful depends on developer's expectations about what it can accomplish for them, how motivated they are to use a new tool, how problematic their alternative tools are, and of course, the nature of their software engineering work. Whether a developer *reports* a tool as useful can also depend on whether they feel obligated to please an experimenter. For example, Dell et al. (2012) found that when interviewing participants about a novel system, they were 2.5 times more likely to prefer the technology they believed to be designed by the interviewer, even when the technology was inferior. Therefore, simply asking a developer whether they find a tool "useful" is likely not to produce meaningful data.

Despite these validity concerns, self-report data is a common way to assess usefulness that can be measured through validated instruments. Rather than simply ask if they find a tool "useful", one can instead use a previously validated instrument, such as the Technology Acceptance Model (embodied as a questionnaire), which has been studied extensively (Chuttur 2009). This model and its standard questionnaire (Davis 1989) prompts users to rate their agreement with several specific aspects of utility, such as "Using this tool improves the quality of the work that I do," "Using this tool gives me greater control over my work," "I am more productive with this tool", and "This tool makes my job easier". These questions focus on concrete aspects of the tool's applicability to the developer's work context.

Note that the *usefulness* of a tool and its *usability* are entirely different constructs. A *useful* tool provides functionality that satisfies a need or provides a benefit, whereas a *usable* tool is accessible, learnable, understandable, etc., but not necessarily useful. Therefore, a tool

could be both very usable and useless; a tool could also be useful but entirely unusable. That said, if the goal is to measure usefulness, the usability of a tool must be good enough for a developer to successfully exploit the utility of a tool, especially in the context of a brief experiment. If the usability is poor, participants may not have enough time to find, understand, and use the useful functionality to accomplish a task. Participants' opinions about the tools may also be affected by poor usability, making it difficult to know whether the tool would be useful in the absence of usability problems. There are many ways to assess usability before running a study, including validated instruments such as the System Usability Scale (SUS) (Bangor et al. 2008) and a wide range of lab-based usability methods (Rubin and Chisnell 2008), including methods that have been adapted to be specifically useful for preparing software development tools for controlled experiments (Ko et al. 2002).

7 Debriefing and Compensation

After a participant has completed the tasks, it is common practice in human subjects research to debrief the participant about the study and compensate them. The debriefing may also include other topics, including those that the experimenter did not detail during informed consent. These include:

- Explaining to the participant what the study investigating.
- Explaining why was the study was important to conduct.
- Explaining how the participant's data will be used to investigate the question.
- Explaining the correct solutions to the tasks? (Participants should not leave a study feeling as if they "failed," especially when tasks may have been designed to ensure that not every participant would succeed. Many ethicists feel that is a necessary part of research with human participants (Walther 2002)).
- If the participant did not use the experimental tool, it may be instructive for them to try it and provide feedback.
- Providing contact information so that the participant can ask further questions if they want to know more about the research.
- Instructions about information that they should not share with others. For example, if one is recruiting from a student population, students might share the answers to the tasks with their friends, who may later participate in the study.

Debriefing can also be an opportunity to get speculative feedback from participants about how they felt about the tool. One can solicit such feedback more formally through a semi-structured interview, gathering data on the participants' perceptions of the tool, the strategies it supported, and the features that they thought were most helpful.

8 Piloting

Designing a study with human participants is necessarily an iterative process. Running an experiment for the first time, like testing software for the first time, will reveal a range of problems, which might include confusing study materials, bugs in the tool, confusion about the tasks, and unanticipated choices made by participants. Therefore, a critical step in preparing an experiment is to run *pilots* (Keppel 1982): test runs of a study intended to find sources of extraneous variation. Our systematic review suggests that pilots may not yet be standard practice in software engineering research. We found that only 11 of the 345 studies

explicitly mentioned performing a pilot with human participants prior to running the experiment.

Some researchers like to begin with a series of "sandbox" pilots. Sandbox pilots are pre-pilots in which the researchers themselves are the participants. Sandbox pilots are easy to schedule and can reveal problems with the experiment without the trouble of recruiting outsiders. These can be particularly helpful when testing a novel tool that may still have bugs, performance problems, or serious usability problems that make it impossible to complete the tasks; but they can be helpful in catching a wide range of problems with the tool, tasks, materials, and measures. Sandbox pilots are an optional optimization: they help find and remove problems before bringing in the true pilot participants, thereby saving participant time and potentially saving experimenters' money.

Another optional pre-pilot step is to do analytical evaluations, which do not require participants. The Cognitive Walkthrough (CW) (Polson et al. 1992) is one such method that is accessible to researchers who are not HCI specialists (John and Packer 1995; Ko et al. 2002). One can use the CW to "walk through" an experiment from the perspective of a participant to find problems that could cause extraneous variation in developer behavior. For example, Ko et al. (2002) describe how the CW was used to improve an experimental evaluation of a testing tool for spreadsheet users. The original (pre-CW) experiment had failed to produce clear outcomes of any sort. The CW revealed several possible causes, uncovering problems in the design of the task; problems in the task instructions; and user interface problems with the tool and its environment that were so distracting, participants were likely to veer off into areas of no relevance to the experimental hypotheses. When the researchers resolved these problems and re-ran a new version of the experiment, it produced clean, clear outcomes that demonstrated the efficacy of the tool (Rothermel et al. 2000).

Whether or not one first begins with pre- piloting, the most important phase of piloting is testing the study design with the actual setup in which the study will be conducted, with as close to the same participants, physical set-up, handouts, tutorials, questionnaires, procedures, and experimenters as possible (although running pilots with lower fidelity is appropriate if one is still developing tasks and materials). Piloting can help reveal problems with all of these aspects, including the tool itself. For novel tools, it can be particularly important to find usability problems with a tool's user interface that interfere with participants' ability to succeed on the tasks. After conducting each pilot run, the problems discovered in the tool or study design should be fixed. Any data collected should be discarded, as the purpose of a pilot is to find problems in the experiment that need to be fixed, not to gather data (as the study setup is in flux, such data would have much extraneous variation).. After the problems have been fixed, another pilot run is conducted to test the fixes and discover other problems that may emerge. (Note that the presence of usability problems does not mean that the tool is not useful, nor is it necessary that a research tool be perfectly polished or industrial strength: rather, usability problems can simply make it difficult to successfully observe and measure a tool's impact, and so they often must be fixed in order to ensure the best chance of observing differences, if such differences exist). Determining when piloting stops and conducting the study begins requires balancing the benefits of pilots – lowering the risk of an undue negative result – against the costs – additional time and recruiting.

9 Conclusion

Clearly, there is more to designing software engineering tool evaluation experiments with human participants than we have described. For example, we have not addressed the

complexities of statistical hypothesis testing (for instance, many distributions are not normally distributed, requiring non-parametric tests). There are also sources of variation due to language and cultural differences that require special attention (Lazar et al. 2010). There are also many study design considerations that may only come with significant experience with a certain domain and the software developers who work in it.

While we believe that controlled experiments are a critical part of generating robust knowledge about the efficacy of software engineering tools, they are not a panacea. Because of the necessary loss of realism in experiments, they are inherently more difficult to generalize to real-world contexts. There is also a strong tendency in many research communities (e.g., HCI (Kaptein and Robertson 2012)), to overemphasize the importance of statistical significance, ignoring other arguably more important outcomes, such as effect size. After all, it is not only the difference between two technologies that is significant to both researchers and practitioners, but also the magnitude and meaning of that difference (Aranda 2011).

Other empirical methods can effectively complement controlled experiments and make up for their deficiencies. For example, case studies, when performed rigorously (Flyvbjerg 2006; Runeson and Höst 2009; Yin 2003), can produce rich, real-world data as to how software engineering tools can or cannot be integrated into practice. Case studies have little ability to isolate variables (the strength of controlled experiments), but because they are not controlled, they avoid the necessarily artificial and narrow view of the controlled experiment, which have been criticized for failing to integrate knowledge into more general and powerful explanations (Newell 1973), and even for inhibiting innovation (Olsen 2007, Greenburg and Buxton 2008). Readers are encouraged to consult the many high quality introductions to case studies and other methods (Runeson and Höst 2009; Dybå et al. 2011; Dittrich 2007) and empirical software engineering more broadly (Wohlin et al. 2000; Juristo and Moreno 2001; Shull et al. 2006).

We hope that the methodological guidance we have provided, combined with the existing empirical software engineering literature, will better arm researchers with practical guidance in using quantitative controlled experiments to evaluate software engineering tools with human participants.

Acknowledgments We thank Bonnie E. John for her early contributions to this work. This material is based in part upon work supported by the NSF Grant CCF-0952733 and AFOSR FA9550-0910213 and FA9550-10-1-0326. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

Appendix

Past literature has considered the role of human participants in software engineering research before 2002 (Sjøberg et al. 2005). We still know little, however, about the trends in empirical evaluations with human participants in software engineering from the past decade. To

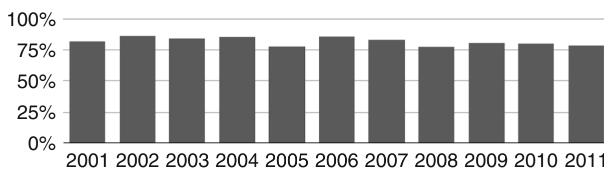

Fig. 3 Proportion of papers contributing software engineering tools

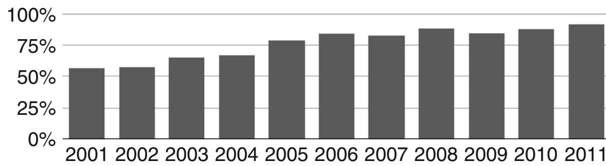

Fig. 4 Proportion of papers contributing tools that also reported an empirical evaluation of the tool

address this gap, and to support the methodological discussion in this paper, we conducted a systematic literature review (Kitchenham et al. 2010) of research from the past decade.

We focused on publications from four of the top software engineering research venues: the *International Conference on Software Engineering* (ICSE), the *ACM SIGSOFT Symposium on Foundations of Software Engineering* (and associated *European Software Engineering Conference* in alternating years), *ACM Transactions on Software Engineering and Methodology*, and *IEEE Transactions on Software Engineering*. We chose these primarily because of their status as flagship software engineering journals and conferences, but also because of their focus on software engineering tools (as opposed to studies focused on understanding software engineering practice). Our sample included all of the research publications in these four venues that were published from 2001 to 2011. For conferences, we included only the technical proceedings, excluding experience reports and software engineering education research papers. The resulting set included 1,701 publications.

To begin, we first sought to identify the subset of these papers that reported on new technologies, methods, formalisms, design patterns, metrics, or other techniques intended to be used by a software professional of some kind. The first two authors redundantly classified a random sample of 99 of the 1,701 papers (the Cohen's Kappa of this classification was 0.8, considered near perfect agreement). We then split the remaining papers and classified them independently in random order, identifying 1,392 papers (81 % of all publications) describing some form of software engineering "tool." As seen in Fig. 3, the proportion of papers contributing tools has not changed significantly in the past decade ($\chi^2(10,N=1,701) = 11.1, p=.35$).

To narrow the sample of "tool papers" to "tool papers with evaluations," we checked each paper for an evaluation of the tool (including evaluations of both *internal* properties such as performance or accuracy and *external* properties of how the tool is used by people). We counted an evaluation as "empirical" if it included any quantitative or qualitative data from observation of the tool's behavior or use. Therefore, if the paper only described a task that *could* be performed with a tool, but was not actually performed, the paper was not included. Achieving high reliability when redundantly classifying the same 99 papers (Kappa=0.83),

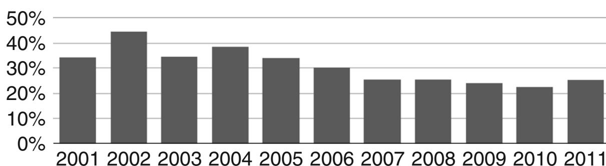

Fig. 5 Proportion of studies contributing evaluations examining how the authors or recruited participants used the tool. Contrary to the graph in Fig. 4, this graph shows a significant decline, suggesting that papers are increasingly evaluating the non-human use of tools

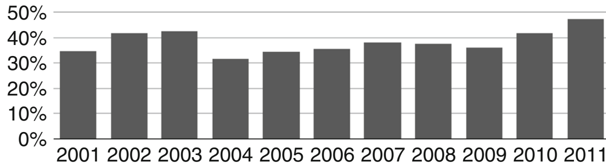

Fig. 6 Proportion of studies evaluating tool use that involved non-author human participants. Although studies of use are less common (Fig. 5), they are increasingly involving human participants

we split and classified the remaining papers, identifying 1,065 reporting on both a tool and empirical evaluation of it; 77 % of tool papers included an empirical evaluation. As seen in Fig. 4, the proportion of papers contributing tools and reporting empirical evaluations of them has increased significantly in the past decade from about 50 % to over 90 % ($\chi^2(10,N=1,392) = 117.9$, $p<.001$). This shows that empirical evaluations are now widely adopted and perhaps even expected for publication of novel tool contributions.

To further narrow our sample to tool papers that evaluated some aspect of the *use* of a tool, we identified the individual studies reported in each paper. We counted any study with a unique method within a paper as a separate study (lab studies that were run multiple times or case studies run on multiple programs were counted as a single study). After achieving high reliability on this definition (Kappa=0.62), we found 1,141 studies across 1,065 papers.

With the studies in each paper identified, we then classified each of these studies as either involving the human use of the tool or not. *Any* person using the tool for some task, including the paper authors themselves if they described their own use of the tool, was included. After achieving high reliability on this classification (Kappa=.70), we classified the papers, finding 345 studies across 289 papers that involved human use of a tool. As shown in Fig. 5, the proportion of studies that involve developers or other software professionals using a tool is actually on a slow decline in our sample, from a peak in 2002 of 44 % to only 26 % of papers in 2011 ($\chi^2(10,N = 1141) = 21.0$, $p<.05$). This means that although more evaluations are being done, a smaller proportion of them are evaluating a tool's use by humans (instead evaluating its internal characteristics such as speed, accuracy, or correctness).

The subset of tool evaluations of human use that recruited human participants and did not use authors is shown in Fig. 6. This plot shows that although studies evaluating tool use are less common (Fig. 5), an increasing proportion of these studies involve human participants. Therefore, when software engineering researchers are studying how a tool is used, they are increasingly recruiting participants rather than using themselves.

Most of the studies instead used lab studies (conducting an study in a controlled setting with participants), *interviews* (demonstrating the tool and asking participants to respond to spoken questions about it), *surveys* (demonstrating the tool and asking participants to respond to written questions about it), *field deployments* (observations of the use of the tool

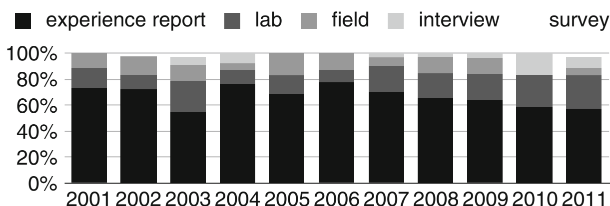

Fig. 7 Proportion of methods used in studies evaluating human use of a tool

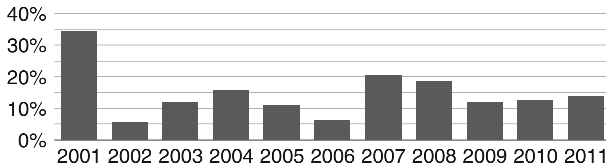

Fig. 8 Proportion of papers evaluating use that were experiments. The number of experiments ranged from 2 to 9 per year

in real settings on real tasks), and a method we will call *tool use experience reports*[2] (the use of the tool with a specific program or data set, either by the authors or some other person). As shown in Fig. 7, the most common method by far was the tool use experience report, which was the method of 67 % of the 345 studies evaluating human use of a tool. As seen in Fig. 7, the other four methods were much less common. The relative proportion of tool application studies compared to other types has not changed significantly in the past decade ($\chi^2(10, N=345) = 9.6$, $p=.473$). The frequency of experiments per year has also not changed significantly over the past decade ($\chi^2(10, N=345) = 8.2$, $p=.611$). Figure 8 shows that only a small subset of these studies are controlled experiments. In fact, the number of experiments evaluating tool use has ranged from 2 to 9 studies per year in these four venues, for a total of only 44 controlled experiments with human participants over 10 years.

These results indicate several major trends in the methods that software engineering researchers use to evaluate tools: (1) empirical evaluations are now found in nearly every paper contributing a tool, (2) the proportion of these evaluations that evaluate the human *use* of the tool is on the decline, (3) an increasing proportion of human use evaluations involve non-author participants, but (4) experiments evaluating human use are still quite rare.

These findings are subject to several threats to validity. We considered only 4 journals and conference proceedings in our review, focusing on those with a strong reputation for contributing new tools. There are many other software engineering publication venues where such work appears. It is possible that the trends we observed are particular to the venues we chose; for example, Buse et al. (2011) found that while ICSE and FSE showed no signs of increases in user evaluations, OOSPLA, ISSTA, and ASE did. There may also be evaluations with human participants that were never published or that were published in other venues after being rejected by the venues that we *did* analyze.

References

Anderson JR, Reiser BJ (1985) The LISP tutor. Byte 10:159–175

Aranda J (2011) How do practitioners perceive Software Engineering Research? http://catenary.wordpress.com/2011/05/19/how-do-practitioners-perceive-software-engineering-research/. Retrieved: 08-01-2011

Atkins DL, Ball T, Graves TL, Mockus A (2002) Using version control data to evaluate the impact of software tools: A case study of the version editor. IEEE Trans Softw Eng 28(7):625–637

Bangor A, Kortum PT, Miller JT (2008) An empirical evaluation of the system usability scale. Int J Human-Comput Interact 24(6):574–594

Basili VR (1993) The experimental paradigm in software engineering. Int Work Exp Eng Issues: Crit Assess Futur Dir 706:3–12

[2] Studies using this last method were often referred to by authors as “case studies,” but this usage conflicts with the notion of case studies as empirical investigations of some phenomenon within a real-life context (Yin 2003), as the tool use experience reports in these papers were not performed in real life contexts. “Case study” was also used to refer to evaluations *without* human use.

Basili VR (1996) The role of experimentation in software engineering: Past, current, and future. International Conference on Software Engineering, 442–449

Basili VR (2007) The role of controlled experiments in software engineering research. Empirical Software Engineering Issues, LNCS 4336, Basili V et al. (Eds.), Springer-Verlag, 33–37

Basili VR, Selby RW, Hutchens DH (1986) Experimentation in software engineering. IEEE Trans Softw Eng, 733–743, July

Basili VR, Caldiera G, Rombach HD (1994) The goal question metric approach. In Encyclopedia of Software Engineering, John Wiley and Sons, 528–532

Beringer P (2004) Using students as subjects in requirements prioritization. International Symposium on Empirical Software Engineering, 167–176

Boehm BW, Papaccio PN (1988) Understanding and controlling software costs. IEEE Trans Softw Eng SE-14(10):1462–1477

Breaugh JA (2003) Effect size estimation: factors to consider and mistakes to avoid. J Manag 29(1):79–97

Bruun A, Gull P, Hofmeister L, Stage J (2009) Let your users do the testing: a comparison of three remote asynchronous usability testing methods. ACM Conference on Human Factors in Computing Systems, 1619–1628

Buse RPL, Sadowski C, Weimer W (2011) Benefits and barriers of user evaluation in software engineering research. ACM Conference on Systems, Programming, Languages and Applications

Carver J, Jaccheri L, Morasca S, Shull F (2003). Issues in using students in empirical studies in software engineering education. Software Metrics Symposium, 239–249

Chuttur MY (2009). Overview of the technology acceptance model: Origins, developments and future directions. Indiana University, USA, Sprouts: Working Papers on Information Systems

Davis FD (1989) Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Q 13(3):319

Dell N, Vaidyanathan V, Medhi I, Cutrell E, Thies W (2012) “Yours is better!” Participant response bias in HCI. ACM Conference on Human Factors in Computing Systems, 1321–1330

Dieste O, Grim´n A, Juristo N, Saxena H (2011) Quantitative determination of the relationship between internal validity and bias in software engineering experiments: Consequences for systematic literature reviews. International Symposium on Empirical Software Engineering and Measurement, 285–294

Dig D, Manzoor K, Johnson R, Nguyen TN (2008) Effective software merging in the presence of object-oriented refactorings. IEEE Trans Softw Eng 34(3):321–335

Dittrich Y (ed) (2007) Special issue on qualitative software engineering research. Inf Softw Technol, 49(6):531–694. doi:10.1016/j.infsof.2007.02.009

Dybå T, Kampenes V, Sjøberg D (2006) A systematic review of statistical power in software engineering experiments. Inf Softw Technol 48(8):745–755

Dybå T, Prikladnicki R, Rönkkö K, Seaman C, Sillito J (2011) Qualitative research in software engineering. Empir Softw Eng 16(4):425–429

Easterbrook S, Singer J, Storey M, Damian D (2008) Selecting empirical methods for software engineering research, in Guide to Advanced Empirical Software Engineering, Springer, 285–311

Feigenspan J, Kastner C, Liebig J, Apel S, Hanenberg S (2012). Measuring programming experience. International Conference on Program Comprehension, 73–82

Fenton N (1993) How effective are software engineering methods? J Syst Softw 22(2):141–146

Flyvbjerg B (2006) Five misunderstandings about case study research. Qual Inq 12(2):219–245

Glass RL, Vessey I, Ramesh V (2002) Research in software engineering: an analysis of the literature. Inf Softw Technol 44(8):491–506

Golden E, John BE, Bass L (2005) The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. ACM/IEEE International Conference on Software Engineering

Greenberg S, Buxton B (2008) Usability evaluation considered harmful (some of the time). ACM Conference on Human Factors in Computing Systems, 111–120

Gwet KL (2010) Handbook of inter-rater reliability, 2nd edn. Advanced Analytics, Gaithersburg

Hanenberg S (2010) An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA), 22–35

Hannay JE, Sjøberg DIK, Dyba T (2007) A systematic review of theory use in software engineering experiments. IEEE Trans Softw Eng 33(2):87–107

Holmes R, Walker RJ (2013) Systematizing pragmatic software reuse. ACM Trans Softw Eng Methodol 21(4), Article 20: 44 pages

John B, Packer H (1995) Learning and using the cognitive walkthrough method: a case study approach. ACM Conference on Human Factors in Computing Systems, 429–436

Juristo N, Moreno AM (2001) Basics of software engineering experimentation. Springer

Kampenes V, Dybå T, Hannay J, Sjøberg D (2007) A systematic review of effect size in software engineering experiments. Inf Softw Technol 49(11–12):1073–1086

Kaptein M, Robertson J (2012) Rethinking statistical analysis methods for CHI. ACM Conference on Human Factors in Computing Systems, 1105–1114

Kelleher C, Pausch R (2005). Stencils-based tutorials: design and evaluation. ACM Conference on Human Factors in Computing Systems, 541–550

Keller FS (1968) Good-bye teacher. J Appl Behav Anal 1:79–89

Keppel G (1982) Design and analysis: a researcher's handbook, 2nd edn. Prentice-Hall, Englewood Cliffs

Kersten M, Murphy G (2006) Using task context to improve programmer productivity. ACM Symposium on Foundations of Software Engineering, 1–11

Kitchenham BA, Pfleeger SL, Pickard LM, Jones PW, Hoaglin DC, Emam, K.E., Rosenberg J (2002) Preliminary guidelines for empirical research in software engineering. IEEE Trans Softw Eng 28(8):721–734

Kitchenham BA, Brereton P, Turner M, Niazi MK, Linkman S, Pretorius R, Budgen D (2010) Refining the systematic literature review process—two participant-observer case studies. Empir Softw Eng 15(6):618–653

Kittur A, Chi EH, Suh B (2008) Crowdsourcing user studies with Mechanical Turk. ACM Conference on Human Factors in Computing Systems, 453–456

Ko AJ, Myers BA (2009) Finding causes of program output with the Java Whyline. ACM Conference on Human Factors in Computing Systems, 1569–1578

Ko AJ, Wobbrock JO (2010) Cleanroom: edit-time error detection with the uniqueness heuristic. IEEE Symposium on Visual Languages and Human-Centric Computing, 7–14

Ko AJ, Burnett MM, Green TRG, Rothermel KJ, Cook CR (2002) Using the Cognitive Walkthrough to improve the design of a visual programming experiment. J Vis Lang Comput 13:517–544

Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. International Conference on Software Engineering (ICSE)

LaToza TD, Myers BA (2010) Developers ask reachability questions. *International Conference on Software Engineering* (ICSE), 185–194

LaToza, TD, Myers BA (2011) Visualizing call graphs. IEEE Visual Languages and Human-Centric Computing (VL/HCC), Pittsburgh, PA

LaToza TD, Myers BA (2011) Designing useful tools for developers. ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU), 45–50

Lazar J, Feng JH, Hochheiser H (2010) Research methods in human-computer interaction. Wiley

Lott C, Rombach D (1996) Repeatable software engineering experiments for comparing defect-detection techniques. Empir Softw Eng 1:241–277

Martin DW (1996) Doing psychology experiments, 4th edn. Brooks/Cole, Pacific Grove

McDowall D, McCleary R, Meidinger E, Hay RA (1980) Interrupted Time Series Analysis, 1st Edition. SAGE Publications

Murphy GC, Walker RJ, Baniassad ELA (1999) Evaluating emerging software development technologies: lessons learned from assessing aspect-oriented programming. IEEE Trans Softw Eng 25(4):438–455

Murphy-Hill E, Murphy GC, Griswold WG (2010). Understanding context: creating a lasting impact in experimental software engineering research. FSE/SDP Workshop on Future of Software Engineering Research

Newell A (1973) You can't play 20 questions with nature and win: projective comments on the papers of this symposium. In: Chase WG (ed) Visual information processing. Academic, New York

Nickerson RS (1998) Confirmation bias: a ubiquitous phenomenon in many guises. Rev Gen Psychol 2(2):175–220

Nimmer JW, Ernst MD (2002) Invariant inference for static checking: an empirical evaluation. SIGSOFT Softw Eng Notes 27(6):11–20

Olsen DR (2007) Evaluating user interface systems research. ACM Symposium on User Interface Software and Technology, 251–258

Polson P, Lewis C, Rieman J, Wharton C (1992) Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. Int J Human-Comput Interact 36:741–773

Ramesh V, Glass RL, Vessey I (2004) Research in computer science: an empirical study. J Syst Softw 70(1–2):165–176

Rombach HD, Basili VR, Selby RW (1992) Experimental software engineering issues: critical assessment and future directions. International Workshop Dagstuhl Castle (Germany), Sept. 14–18

Rosenthal R (1966) Experimenter effects in behavioral research. Appleton, New York

Rosenthal R, Rosnow R (2007) Essentials of behavioral research: methods and data analysis. McGraw-Hill, 3rd edition

Rosenthal R, Rubin DB (1978) Interpersonal expectancy effects: the first 345 studies. Behav Brain Sci 1(3):377–386
Ross J, Irani L, Silberman MS, Zaldivar A, Tomlinson B (2010) Who are the crowdworkers? Shifting demographics in Mechanical Turk. ACM Conference on Human Factors in Computing Systems, 2863–2872
Rothermel KJ, Cook C, Burnett MM, Schonfeld J, Green TRG, Rothermel G (2000) WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation. ACM International Conference on Software Engineering, 230–239
Rubin J, Chisnell D (2008) Handbook of usability testing: how to plan, design, and conduct effective tests. Wiley
Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14(2):131–164
Shull F, Singer J, Sjøberg DIK (2006) Guide to advanced empirical software engineering. Springer
Sillito J, Murphy G, De Volder K (2006) Questions programmers ask during software evolution tasks. ACM SIGSOFT/FSE, 23–34
Sjoberg DIK, Dyba T, Jorgensen M (2007) The future of empirical methods in software engineering research. In 2007 Future of Software Engineering (FOSE '07), 358–378
Sjøberg D, Anda B, Arisholm E, Dyba T, Jorgensen M, Karahasanovic A, Koren E, Voka M (2003) Conducting realistic experiments in software engineering. Empirical Software Engineering and Measurement
Sjøberg DIK, Hannay JE, Hansen O, Kampenes VB, Karahasanović A, Liborg N-K, Rekdal AC (2005) A survey of controlled experiments in software engineering. IEEE Trans Softw Eng 31(9):733–753
Steele CM, Aronson J (1995) Stereotype threat and the intellectual test performance of African-Americans. J Personal Soc Psychol 69:797–811
Tichy WF (1998) Should computer scientists experiment more? 16 excuses to avoid experimentation. IEEE Comput 31(5):32–40
Tichy WF, Lukowicz P, Prechelt L, Heinz EA (1995) Experimental evaluation in computer science: a quantitative study. J Syst Softw 28(1):9–18
Walther JB (2002) Research ethics in internet-enabled research: human subjects issues and methodological myopia. Ethics Inf Technol 4(3):205–216
Wickelgren WA (1977) Speed-accuracy tradeoff and information processing dynamics. Acta Psychol 41(1):67–85
Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. Springer
Yin RK (2003) Case study research: design and methods. Sage Publications
Zannier C, Melnik G, Maurer F (2006) On the success of empirical studies in the international conference on software engineering. ACM/IEEE International Conference on Software Engineering, 341–350

Andrew Ko is an Assistant Professor at the University of Washington Information School and an Adjunct Assistant Professor in Computer Science and Engineering. His research areas are human-computer interaction, computing education, and software engineering. His research specifically focuses on software defects and how people and society discover, diagnose, repair and recover from them, spanning everyone from the people use software to the people who develop it. He is the author of over 60 peer-reviewed publications, 6 of which received best paper awards. In 2010, he was awarded an NSF CAREER award to support his research and teaching on evidence-based bug triage. He received his Ph.D at the Human-Computer Interaction Institute at Carnegie Mellon University in 2008. He received degrees in Computer Science and Psychology from Oregon State University in 2002.

Thomas LaToza is an Asst. Project Scientist in the Department of Informatics at the University of California, Irvine. His research has included a number of studies of software developers, published in conferences such as ICSE and FSE. His current research focuses on information needs in software development, software design at the whiteboard, and crowdsourcing software engineering. He has degrees in psychology and computer science from the University of Illinois and a PhD in software engineering from Carnegie Mellon University.

Margaret Burnett is a professor in Oregon State University's School of Electrical Engineering and Computer Science. She has over 20 years experience in research that includes empirical components. Her current research focuses on end-user programming, end-user software engineering, information foraging theory as applied to programming, and gender differences in those contexts. She is also the principal architect of the Forms/3 and the FAR visual programming languages and, together with Gregg Rothermel, of the WYSIWYT testing methodology for end-user programmers. She was the founding project director of the EUSES Consortium, is currently on the Editorial Board of ACM Trans. Interactive Intelligent Systems and of IEEE Trans. Software Engineering, is on the ACM/IEEE ICSE'14 (International Conference on Software Engineering) Program Board and the ACM DIS '14 (ACM Conference on Designing Interactive Systems) Program Committee, and has in prior years served on a variety of other ACM and IEEE conference program committees, chairing a few of them. She also co-chairs the Academic Alliance of the National Center for Women In Technology (NCWIT), and her photo includes the NCWIT signature "red chair".