TestLoop: A Process Model Describing Human-in-the-Loop Software Test Suite Generation

MATTHEW C. DAVIS, Carnegie Mellon University, USA
SANGHEON CHOI, Rose-Hulman Institute of Technology, USA
AMY WEI, University of Michigan, USA
SAM ESTEP, Carnegie Mellon University, USA
BRAD A. MYERS, Carnegie Mellon University, USA
JOSHUA SUNSHINE, Carnegie Mellon University, USA

There is substantial diversity among testing tools used by software engineers. For example, fuzzers may target crashes and security vulnerabilities while Automatic Test sUite Generators (ATUGs) may create high-coverage test suites. In the research community, test generation tools are primarily evaluated using metrics like bugs identified or code coverage. However, achieving good values for these metrics does not necessarily imply that these tools help software engineers efficiently develop effective test suites. To understand the test suite generation process, we performed a secondary analysis of recordings from a previously-published user study in which 28 professional software engineers used two tools to generate test suites for three programs with each tool. From these 168 recordings (28 *users* × 2 *tools* × 3 *programs*/*tool*), we extracted a process model of test suite generation called TestLoop that builds upon prior work and systematizes a user's test suite generation process for a single function into 7 steps. We then used TestLoop's steps to describe 8 prior and 10 new recordings of users generating test suites using the Jest, Hypothesis, and NaNofuzz test generation tools. Our results showed that TestLoop can be used to help answer previously hard-to-answer questions about how users interact with test suite generation tools and to identify ways that tools might be improved.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Software testing and debugging**; • **Human-centered computing → HCI theory, concepts and models**; **User studies**; **User studies**.

ACM Reference Format:
Matthew C. Davis, Sangheon Choi, Amy Wei, Sam Estep, Brad A. Myers, and Joshua Sunshine. 2024. TestLoop: A Process Model Describing Human-in-the-Loop Software Test Suite Generation. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2024), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Estimates of software testing effort range from 28% [17] to 50% [106] of software engineering labor. Generating tests may be "among the most labour-intensive tasks in software testing," [8] yet software engineers largely generate tests manually [11, 36, 40, 62]. **Test generation tools** often intend to help **users**—in this case, software engineers—efficiently generate and run effective tests. But other researchers say that existing tools have usability problems [11, 18, 42, 66, 75, 85, 92, 101] and that improving usability requires understanding the steps that users perform [13, 55, 70], where **steps** are groups of one or more activities performed by a user to achieve a goal [55].[1] While there

[1]ISO [55] uses the term, "tasks." Norman [77, 78] uses the term, "stages." Ko et al. [57, 61] use the term "activities." We call them "steps" for consistency within this article and to avoid confusion with, e.g., user study tasks.

Authors' addresses: Matthew C. Davis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, mcd2@cs.cmu.edu; Sangheon Choi, Rose-Hulman Institute of Technology, Terre Haute, Indiana, USA, chois3@rose-hulman.edu; Amy Wei, University of Michigan, Ann Arbor, Michigan, USA, weia@umich.edu; Sam Estep, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, estep@cmu.edu; Brad A. Myers, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, bam@cs.cmu.edu; Joshua Sunshine, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, sunshine@cs.cmu.edu.

2024. 1049-331X/2024/1-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

is much testing research, "less seems to be known about software testing than about any other aspect of software development" [72], and little prior work attempts to characterize what steps a user may *actually* perform to generate tests for a test suite. Consequently, academic, industrial, and community researchers, who research, design, or build test generation tools, are left with important questions that can be hard to answer, e.g., where are users spending their time, how does a user's effort on a particular step compare among tools, which steps might benefit from better tool support, and did my new idea *actually* help users or did it simply move work to another step?

Given these important gaps in knowledge, we aimed to answer the following questions:

Research Questions (RQ)

(RQ1) What steps might a user perform to generate a test suite for a single function?

(RQ2) To what extent can the steps help describe user test suite generation sessions?

Theories are the "building blocks" of scientific knowledge that help explain "how and why certain phenomena occur and allow predictions to be made" within a specific context [35] and provide a useful lens for studying rich and complex phenomena [21, 90]. A **process model** is a specific type of theory that seeks to describe *what* steps may be performed, *who* performs the steps, and *why* those steps are performed. ISO's Usability Definitions and Concepts (9241-11:2018) [55] emphasizes that one must first understand the *what*, *who*, and *why* of the tool's use in order to understand a tool's **usability**, which is the "extent to which a system, product, or service can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction" [55].

Norman's **seven stages of action** [77, 78] is a widely-used [24] process model that organizes a user's interactions with a system into a looping set of seven steps that users perform in various orders in pursuit of a goal. The steps are: (i) form a goal, (ii) plan to act, (iii) choose and (iv) perform a sequence of activities, (v) perceive and (vi) interpret the state of the system, and (vii) compare the system's new state to the goal. A **gulf** describes "a wide gap, as in understanding" [1]. Norman's process model incorporates two gulfs that he helped identify with Hutchins et al. [54]. **The Gulf of Execution** represents the difference between the set of activities that a user would like to perform in pursuit of their goal and the set of activities actually afforded by the system. After the user performs the activities, **The Gulf of Evaluation** represents the difference between the set of information provided by the system as feedback and the set of information that the user needs to know in order to perceive, interpret, and compare the system's new state to that of the user's goal. Larger gulfs may burden the user with more effort or with more **cognitive load**, which is the demand that an activity places on the user's working memory [34]. Designers care about Norman's model because narrowing the gulfs for common user activities can be an important way to improve user efficiency, effectiveness, and satisfaction. Although our steps and one of our gulfs are distinct from those of Norman's model, we build upon his work by applying his concept of gulfs to the complex domain of test suite generation for a single function.

Many prior models prescribed how users *should* generate test suites (e.g., [6, 9, 72, 79, 84]). Such models are valuable in a wide variety of contexts. However, our research questions made it vital for us to observe what users were *actually* doing in order to build a vocabulary of steps that may describe users' activities while generating a test suite. Therefore, to answer RQ1, we used inductive thematic analysis (**ITA**) [20] to extract a set of 7 abstract steps, which users may perform in various orders, from a rich empirical dataset of recordings in which individual professional software engineers generated a test suite for one of six functions using one of two different test suite generation tools, Jest [102] or NaNofuzz [27]. To answer RQ2, we used those 7 abstract steps to describe a different[2] set of 18 recordings of individual software engineers generating a test suite for one of eight functions using one of Jest [102], NaNofuzz [27], or Hypothesis [69]. We call the process model composed of those steps "TestLoop." Although some of TestLoop's steps may seem obvious or common-sense to some, these particular steps have not previously been described together in the literature.

This article makes the following contributions:

Contributions (C)

(C1) The Gulf of Expectation is a new gulf that we discuss in Sections 4 and 6 and which contextualizes the user's encounter with the **oracle problem**—or determining what the program is *expected* to do or not do—within the test suite generation process.

(C2) TestLoop is a model of the test suite generation process for a single function that we extracted from empirical observations of software engineers and that builds upon prior work. In Section 4, we describe 7 steps that a user performs to generate a test suite. We discuss in Section 6 how the steps may be used to help answer some otherwise hard-to-answer questions.

(C3) Our **evaluation** of TestLoop in Section 5 uses TestLoop's 7 steps to successfully describe recordings of software engineers generating test suites using the Jest, NaNofuzz, and Hypothesis tools. This evaluation provides evidence that TestLoop's 7 steps might be useful to researchers who would like to understand how a user interacts with test suite generation tools.

TestLoop aims to describe the process of a single user generating a test suite for function testing of a single function, where **function testing** means attempting to find differences between the program's actual and expected behavior [72]. Thus, our model may differ in important respects from industrial-scale software testing processes involving, e.g., many users or many functions. While there may be similarities among function testing and the related processes of user, integration, system, installation, and acceptance testing, as defined by Myers et al. [72], we do not investigate these *other* testing processes here. In software testing, an **oracle** determines whether the output of a **program under test (PUT)** is correct. The observations on which TestLoop is based include three types of oracles—property-based, implicit-based, and example-based—which we describe in Section 2. While some aspects of our findings might apply to *other* types of oracles, we do not attempt to investigate that here. While this article represents an appropriate first study of TestLoop, more work is required, and in Section 7 we propose further studies to evaluate and extend TestLoop for use in further and more complex contexts.

2 HOW WE DESCRIBE TEST GENERATION TOOLS IN THIS ARTICLE

Test generation tools are growing more sophisticated, and the boundaries that once divided tools into distinct types are becoming more blurred. For example, JQF [81] randomly generates inputs that maximize code coverage and may also evaluate output correctness. Thus, JQF might be called any of: transparent-box fuzzer, semantic fuzzer, coverage-guided fuzzer, constraint-based, property-based, metamorphic, hypertester, parameterized tester, and so on. Within this article, we use the following three dimensions to describe test generation tools:

(TDO) Test oracle dimension. Test generation tools evaluate specific example executions of a PUT to determine whether the test should pass or fail. There are multiple types of oracle, e.g., a **property-based oracle** describes a general property of the PUT's inputs and outputs that should

[2]The set of recordings used to answer RQ2 is disjoint from the set used to answer RQ1.

hold across multiple executions of the PUT; an **implicit-based oracle** is a weaker form of a property-based oracle that considers only the PUT's outputs (e.g., that the PUT not crash); an **example-based oracle** describes a set of example inputs and the corresponding set of expected outputs; a **regression-based oracle** is a special type of example-based oracle where the current behavior of the PUT is used as a source of examples such that future versions of the PUT might be compared to the present version of the PUT's behavior.

(TDG) Input guidance dimension. Test generation tools generate test inputs that, e.g., are randomly sampled from the PUT's input domain (**random-guided**), are manually selected by a human (**human-guided**), are generated by a Large Language Model (**LLM-guided**), increase the amount of code executed during testing (**coverage-guided**), or detect buggy changes prospectively inserted into the PUT during testing (**mutant-guided**).

(TDP) Test persistence dimension. Tools that automatically generate tests without creating persistent test suites are **Automatic Test Generators (ATGs)**. Tools that create persistent test suites that may be executed again in the future are called **Test sUite Generators (TUGs)**.

Test generation tools may be characterized by TDO + TDG + TDP. For example: JQF [81] may be called a property-based coverage-guided ATG; Jest [102] and JUnit [105] may be called example-based human-guided TUGs; Hypothesis [69] may be called a property-based random-guided TUG; and EvoSuite [38, 39] may be called a regression-based coverage/mutant-guided TUG.

While *further* dimensions and descriptors are possible, the three above are sufficient for this article.

3 BACKGROUND & RELATED WORK

Prior process models of test suite generation. Psychology researchers have modeled the problem solving process as a refining cycle, or loop, of steps that repeat until a goal is achieved or an end state is reached [19, 49, 74, 86]. Norman's influential seven stages of action model [77, 78] describes user interactions with a system as a cycle of repeating steps that allows a user to backtrack or skip steps. Enoiu et al. [37] argued that testing is a form of problem solving and modeled users' cognitive processes while testing as a cycle of repeating steps. Aniche [9] described test suite generation as "an iterative process" and suggested that users repeat a set of steps to generate an effective test suite. Aniche et al. [10] in a think-aloud study noted that users generating test cases iteratively refined a mental model of the program as they added test cases and interpreted test results. Beizer [16] described testing as a "continuing process" of "creating, selecting, exploring, and revising." These prior works indicate that the test suite generation process might be modeled as a repeating cycle of steps that may include backtracking and skipping among the steps.

Prior process models that prescribed how users *should* generate a test suite exhibit diversity in which aspects of the process they model as well as how they choose to divide the process into steps. For example, Myers et al. [72] prescribed how a user should design test cases and divided the process into two steps: (1) identify equivalence partitions and (2) identify the test cases. The latter step included various techniques such as boundary-value analysis, cause-effect graphing, or error guessing, but the model did not include steps to automate or run the test cases with a testing tool. Pezzè and Young [84] divided the test case design process into three steps and added a fourth step to automate, but not run, the tests cases with a testing tool: (1) identify testable features, (2) identify representative values (or derive a model), (3) specify test cases, and (4) automate tests. Ostrand and Balcer [79] presented a model containing six steps: (1) analyze specification, (2) identify choices, (3) determine constraints among choices, (4) write test specification and run tool, (5) evaluate tool output, and (6) automate tests. This model presumes the use of an automated tool and therefore focuses on choosing specific test cases. The model does not include a step for running the tests. The three preceding models focus primarily on test case design and exclude activities that may be particularly important to test tool researchers, such as running the test suite, understanding the test results, and identifying interesting test results.

More recent process models have covered more steps that users *actually* perform while generating test suites. For example, Ammann and Offutt [6] elegantly described a process comprised of four steps: (1) test design, (2) test automation, (3) test execution, and (4) test evaluation. The first two steps incorporate the test case design and test automation activities of prior models.[3] The last two steps model the user's execution of the test suite and the user's review of the test results, which are important steps that were overlooked in previous and some subsequent models. While elegant, the model's small number of steps may obscure details that test tool researchers may find important. For instance, step 1 includes many complex activities that are distinct steps in other models, such as understanding expected behavior and choosing what to test. Aniche [9] described an important recent model that prescribed seven steps for users to follow to generate an effective test suite: (1) understand requirements, (2) explore program, (3) identify equivalence partitions, (4) analyze boundaries, (5) specify test cases, (6) automate test cases, and (7) augment test cases. Steps 2 and 7 are not in the models we described above and may provide important guidance to users who need to generate an effective test suite. However, this model does not include steps that researchers may find important to describe, such as running the test suite, understanding the test results, and identifying interesting test results. While the steps included in prior models vary, none include a step to collect information about the program's expected behavior, none include a connection to gulfs that tool designers find useful, none report being based on empirical observations, and none say to what extent they describe steps users *actually* perform when generating a test suite.

Despite their diversity, models of test suite generation share similar steps since they describe the same underlying process. For example, the models of Ammann and Offutt [6], Aniche [9], Ostrand and Balcer [79], Pezzè and Young [84] all include a similar step for updating the test suite that is subsequently executed by a test automation tool. The sharing of similar-looking steps among models does not necessarily make any particular model less important. Rather, the sharing of a step may instead emphasize the importance of that particular step.

Processes related to test suite generation. Many processes *related* to or *adjacent* to test suite generation have been studied by other researchers, such as:

- **Debugging.** An interesting test result may cause the user to debug a test case, the PUT, or both to understand more about their respective actual behaviors [72]. Zeller [115] prescribed practical debugging methods for users to reproduce, isolate, and find causes of unexpected behavior. Hale and Haworth [46] integrated the prior debugging models of Gould [44] and Vessey [108] into a cognitive model of debugging based on the structural learning theory of Scandura [96]. Later, Hale et al. [47] found support for this cognitive model of debugging in a user study. Despite these theoretical improvements, debugging tools and practice remained largely unchanged [67]. Ko and Myers [59] observed and identified the importance of users' "why" and "why not" questions during debugging and the lack of corresponding support for such questions in contemporary debugging tools. From these findings, Ko and Myers designed and built a new class of prototype debugging tools, called Whyline, for the Alice and Java programming languages that provided support for asking and answering "why" and "why not" questions. In separate user studies, both tools were shown to help users complete more tasks and to complete tasks more quickly than with traditional debugging tools [59, 60].
- **Requirements mining.** Manually creating a specification might be impractical or tedious, especially in cases involving complex interactions among software components. Ammons et al. [7] and others, such as Alur et al. [4], described requirements mining approaches that may

[3]Excepting Myers et al. [72], which did not include a step to automate test cases.

automatically infer a specification from actual program executions. Such specifications may be manually refined and used as a source of expected behavior in the test suite generation process.

- **Software documentation.** Parnas and Vilkomir [83] explained that precise software documentation allows testing to begin sooner, helps to distinguish correct from incorrect results, and aids functional partitioning in support of testing. However, some who try to use program documentation say that it can be of poor quality [2, 3, 107], which can lead to inefficiency and mistakes [89, 91, 107]. Imprecise or missing documentation may become apparent when a user tries to understand expected behavior using the documentation. Consequently, documentation may need to be created, revised, or annotated to address problems identified during testing.
- **Program comprehension.** While testing, the user may need to comprehend aspects of the PUT as well as the test suite, which is often represented as code. However, understanding what a program does and how it does what it does is a complex cognitive process [65, 100]. The ways in which users comprehend programs may be modeled in various ways. For instance, in top-down models, users apply their extant application domain knowledge to structure and contextualize familiar-looking code according to an appropriate mental model that they already possess [113]. In bottom-up models, users encountering unfamiliar-looking code iteratively build and refine a new mental model of the code by trying to understand its control and data flow [113]. Users have been observed to use a combination of both models [111–113], and Shaft and Vessey [98] suggested that users may prefer to use top-down strategies when possible because they require less time and cognitive load. Ko et al. [61] modeled user program comprehension activities as "searching, relating, and collecting" information about the code such that these activities might describe both top-down and bottom-up approaches.
- **Information search and retrieval.** Users may formulate and try to find answers to their questions [95] among the PUT's specification, documentation, bug reports, and other information. The interactions necessary to search for, retrieve, and understand the returned information carry costs for the user (e.g., time, attention, and cognitive load) that may influence user behavior [12].
- **Program maintenance.** Program maintenance focuses on correcting, adapting, or improving an existing program [56, 104] and is related to testing because automated test suites are often represented as programs that the user must maintain. Additionally, a user may identify a bug during the course of testing and decide to fix it.

The need for process models based on user observations. Describing the steps that users of software engineering tools are *actually* performing can impact practice in important ways. For instance, Ko et al. [57, 61] used an inductive qualitative analysis method to extract a set of steps from recordings of users performing program maintenance tasks within the Eclipse Integrated Development Environment (**IDE**). The extracted steps allowed the researchers to describe how study participants spent a surprising 35% of their time simply navigating and scrolling among code. Based on their observations, the authors proposed time-saving code navigation interventions that are now widely adopted in modern IDEs.

Fraser et al. [40] evaluated the EvoSuite [39] regression-based coverage-guided TUG against the JUnit [105] example-based human-guided TUG and found that users did not find more bugs with EvoSuite. This negative result was surprising considering that EvoSuite offered users a high level of automation relative to JUnit. The study's authors analyzed users' post-survey data to posit that poor test case readability *might* be a negative factor in the result. This explanation motivated many researchers (e.g., [45, 73, 82, 94, 97]) to make important contributions to improve generated test case readability over the next decade. Despite these researchers' successes, tools like EvoSuite remain relatively unused in practice, according to one of EvoSuite's authors [11].

In our own **NaNofuzz User Study** [27], NaNofuzz users were more effective, efficient, and confident at generating bug-finding test suites than Jest [102] users for a set of functions. We also relied on post-survey data to explain the reasons for NaNofuzz' positive effects. However, we also wanted to explain our results in more detail by clearly describing what *actually* happened in the user sessions—e.g., how did users spend their time differently in each tool? Clearly users were performing *some* steps to generate test suites, and NaNofuzz provided better support for *some* of those steps. But we could not easily answer our questions using only post-survey data.

The user study results above suggest that a process model for test suite generation based on *actual* user observations might help researchers explain empirical results because such a model would provide a common vocabulary and a set of steps with which to describe what *actually* happened. In the remainder of this article, we discuss our work to extract and then use such a process model.

4 EXTRACTING THE ABSTRACT STEPS (RQ1)

To answer RQ1 ("What steps might a user perform to generate a test suite for a single function?"), we extracted a set of abstract steps that might describe *how* engineers generate a test suite. One possible approach was to survey or interview users to ask *how* they generate test suites, but Norman [77] cautions that "most of human behavior is a result of subconscious processes" and that "many of our beliefs about how people behave—including beliefs about ourselves—are wrong." Consequently, we decided to build our process model using close empirical observations of multiple users generating test suites for multiple PUTs using multiple tools in order to maximize our model's potential usefulness. Human evaluations in software engineering are rare [58], and recruiting a large number of professional software engineers can be difficult [5, 14, 26, 58]. Baltes and Ralph [14] suggests selecting "accessible, information-rich cases, sites, organizations or contexts from which researchers can learn about their topic of study." Our recently-published NaNofuzz user study [27] provided a rich data set of recordings that met these requirements. However, before deciding whether to use data from a prior study, it is important to understand how the weakness and potential biases of the previous study might *also* affect the subsequent study [26]. We discuss the potential weaknesses and biases in Section 8 and summarize the design of the study below.

4.1 NaNofuzz User Study

We designed and performed a randomized controlled human trial [93] with a between-subjects design in which 28 professional TypeScript software engineers each generated 3 test suites using a control treatment (Jest [102]) and 3 test suites using an intervention treatment (NaNofuzz [27]). Figure 1 provides an overview of the study, and Table 1 lists the task programs. The study aimed to answer three variants of the question: *"Relative to standard practice (e.g., Jest), to what extent may NaNofuzz affect X?"* where the three values of X were:

- X = the number of bugs an engineer *accurately* identified
- X = the engineer's *time* on the testing task
- X = the engineer's *confidence* in the test activity

We designed the study to last no more than 90 minutes so that professional software engineers might be more likely to participate [26]. A think-aloud protocol provides insights into the user's thoughts but can make timing data imprecise due to the user's need to verbalize their thoughts [76]. Because this study needed precise timing data, a think-aloud protocol was not appropriate.

4.1.1 *Participants.*

We provide a summary description of the participants below and provide more details within the appendices. We recruited professional software engineers into the user study via LinkedIn, Mastodon, Twitter, and e-mail. The recruiting messages described the study and included a link to the screener survey, which screened for participants: (i) in the United States or Canada (as

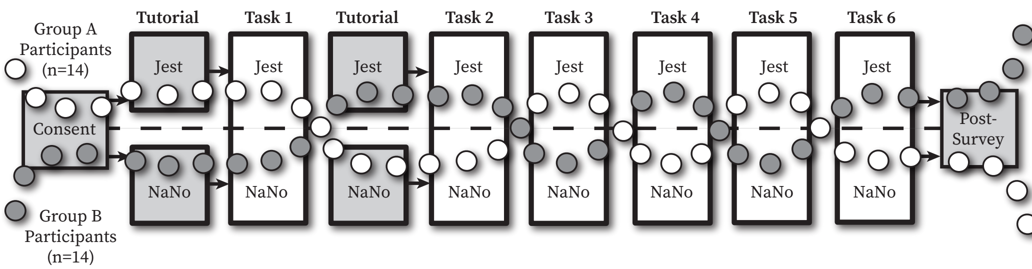

Fig. 1. NaNofuzz User Study sequence (see Section 4.2). Participants were randomly assigned by pairs to Group *A* or *B*, which determined each task's treatment. Shaded activities (e.g., tutorials) were not timed.

Table 1. Programs Under Test, by user study

Task Number	Program Name	Origin of Program	Lines of Code	Error Class	Number of Bugs	Time Limit
NaNofuzz User Study (Section 4.1)						
1	6.ts	Stack Overflow	3	Exception	1	15 minutes
2	3.ts	Stack Overflow	4	NaN output	1	15 minutes
3	7.ts	Stack Overflow	12	Divide by zero	1	15 minutes
4	11.ts	Rosetta Code	14	Exception	1	15 minutes
5	14.ts	Rosetta Code	15	Infinite loop	1	15 minutes
6	10.ts	GitHub	57	NaN output	2	15 minutes
Hypothesis User Study (Section 5.1)						
1	ex11.py	Rosetta Code	5	Unexpected alias	1	30 minutes
2	ex12.py	Rosetta Code	15	Logic error	1	30 minutes

required by our IRB), (ii) who were over 18, (iii) had at least one year of professional programming experience, and (iv) had programming experience with TypeScript. Qualified participants were offered a $30 Amazon gift card, and no bonuses were offered. The screener included timed questions recommended by Danilova [25] to eliminate non-programmers.

Participants were assigned to groups *A* and *B* using matched pair random assignment based on a physical coin flip and participants' self-reported professional coding experience: 1–5 years, 6–10 years, and 11+ years. When a participant scheduled a session, they were assigned a participant number, and a researcher checked to see if a previous participant with the same experience level was awaiting a match. If no participant with the same experience level was awaiting a match, the researcher flipped a physical coin to determine the participant's group, and the participant was flagged as needing a match. When the next participant with the same experience level scheduled a time slot, the new participant would be matched to the previous one such that one participant would be randomly assigned to group *A* and the other randomly assigned to group *B*.

Participant demographics were as follows: 5 participants identified as female, 21 as male, and 2 did not disclose; 4 participants had 10+ years of professional experience, 4 had 6–9 years, and 20 had 1–5 years; 11 participants reported spending 30+ hours coding per week, 13 reported spending 10–29 hours per week, and 4 reported 5–10 hours per week.

4.1.2 Treatments. The user study included two treatments:

Jest [102] is an example-based human-guided TUG for the TypeScript and JavaScript programming languages. Participants in the study interacted with Jest by writing test cases in TypeScript or JavaScript code and running Jest tests via a button in the IDE. As of 2023, Jest was a state-of-the-art TUG with 50 million monthly downloads, was used in over 3.8 million public GitHub repositories, and was used by Meta (Jest's creator), Twitter, Spotify, Airbnb, and other companies [102].

NaNofuzz [30] is an implicit-based[4] random-guided TUG for the TypeScript programming language. Users in the study interacted with NaNofuzz through a Visual Studio Code extension that asked the user for information about the PUT's input domain and then randomly generated example test cases that the participant could add to the persistent test suite by clicking an IDE button. NaNofuzz organized test cases and test results into a tabbed grid that prioritized display of test cases that might be more likely to indicate a bug.

4.1.3 Tasks. As shown in Figure 1, each participant completed six timed instances of the same test suite generation task, where each task instance varied the PUT according to the study protocol and varied the treatment according to the participant's random group assignment (*A* or *B*). Prior to using each treatment in a task, each participant completed a non-timed tutorial that included two exercises that showed them how to use the tool. The PUTs used in the study are shown in Table 1 and are available in the NaNofuzz study's supplemental material [28]. At the beginning of each task, a researcher read from a script and verbally instructed the participant to open the PUT in the IDE and explained that the goal of the task was to find inputs to the program that cause any of the following results: null, undefined, **NaN** (Not-a-Number), infinity, a runtime exception, or apparent non-termination. The researcher specified which treatment to use and, if Jest, directed the participant to open the Jest test file "to the side" so that both the tests and the code under test were simultaneously visible. After providing the instructions and the participant indicated they were ready to begin, the researcher manually recorded the start time. Using Zoom, the researcher monitored each participant's screen and audio to ensure use of the intended treatment and program. During the task, the participant tested the program using the designated treatment and created test cases. At the end of 15 minutes or when the participant said they were done, the researcher recorded the stop time and verbally instructed the participant to complete a post-task survey where they typed up their understanding of the generalized input domains where bugs occurred and their confidence in the testing activity according to a 5-point Likert scale. By using both treatments, each participant could provide comparative feedback about the two treatments at the end of the study.

4.1.4 Data Collected. In this article, we used the following data collected by the user study:

(D1) Jest and NaNofuzz Session Recordings. We used Zoom to collect a total of 168 screen and audio recordings of 28 professional software engineers generating test suites for six different PUTs using 2 different tools (3 PUTs each for Jest and NaNofuzz) within the Visual Studio Code IDE.

4.2 Data Analysis & Results

We performed the following analysis, which are summarized in Table 2:

(A1) Extract Abstract Steps. The second author randomly selected participants from the underlying data set using the Google Random Number Generator (**RNG**) [43] and performed open coding of activities in the recordings selected. Random selection and analysis continued until the second author stopped identifying new activities in the recording data. The list of participants selected was P14, P18, and P35.[5] The first and second authors iteratively organized these activities into discrete trial themes and then tested and refined the themes against the sampled recording data until each

[4] Subsequent versions of NaNofuzz added support for *additional* oracles.

[5] We describe how we used some *other* participants' data from this study in Section 5.2.

Table 2. Data analyses, inputs, and results used in this article

ID	Analysis Input(s)	Analysis Method	Analysis Result(s)
Extracting the Abstract Steps (Section 4, RQ1)			
A1	Jest, NaNofuzz Session Recordings (D1)	Inductive Thematic Analysis [20]	Coding Guide (R1) Abstract Steps (R2)
Applying the Abstract Steps (Section 5, RQ2)			
A2	Jest, NaNofuzz Session Recordings (D1) Hypothesis Session Recordings (D2)	Coding Guide (R1)	Step Transcripts (fixed) (R3)
A3	Step Transcripts (fixed) (R3)	See Section 5.2, A3	Step Transcripts (variable) (R4)
A4	Step Transcripts (fixed) (R3)	Cohen's Kappa [93]	Inter-rater Reliability (R5)
A5	Step Transcripts (variable) (R4)	See Section 5.2, A5	Abstract Step Transitions (R6)
A6	Step Transcripts (variable) (R4)	See Section 5.2, A6	Complete Loop Iterations (R7)
A7	Step Transcripts (variable) (R4)	See Section 5.2, A7	Step Summaries (R8)

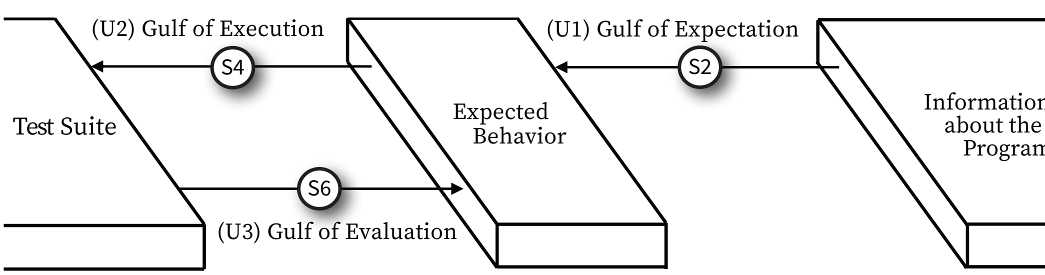

Fig. 2. Overview of TestLoop's three gulfs. The user must cross: The Gulf of Expectation (U1) in step Understand Expected Behavior (S2) to determine the PUT's expected behavior using the information about the program, The Gulf of Execution (U2) in step Update Test Suite (S4) to encode the expected behavior into the tool's required test suite representation, and The Gulf of Evaluation (U3) in step Understand Test Results (S6) to determine differences among the test results and what the user is expecting.

theme was clearly defined relative to the other themes and was supported by the underlying video data. We did not assume that steps might follow a particular order, and Section 5.2 (R6) shows that steps actually occurred in various orders. To identify a gulf, researchers should point to an existing gap of understanding that the user must cross (e.g., the oracle problem [15]) and then show how re-framing the gap as a gulf might be useful.[6] Thus, the first and second authors then compared each of the Abstract Steps (R2) to Norman's descriptions of gulfs [54, 77, 78] to identify: (1) steps that may require the user to cross a gap in understanding and (2) which of those steps may be useful to re-frame as a gulf. The results are reported below in the Coding Guide (R1, Table 3) and Abstract Steps (R2).

The results of the data analysis are below. To avoid confusion, here we numbered the Abstract Steps according to the way we describe them in Section 5.2.

(R1) Coding Guide. The coding guide is provided in Table 3.

(R2) Abstract Steps. The 7 Abstract Steps are described in Table 3. Table 4 illustrates a particular instantiation of the steps observed within a user session. Completing the Abstract Steps involves

[6]Subramonyam et al. [103] is a recent example of identifying a new gulf by pointing to existing gaps in understanding and showing how re-framing the problem as a gulf is useful.

Table 3. Coding Guide (R1) produced by Inductive Thematic Analysis (A1)

Abstract Step (Theme)	**Abstract Step Description / Activities**
S1. Collect program information	Collect information about the PUT's expected behavior • Gather information to prepare for testing • Seek sources of additional information about the program
S2. Understand expected behavior†	Understand expected behavior from the PUT information • Read program information or source code • Move cursor between test suite and program code • Run initial unmodified test suite & review results • Ask questions regarding expected behavior
S3. Choose scenarios to test	Choose what expected behavior to test • Specify cases to test (verbal or text) • Specify boundaries/ranges/types to test
S4. Update test suite†	Modify the test suite to test the chosen expected behavior • Add/remove/edit test cases • Edit input and output values
S5. Collect test results	Collect observations about the PUT's actual behavior • Execute tests, e.g., by pressing `Test` button)
S6. Understand test results†	Understand actual behavior relative to expected behavior • Read / scroll through test case results
S7. Choose interesting test results	Choose test results that may warrant further investigation • Note an interesting or unexpected test case (e.g. test fails when expected to pass), mark or save test • Compare test case outputs to PUT description • Use test case as a basis for the next test • Change emotional state (e.g., verbal cues)

Abstract Steps are numbered according to Section 5.2; †=Step involves crossing a gulf shown in Figure 2
Observations of S1 are implicit in this study (see Section 4.2, S1)

crossing 3 gulfs, which are shown in Figure 2. We show TestLoop's steps in relation to the prescriptive process models of prior work in Table 5.

Below we discuss the 7 Abstract Steps (R2) that describe how the sampled users generated a test suite for a single function as well as 3 Gulfs that users must cross to complete these steps. Table 5 summarizes how TestLoop's steps relate to those of prior process models of test suite generation.

(S1) Collect program information. To differentiate expected from unexpected behavior, a user requires *some* information about the PUT. If the information exists, it may be time-consuming to locate and/or might be inaccurate [64]. If the information does not exist, then the related processes of requirements mining and software documentation might be relevant (see Section 3). In the underlying controlled user study data, it was necessary to provide to the user the necessary program information in order to avoid confounds that might arise if the user were required to, e.g., perform internet searches to find information about the program. This means that during our inductive thematic analysis, we saw the effect of this step having been performed, but we could not code it. Given that the step was clearly happening and we could see its effects, we decided to include S1

Table 4. Description of Abstract Steps (R2) and their observation in session P16 Task 6 (NaNofuzz)

Begin Time	End Time	Abstract Step	Observed User Activities
0:00	0:00	S1†	• At the session's start, the researcher had provided the program's natural language specification to the user at the top of the PUT's source file
0:00	0:18	S2	• Clicked `Test` button to run initial unmodified test suite • Moved cursor to test results & hovered over individual test results
0:19	0:29	S7	• Noted a test with a *null* output using NaNofuzz' `Pin` button
0:30	2:29	S2	• Moved cursor between unmodified test suite and program code • Slowly scrolled through PUT's specification & source code • Moved cursor to test results of initial unmodified test suite • Scrolled through & hovered over individual test results
2:30	2:30	S3	• Moved cursor to test input range specification
2:31	2:40	S4	• Changed input array max lengths from 10 to 0
2:41	2:41	S5	• Clicked Test button, which generated one test input, `[]`
2:42	2:45	S6	• Moved cursor to test results & hovered over individual results
2:46	2:46	S3	• Moved cursor back to test input range specification
2:47	2:51	S4	• Changed input array max lengths from 0 to 1
2:52	2:52	S5	• Clicked `Test` button, which generated additional test inputs
2:53	2:55	S6	• Moved cursor to test results & hovered over individual results
2:56	2:59	S7	• Noted a test with a *null* output using NaNofuzz' `Pin` button
...	...	...	...

Times shown as minutes:seconds; †=observations of S1 are implicit in this study (see Section 4.2, S1).

Table 5. TestLoop Abstract Steps (R2) and the steps of prior models of test suite generation

TestLoop	Aniche [9]	Ammann and Offutt [6]	Myers et al. [72]	Pezzè and Young [84]	Ostrand and Balcer [79]
S1	–	–	–	–	–
S2	1	1	1	1	1
S3	3–5, 7†		2	2–3	2-5
S4	6	2	–	4	6
S5	–	3	–	–	–
S6	–	4	–	–	–
S7	–		–	–	–
other	2†	–	–	–	–

†=see note in Section 4.2, R2; See Section 3 for descriptions of the prior models' steps;
S1=Collect program info; S2=Understand expected behavior; S3=Choose scenarios to test;
S4=Update test suite; S5=Collect test results; S6=Understand test results;
S7=Choose interesting test results.

in the coding guide. Outside of a controlled user study, the necessary program information may not be provided to a user; in which case, the user may need to spend time finding documentation, specifications, bug reports, traces, free-form notes, colleagues' input, and so on.

(S2) Understand expected behavior. A machine-readable formal specification that precisely describes a PUT's expected behavior is rarely available [80], and when one is available, the user still needs to understand it. In this step, the user attempts to extract a subset of expected behavior from the PUT information gathered in S1 (Collect program information) and updates their mental model of the PUT's expected behavior. Building a mental model of a program's behavior from available information is complex [10, 68] and may be *further* complicated by inaccurate [3, 64, 116] or missing [2, 3, 88, 99] information about the PUT. Therefore, building a mental model of the PUT's expected behavior may be difficult. **The Gulf of Expectation (U1)** is a newly-identified gulf that the user must cross to build this mental model of expected behavior. Norman's seven-stages of action model [78] implicitly assumes that the user already knows or can readily determine what to expect a system or object to do; however, this is not the case in testing.[7] To cross this gulf, the user must determine what the PUT is expected to do given certain inputs and/or preconditions. Testing researchers call this determination the **oracle problem** [15] and report that correct behavior can be difficult to determine or even be unknowable [15, 48]. By contextualizing the oracle problem within a particular abstract step and gulf, our work helps testing researchers to consider how testing tools might narrow this gulf and, therefore, support users in completing this step by helping the user organize [52], annotate [51, 53], and/or manage [50] various types of program information as well as extract expected behavior [109, 110]. It should be noted that this gulf is distinct from the Gulf of Evaluation, which we discuss later in this section, and which assumes the user already knows *what* behavior to expect and therefore only needs to compare the result they expected to what actually *did* happen. The related processes of software documentation, program comprehension, and information search (see Section 3) may be relevant to this step as the user attempts to extract expected behavior from the assembled program information.[8]

(S3) Choose scenarios to test. In this step, the user chooses which particular aspect(s) of the expected behavior to test next based on the criteria the user finds appropriate at the time. The user must also choose how such behavior might be tested, which can be quite complex, and may require techniques such as boundary-value analysis, cause-effect graphing, and error guessing that are comprehensively discussed in prior works, such as Aniche [9], Ammann and Offutt [6], Myers et al. [72], Pezzè and Young [84], and Beizer [16] to which the reader may refer for in-depth explanations.

(S4) Update test suite. To test the behavior selected, the user must encode appropriate test cases into the tool's required test suite representation, which may vary significantly by tool. Example-based tools such as Jest [102] or JUnit [105] may require the user to modify test suite code that encodes a set of example inputs and outputs from the selected scenario. Some property-based tools such as Hypothesis [69] or Quickcheck [23] require the user to write code that generates inputs as well as code that determines whether each output is correct or incorrect relative to a generated input. **The Gulf of Execution (U2)** [54] describes the semantic and articulatory distance between the user's mental model of expected behavior and the specific way in which the tool requires the user to encode that behavior. Complex representations such as those used in property-based tools may increase the size of this gulf as well the cognitive effort required to cross the gulf. However, if a representation is too simplistic, then the user may be unable to encode important testing scenarios or find it tedious to do so. The related process of program maintenance (see Section 3) might be relevant as the user updates the test suite, which is often represented as code.

(S5) Collect test results. The user presses an IDE button, executes a terminal command, or performs some action that executes the PUT with test inputs and collects the corresponding outputs. The related process of debugging (see Section 3) might be relevant if the user needs to debug a test case or the PUT to answer, e.g., "why" or "why not" questions about actual program behavior.

(S6) Understand test results. In this step, the user reads and attempts to understand the cases in which the PUT's observed behavior agrees and disagrees with the user's mental model of expected

[7]We posit that the Gulf of Expectation may be present in other domains and suggest further studies in Section 7.
[8]The quality of the assembled documentation may also affect the user's ability to test effectively and efficiently [83].

behavior. The user's goal may be to identify bugs in the PUT, the test suite, or both. **The Gulf of Evaluation (U3)** [54] refers to the distance between the user's mental model of expected behavior and the test tool's concrete representation of the test results. Some test tools output test results to the terminal and indicate whether a test case passed or failed. If the information needed by the user is not present in the tool's output (e.g., the inputs and outputs tested), then the user may need to *additionally* gather the missing information elsewhere, e.g., from the encoded test case.

(S7) Choose interesting test results. From among the test results the user understood in S6, the user may choose interesting test results that they believe warrant further attention. "Interesting" does not simply mean "failing": passing tests results may *also* be interesting, such as in the case where a passing test should have failed given that errors may exist in the PUT, in the program information, or even in the test itself [10, 63]. Further, users maintain mental models of programs [10, 16, 64, 68], and the error may exist within the user's own mental model of the PUT. Consequently, users may find *many* types of test results interesting, including test results that presently "pass."

Relation of TestLoop's Abstract Steps (R2) to the steps of prior process models. In Table 5 we show how TestLoop's steps relate to those of prior models. For instance, all prior models cover the steps of understanding the requirements (S2) and choosing scenarios to test (S3). Aniche [9], Pezzè and Young [84] and Myers et al. [72] break choosing test scenarios (S3) into multiple steps that describe in great detail the various ways that the user might choose test cases. All but one prior model, Myers et al. [72], included a step for updating the test suite (S4). Only one prior model, Ammann and Offutt [6], included steps to collect test results (S5), understand the test results (S6), and choose interesting test results (S7). However, the steps of Ammann and Offutt [6] do not distinguish between understanding the test results (S6) and choosing interesting test results (S7), nor does it distinguish between understand expected behavior (S2) and choose scenarios to test (S3), which may be important for testing tool researchers who want to describe how their tools might help users, e.g., organize expected behavior information and find unusual or outlying test results. Aniche [9] included two steps that differed from prior models: steps 2 ("Explore the program") and 7 ("Augment the test suite with creativity and experience"). Step 2 is concerned with exploring the PUT's actual behavior, which, strictly speaking, is not function testing. However, having many examples of actual behavior might help a user build a mental model of actual behavior, which can be helpful to a user who needs to generate a test suite for function testing. Such exploration without regard for expected behavior might be modeled using TestLoop (e.g., steps S3–S7), and tools like NaNofuzz that generate and coherently organize examples of actual behavior might provide automation support for such an exploration activity. Step 7 of Aniche [9] is defined as, "Perform some final checks. Revisit all the tests you created, using your experience and creativity. Did you miss something? Does your gut feeling tell you that the program may fail in a specific case? If so, add a new test case." This step might be modeled in TestLoop as more-creative instances of either S3 (choose test scenarios) or as Complete Loop Instances. Notably, no prior model included a step for collecting information about the program (S1), and no prior model had a connection to gulfs.

In the following section, we attempt to use the 7 abstract steps and 3 gulfs to describe recorded sessions in which users generated test suites with various testing tools.

5 APPLYING THE ABSTRACT STEPS TO TEST GENERATION SESSIONS (RQ2)

In Section 4, we extracted 7 Abstract Steps from user sessions involving two different TUGs, Jest and NaNofuzz. But are these steps useful for analyzing *other* test suite generation sessions, including sessions that do not involve Jest or NaNofuzz? To answer RQ2 ("To what extent can the steps help describe user test suite generation sessions?"), we attempted to describe *additional* Jest and

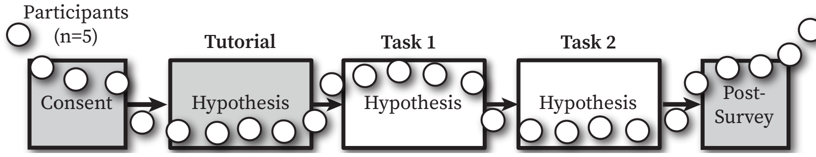

Fig. 3. Hypothesis user study sequence (see Section 5.1). Shaded activities (e.g., tutorial) were not timed.

NaNofuzz test generation sessions that our process model had not been used on previously, as well as sessions involving a popular testing tool, Hypothesis [69].

5.1 Hypothesis User Study

We designed and performed a new user study in which 5 participants generated test suites for 2 PUTs using Hypothesis [69], a popular random-guided property-based TUG for Python (see Section 5.1.2). The study protocol and tasks were adapted from the prior NaNofuzz user study; however, the purpose of this study was to observe users interacting with Hypothesis while generating a test suite rather than to measure the effects of various tools. As shown in Figure 3, the study involved a single treatment, Hypothesis. Study sessions were designed to last less than 90 minutes so that professional software engineers might be more likely to participate [26]. In tools such as Hypothesis, where there is not a particular user interface for choosing test scenarios, a transition from, e.g., S2 (Understand expected behavior) to S3 (Choose scenarios to test) might occur entirely within the user's internal thoughts. Nielsen [76] states that think-aloud protocols are useful for understanding "what the users are doing and why they are doing it while they are doing it." However, an important trade-off of a think-aloud protocol is that timing data may be imprecise due to the participant's need to consider and verbalize their thoughts [76]. Fortunately, answering RQ2 did not require precise timing data. Thus, we chose a think-aloud protocol for this study to help us understand whether or not the abstract steps might actually be a good fit for analyzing Hypothesis sessions.

5.1.1 Participants. We provide a summary description of the participants below and provide more details within the appendices. We recruited participants into the study using announcements to Mastodon, a mailing list of Carnegie Mellon University Professional Software Engineering Masters students, and the Slack for a PhD-level software engineering course in which Hypothesis was taught. The recruiting messages to Mastodon and the mailing list included a link to a screener survey, which screened for participants who were: (i) in the United States or Canada (as required by our IRB), (ii) over 18 years old, and (iii) had programming experience with Python. In the screener survey, potential participants self-reported their gender; professional software engineering experience; hours of coding per week; and experience with Python, property-based testing tools like Hypothesis, and Visual Studio Code. The screener included timed questions recommended by Danilova [25] to eliminate non-programmers. Qualified participants were offered a $30 Amazon gift card, and no bonuses were offered. Demographics for the 5 participants were as follows: 2 participants identified as female and 3 as male; 4 participants had 1–5 years of professional programming experience, 1 reported 0 years; 3 participants reported spending 30+ hours coding per week, 1 reported spending 10–29 hours per week, and 1 reported 5–10 hours per week.

5.1.2 Treatment. **Hypothesis** [69] is a popular [42] and widely-used [69] property-based testing tool inspired by Quickcheck [23]. Other researchers [41, 42, 114] have said that property-based tools like Hypothesis present a unique set of barriers that differ from those of example- or implicit-based tools such as Jest and NaNofuzz. For example, Jest users write code that specifies an example input, calls the PUT with that input, and then compares the actual output with the expected example output.

NaNofuzz randomly generates many inputs from user-specified ranges, calls the PUT using each generated input and tests each output with an implicit-based oracle. In contrast to Jest and NaNofuzz, Hypothesis users write code that generates a large number of inputs to test, calls the PUT with each generated input, and then validates the correctness of each output relative to each generated input. Consequently, Hypothesis users must think in terms of the program's *general properties* that hold for many inputs rather than in terms of specific input and output *examples* of the program's behavior. When displaying test results, NaNofuzz and Jest show many results; however, Hypothesis follows the lead of other property-based tools and, by default, stops testing when the first failure is detected and displays one failing example to the user. Unlike Jest and NaNofuzz, which support programs written in the JavaScript or TypeScript programming languages, Hypothesis supports testing programs written in Python, which IEEE reported was the top programming language of 2023 [22].

5.1.3 Tasks. As shown in Figure 3, each participant completed one non-timed Hypothesis tutorial and two instances of the same test generation task. Each task instance varied the PUT (Table 1) according to the study protocol. To ensure the study could be completed within ninety minutes, we set a 30-minute limit for the two tasks. We utilized Ko et al.'s [58] suggestion to use "found" tasks. We leveraged the 14 "found" PUTs considered for the earlier NaNofuzz study, of which four from Rosetta Code had Python versions (and were not used in the NaNoFuzz user study). Of these 4, we selected 2 that our previous study pilot participants found less confusing. The Python version of `ex12.py` on Rosetta Code appeared to be correct, so we deleted a statement that caused the program to behave contrary to its specification for some inputs. Each program included a description of its expected behavior and its allowed input domain. At the start of each testing task, a researcher provided verbal instructions from a script directing the participant to open the PUT in the IDE and to find allowed program inputs, if any, that cause the program to behave contrary to its specification. The researcher specified that Hypothesis was to be used to generate the test suite and told the participant that the description at the top of the program specified the PUT's allowed inputs and expected behavior. After providing the instructions and the participant indicated they were ready to begin the task, the start time was recorded and the task began. Each participant's screen and audio were monitored and recorded via Zoom to ensure use of Hypothesis and the PUT. During the task, the participant tested the program using Hypothesis and generated test cases. As this study used a think-aloud protocol, the researcher instructed and reminded the participant to verbalize their thoughts and actions (e.g., "please continue"). At the end of 30 minutes or when the participant indicated they were done testing, the researcher recorded the stop time.

5.1.4 Data Collected. We collected and used the following data from the Hypothesis user study:

(D2) Hypothesis Session Recordings. Using Zoom, we recorded a total of 10 sessions, in which 5 software engineers generated test suites for 2 different PUTs using Hypothesis. Each recording included the participant's screen and audio.

5.2 Data Analyses & Results

Below we describe our data analyses, which we summarized in Table 2.

(A2) Code Session Recordings. As raters may agree that a step is taking place but vary slightly by which exact second the step began or ended, raters were instructed to stop the recording every 30 seconds and code all steps observed within the previous 30 second interval. Multiple steps were allowed to be reported per 30-second interval. The second author randomly selected Jest and NaNofuzz Session Recordings (D1)[9] via Google RNG [43]. The recordings used to extract the model

[9]The selected sessions were: P16 sessions 2 & 6, P21 session 1, P22 session 2, P23 session 5, P35 session 2, and P37 sessions 2 & 3. The original sessions used for extracting the steps in Section 4 were excluded from this random selection.

Table 6. Excerpt of Step Transcript (fixed intervals) (R3) for P35 Task 2

Session: P35 Task 2 (Jest)		**Steps observed**						
Begin Time (mm:ss)	**End Time (mm:ss)**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
0:00	0:00	†						
0:00	0:29		X					
0:30	0:59				X			
1:00	1:29		X		X	X		
1:30	1:59				X		X	
2:00	2:29		X		X	X	X	X
2:30	2:59		X					
3:00	3:29		X					
3:30	3:59		X		X			
4:00	4:29		X		X			
4:30	4:59					X	X	X
...	...	...	...	...	...	...	...	...

Times shown as minutes:seconds; †=observations of S1 are implicit in this study (see Section 4.2, S1); S1=Collect program info; S2=Understand expected behavior; S3=Choose scenarios to test; S4=Update test suite; S5=Collect test results; S6=Understand test results; S7=Choose interesting test results.

Table 7. Excerpt of Step Transcript (variable intervals) (R4) for P35 Task 2

Session: P35 Task 2 (Jest)		**Steps observed**						
Begin Time (mm:ss)	**End Time (mm:ss)**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
0:00	0:00	†						
0:00	0:29		X					
0:30	0:59				X			
1:00	1:29				X			
1:07	1:16		X					
1:17	1:22				X			
1:23	1:29					X		
1:30	1:59						X	
2:00	2:10		X					
2:11	2:24				X			
2:15	2:22					X		
2:22	2:28						X	
2:29	2:29							X
...	...	...	...	...	...	...	...	...

Times shown as minutes:seconds; steps where both rater agreed are shown; †=observations of S1 are implicit in this study (see Section 4.2, S1); S1=Collect program info; S2=Understand expected behavior; S3=Choose scenarios to test; S4=Update test suite; S5=Collect test results; S6=Understand test results; S7=Choose interesting test results.

in Section 4 were excluded from this selection. The second and third authors used the Coding Guide (R1) to independently code the abstract steps observed in the Jest and NaNofuzz Session Recordings (D1) and produced 16 Step Transcripts (fixed intervals) (R3) (2 raters × 8 sessions). Random sampling and analysis of Jest and NaNofuzz sessions stopped when both raters reported they had reached saturation. Due to there only being 10 Hypothesis sessions, it was feasible for the

Table 8. Abstract Step Transitions (R6) by tool, current, and next step

Current Step	Next Step S2	S3	S4	S5	S6	S7	Σ
NaNofuzz							
S1	**5 (100%)†**	—	—	—	—	—	5 (100%)
S2	—	**5 (63%)**	1 (13%)	—	—	2 (25%)	8 (100%)
S3	—	—	**18 (95%)**	—	—	1 (5%)	19 (100%)
S4	—	—	—	**20 (100%)**	—	—	20 (100%)
S5	—	—	—	—	**17 (94%)**	1 (6%)	18 (100%)
S6	3 (19%)	**8 (50%)**	1 (6%)	—	—	4 (25%)	16 (100%)
S7	1 (13%)	**6 (75%)**	—	—	1 (13%)	—	8 (100%)
Σ	9	19	20	20	18	8	94
Jest							
S1	**3 (100%)†**	—	—	—	—	—	3 (100%)
S2	—	—	**10 (91%)**	1 (9%)	—	—	11 (100%)
S3	—	—	**1 (100%)**	—	—	—	1 (100%)
S4	2 (13%)	1 (6%)	—	**12 (75%)**	1 (6%)	—	16 (100%)
S5	—	—	—	—	**14 (100%)**	—	14 (100%)
S6	4 (29%)	—	3 (21%)	2 (14%)	—	**5 (36%)**	14 (100%)
S7	**2 (40%)**	—	**2 (40%)**	1 (20%)	—	—	5 (100%)
Σ	11	1	16	16	15	5	64
Hypothesis							
S1	**10 (100%)†**	—	—	—	—	—	10 (100%)
S2	—	9 (20%)	**24 (55%)**	5 (11%)	5 (11%)	1 (2%)	44 (100%)
S3	3 (14%)	—	**17 (77%)**	2 (9%)	—	—	22 (100%)
S4	13 (13%)	10 (10%)	—	**76 (75%)**	2 (2%)	—	101 (100%)
S5	1 (1%)	—	3 (3%)	—	**73 (80%)**	14 (15%)	91 (100%)
S6	15 (19%)	3 (4%)	**47 (60%)**	5 (6%)	—	8 (10%)	78 (100%)
S7	5 (22%)	1 (4%)	**13 (57%)**	3 (13%)	1 (4%)	—	23 (100%)
Σ	47	23	104	91	81	23	369

Bold=most common next step; †=observations of S1 are implicit in this study (see Section 4.2, S1); S1=Collect program info; S2=Understand expected behavior; S3=Choose scenarios to test; S4=Update test suite; S5=Collect test results; S6=Understand test results; S7=Choose interesting test results

third author to code all 10 Hypothesis Session Recordings (D2) using the Coding Guide (R1), which produced 10 Step Transcripts (fixed intervals) (R3). The first author randomly selected 30% (3/10) of the Hypothesis sessions (H01 Task 2, H03 Task 1, and H03 Task 2) using Google RNG [43] and independently coded those sessions to produce an additional 3 Step Transcripts (fixed intervals) (R3).

(A3) Determine Exact Step Timings. Some 30-second intervals captured multiple steps, which obscured the relative sequence among the steps. Thus, the 30-second intervals were insufficient to idenfity the sequence of **Step Transitions**, or specific points in time in which a user stops performing the current step and starts performing the next step. We combined the two raters' Jest and NaNofuzz Step Transcripts (fixed intervals) (R3) such that the 8 merged transcripts included only the steps that both raters observed at each interval for each session and excluded intervals where no agreed-upon steps were recorded. For intervals with multiple step observations, the first author re-watched the recordings for that interval and recorded sub-intervals of variable length

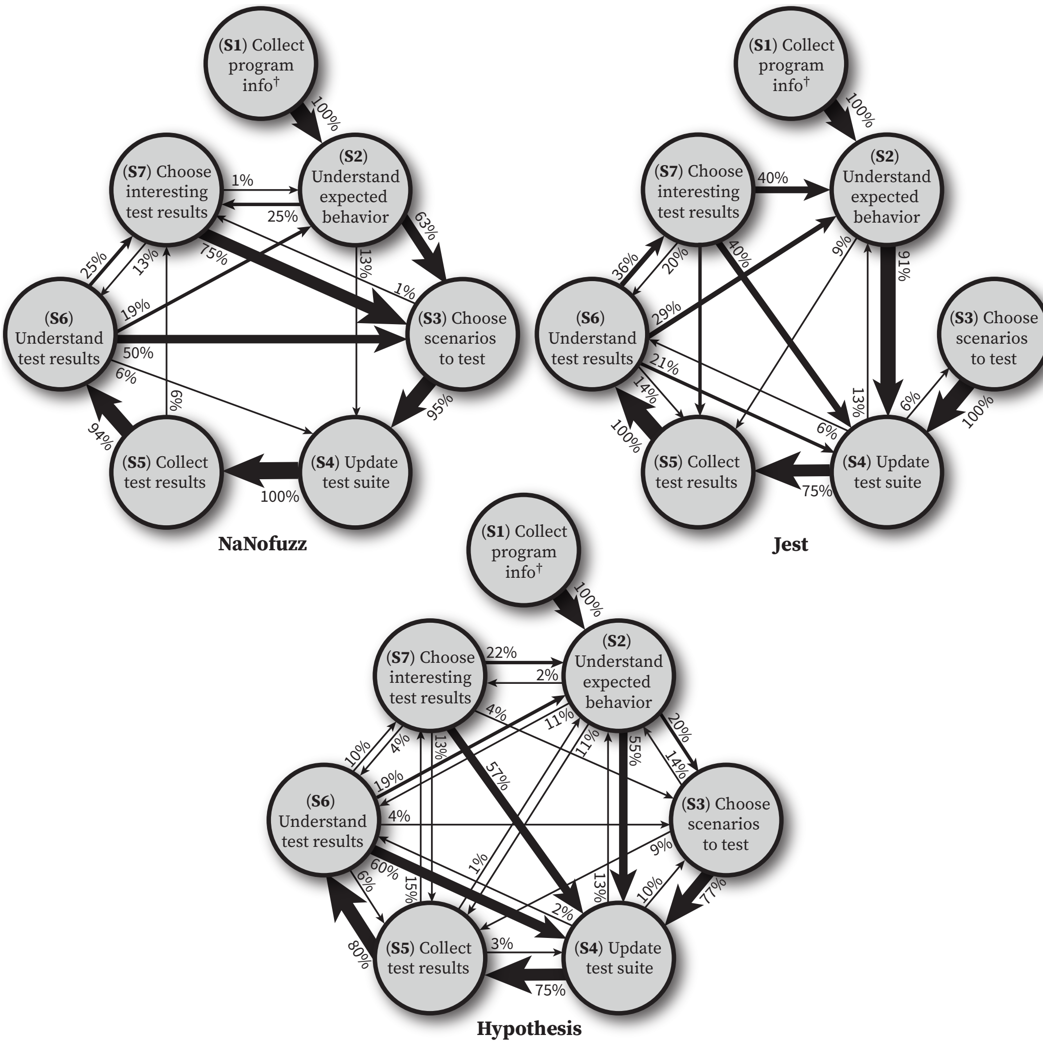

Fig. 4. Abstract Step Transitions (R6) by tool. Weight of arcs is proportional to transition frequency from that step (see Table 4). †=Observations of S1 are implicit in this study (see Section 4.2, S1).

with the exact beginning and ending timestamps of the steps. As in prior analyses, the Coding Guide (R1) was used to identify steps. The third author then randomly sampled 10 of the 40 intervals above[10] using Google RNG [43], independently followed the procedure above, and agreed with the first author's sub-intervals. The procedure above produced 8 Step Transcripts (variable intervals) (R4). The third author then re-watched Hypothesis Session Recordings (D2) for intervals with multiple steps coded in their corresponding Step Transcripts (fixed intervals) (R3) and recorded the exact times for each step to produce 10 Step Transcripts (variable intervals) (R4). The first author

[10]The intervals we re-sampled were: P16 T6 0:00–0:30, P16 T6 4:30–5:00, P16 T6 5:30–6:00, P22 T2 6:00–6:30, P16 T2 5:30–6:00, P16 T2 7:30–8:00, P21 T1 3:30–4:00, P21 T2 4:30–5:00, P37 T2 1:30–2:00, P35 T2 7:30–8:00

Table 9. Complete Loop Iterations (R7) by tool and session

Session	Session Length (in seconds)	Complete Loop Iterations	Seconds per Loop Iteration
NaNofuzz			
P16 Task 2	582	5	116
P16 Task 6	528	7	75
P22 Task 2	551	5	110
P23 Task 5	588	1	588
P37 Task 3	196	1	196
Mean	489	3.8	129
Jest			
P21 Task 1	322	2	161
P35 Task 2	714	6	119
P37 Task 2	338	2	169
Mean	458	3.3	137
Hypothesis			
H01 Task 1	1,800	3	600
H01 Task 2	1,800	4	450
H02 Task 1	623	4	156
H02 Task 2	1,690	9	188
H03 Task 1	595	2	298
H03 Task 2	709	6	118
H04 Task 1	694	3	231
H04 Task 2	1,030	3	343
H05 Task 1	419	3	140
H05 Task 2	608	3	203
Mean	997	4	249

then randomly sampled[11] 10% (14/135) of the fixed 30-second intervals with multiple steps using Google RNG and re-watched the corresponding Session Recordings (D2) to independently code the interval timing detail, which agreed with the third author's timing detail.

(A4) Calculate Reliability of Coding. Of the 18 recorded sessions across both user studies, 11 Step Transcripts (fixed intervals) (R3) were coded by two raters and naturally partitioned into two groups: 8 from Jest and NaNofuzz Session Recordings (D1) and 3 from Hypothesis Session Recordings (D2). We used the 11 pairs of Step Transcripts (fixed intervals) (R3) to calculate Inter-rater Reliability (R5) for the two groups of sessions using Cohen's Kappa [93].

(A5) Extract Abstract Step Transitions. For each session and interval, we listed the transitions observed from the step in the current interval to the step in the subsequent interval within Step Transcripts (variable intervals) (R4). We then created three matrices—one matrix per tool—where the first dimension of the matrix was *currentStep*, the second dimension was *nextStep*, and the intersecting element value was the count of transitions observed from *currentStep* to *nextStep*.

(A6) Extract Loop Iterations. Due to normal backtracking and step-skipping behavior by users, it was not reasonable to assume any particular step signaled a new loop iteration, nor was it

[11] The intervals we randomly sampled and re-watched were: H01 T1 01:00–01:29, 23:00–23:29; H01 T2 06:00–06:29; H02 T2 03:30–03:59, 06:00–06:29, 06:30–06:59, 14:30–14:59, 20:30–20:59; H03 T1 00:30–00:59; H03 T2 05:00–05:29; H04 T1 02:30–02:59; H04 T2 10:30–10:59; H05 T2 07:00–07:59, 09:30–09:59.

Table 10. Time spent in each step (R8) by tool and session

Session	S2	S3	S4	S5	S6	S7
NaNofuzz						
P16 Task 2	2:00 (36%)	0:50 (15%)	1:01 (18%)	0:18 (5%)	0:25 (8%)	0:56 (17%)
P16 Task 6	2:01 (34%)	0:28 (8%)	0:37 (10%)	0:41 (11%)	1:28 (24%)	0:45 (12%)
P22 Task 2	2:00 (27%)	2:20 (31%)	1:02 (14%)	0:25 (6%)	1:08 (15%)	0:35 (8%)
P23 Task 5	9:30 (100%)	0:00 (0%)	0:00 (0%)	0:00 (0%)	0:00 (0%)	0:00 (0%)
P37 Task 3	1:00 (40%)	0:05 (3%)	0:02 (1%)	0:04 (3%)	0:15 (10%)	1:04 (43%)
Σ	**16:31 (53%)**	**3:43 (12%)**	**2:42 (9%)**	**1:28 (5%)**	**3:16 (11%)**	**3:20 (11%)**
Jest						
P21 Task 1	2:30 (50%)	0:00 (0%)	1:11 (24%)	0:39 (13%)	0:40 (13%)	0:00 (0%)
P35 Task 2	3:58 (35%)	0:00 (0%)	3:57 (35%)	1:26 (13%)	1:37 (14%)	0:16 (2%)
P37 Task 2	0:30 (13%)	0:19 (8%)	2:32 (64%)	0:06 (3%)	0:24 (10%)	0:05 (2%)
Σ	**6:58 (35%)**	**0:19 (2%)**	**7:40 (38%)**	**2:11 (11%)**	**2:41 (13%)**	**0:21 (2%)**
Hypothesis						
H01 Task 1	8:23 (28%)	0:15 (1%)	16:45 (56%)	0:30 (2%)	3:03 (10%)	1:04 (4%)
H01 Task 2	13:34 (44%)	2:23 (8%)	12:06 (39%)	0:10 (1%)	0:25 (1%)	2:22 (8%)
H02 Task 1	3:10 (37%)	0:29 (6%)	0:41 (8%)	0:10 (2%)	3:30 (41%)	0:30 (6%)
H02 Task 2	8:46 (31%)	0:07 (0%)	11:02 (39%)	1:03 (4%)	5:34 (20%)	1:28 (5%)
H03 Task 1	1:35 (18%)	0:40 (7%)	5:22 (60%)	0:43 (8%)	0:40 (7%)	0:00 (0%)
H03 Task 2	2:41 (21%)	1:40 (13%)	6:51 (53%)	0:19 (2%)	1:03 (8%)	0:26 (3%)
H04 Task 1	1:29 (13%)	0:24 (3%)	4:32 (39%)	1:03 (9%)	1:30 (13%)	2:32 (22%)
H04 Task 2	5:50 (33%)	0:16 (2%)	7:17 (42%)	1:36 (9%)	1:48 (10%)	0:43 (4%)
H05 Task 1	2:53 (44%)	0:00 (0%)	3:12 (49%)	0:09 (2%)	0:16 (4%)	0:00 (0%)
H05 Task 2	6:54 (69%)	0:00 (0%)	1:40 (17%)	0:18 (3%)	0:25 (4%)	0:43 (7%)
Σ	**55:15 (33%)**	**6:14 (4%)**	**69:28 (42%)**	**6:01 (4%)**	**18:14 (11%)**	**9:48 (6%)**

Times shown as minutes:seconds; S2=Understand expected behavior; S3=Choose scenarios to test; S4=Update test suite; S5=Collect test results; S6=Understand test results; S7=Choose interesting test results

reasonable to expect that users would perform steps in any particular order. As the Step Transcripts (variable intervals) (R4) showed evidence of skipping and backtracking among steps S1–S4 and S4–S7, we selected step S4 as a boundary such that a **Complete Loop Iteration** might be defined as executing two different steps from among S1–S4 and then S4–S7; subsequently, any execution of a step from among S1–S3 marked the start of the next loop iteration. Using this definition, we counted loop iterations for each session by first setting the loop iteration counter to 1 for the first interval of each Step Transcript (variable intervals) (R4). We tested each interval in ascending time order to identify whether the interval started a new loop iteration. At the start of each new loop iteration, we incremented the loop counter. We then calculated the mean seconds per iteration for each session by dividing the session's length in seconds by the session's ending loop counter.

(A7) Calculate Step Summaries. We summed up the total interval times coded for each step in each Step Transcript (variable intervals) (R4). We then divided the sum of each step's time by the total session time. This produced 18 Step Summaries (R8), one summary per session.

Below we report the result of the data analyses described above.

(R3) Step Transcripts (fixed intervals). We provide the transcripts as supplemental material and show an excerpt of one transcript in Table 6.

(R4) Step Transcripts (variable intervals). We provide the transcripts as supplemental material and show an excerpt of one transcript in Table 7.

(R5) Inter-rater Reliability. For the Jest and NaNofuzz sessions (D1), the raters' coding exhibited good to excellent [87] agreement for steps: S1 κ=1.000, S2 κ=0.659, S3 κ=0.557, S4 κ=0.685, S5 κ=0.680, S6 κ=0.432, and questionable agreement for step S7 κ=0.254. For the Hypothesis sessions (D2), the raters' coding exhibited good to excellent [87] agreement for all steps: S1 κ=1.000, S2 κ=0.725, S3 κ=0.406, S4 κ=0.787, S5 κ=0.958, S6 κ=0.918, S7 κ=0.672.

(R6) Abstract Step Transitions. Table 8 and Figure 4 show the step transitions observed.

(R7) Complete Loop Iterations. Table 9 shows the number of Complete Loop Iterations observed.

(R8) Step Summaries. Table 10 shows the time the user was observed performing each abstract step.

We analyze and discuss the implications of these results in Section 6.

6 DISCUSSION & RESULTS ANALYSIS

We now discuss how the results reported in Sections 4 and 5 address the two research questions introduced in Section 1 as well as the implications of these results.

6.1 What steps might a user perform to generate a test suite for a single function? (RQ1)

In Section 4, we extracted a Coding Guide (R1) and 7 Abstract Steps (R2) from Jest and NaNofuzz Session Recordings (D1) in which professional software engineers generated tests for a test suite using Jest and NaNofuzz. We showed in Section 5.2 that two raters using the Coding Guide (R1) achieved good to excellent agreement coding Hypothesis Session Recordings (D2), which is a new data set involving a different type of tool that our process model had not seen previously.

Table 8 and Figure 4 (R6) provided evidence that users in the recorded sessions generated test suites by performing the 7 abstract steps and that a user's current step might help predict the likely next step—e.g., $\geq$ 77% of transitions from S3 were to S4, $\geq$ 75% of transitions from S4 were to S5, and $\geq$ 80% of transitions from S5 were to S6.

Problem solving has been modeled as a process of repeating a set of steps until some goal is achieved [19, 49, 74, 86]. Norman [78] says that users may perform a cycle of steps as well as backtrack and skip steps when interacting with a system. Test suite generation has also been described as repeating and moving among a set of steps [9, 10, 16, 84]. Table 8 and Figure 4 (R6) provided empirical evidence that users may execute the 7 Abstract Steps in various orders as they generated tests for a test suite; e.g., Jest users backtracked from S6 to S4 21% of the time. Table 9 (R7) provided empirical evidence that users repeated the set of steps while generating a test suite and that 89% (16/18) of the sessions described in this article exhibited multiple Complete Loop Iterations.

6.2 To what extent can the steps help describe user test suite generation sessions? (RQ2)

In Section 5, we described how we used the Coding Guide (R1) and Abstract Steps (R2) to describe recorded sessions that our process model had not seen previously. These sessions were sourced from two separate data sets (D1, D2) from two user studies across which users generated tests using three TUGs of varying types: Jest, NaNofuzz, and Hypothesis. Tables 8 to 10 (R6–R8) show that we were able to use the steps to describe these diverse user test suite generation sessions. Human-centered Software Engineering talks about "hard-to-answer questions" [63] that can be difficult for users to answer without additional support. In the remainder of this section, we discuss how TestLoop's Abstract Steps can support researchers who want to answer four hard-to-answer questions about how users *actually* interact with TUGs.

Where are users spending their time? Understanding where users *actually* spend their time while using a tool can illuminate surprising problems or bottlenecks that have important implications for researchers who want to improve tool usability [13, 55, 70, 71]. For example, Ko et al. [57, 61] reported that programmers spent a surprising 35% of their program maintenance time navigating (e.g., scrolling) among parts of the code, which inspired new IDE designs that improved support for users' *actual* navigational needs. In testing, typing up a chosen test case in a tool's required format has been called a "mostly a mechanical task" [9]. However, Table 10 shows that Jest users in our study spent the most time (38%) typing their test cases into Jest's required format (S4). Similarly, Hypothesis users spent 42% of their time on the same step (S4). While the timing data for Hypothesis sessions may be imprecise due to our use of a think-aloud protocol (see Section 5.1), we were surprised that users spent so much time trying to type up their desired test cases for both Jest and Hypothesis. Given the broad adoption of tools similar to Jest and Hypothesis across millions of public GitHub repositories [102], modest improvements that reduce the user effort required to encode tests cases—e.g., by allowing the user to encode correctness in various ways, and by reducing the amount of boilerplate code the user must type correctly or remember to update—might yield a substantial productivity gain across the broad community of software engineers and projects.

How does a user's effort on a particular step compare among tools? With the important caveat that our Hypothesis session timing data may be imprecise due to that study's use of a think-aloud protocol (see Section 5.1), Table 10 shows that Hypothesis, Jest, and NaNofuzz users spent 42%, 38%, and 9% of session time, respectively, encoding tests into the tool's required format (S4). The Gulf of Execution (U2) helps us consider that the varying amounts of time spent might partially reflect the varying complexity and expressive power of the tools' respective property-based, example-based, and implicit-based oracles. For instance, a property-based oracle is naturally more complex than an example-based oracle given that the user must encode a test that holds across diverse inputs and outputs. However, The Gulf of Execution (U2) also helps us consider that the varying time spent may also partially reflect the dissimilarity of the tool's available functions to those functions that the user may naturally want to perform. The think-aloud protocol of the Hypothesis sessions helped us to confirm that some users, such as H01, struggled to specify the PUT's expected behavior in the particular way that Hypothesis required. If an implicit-based oracle may be sufficient to find some bugs with less effort, then the user might prefer to first find easier bugs with an implicit-based oracle and then step up to use more complex oracles to find harder bugs.

Which steps might benefit from better tool support? Beyond the time spent by users on each step, TestLoop's gulfs can provide a further lens to identify opportunities for improved tool support. For example, The Gulf of Expectation (U1) in Understand Expected Behavior (S2) requires the user to build a mental model of expected behavior from the program information. Few tools presently provide support that might help narrow this gulf,[12] and tools that help users extract, annotate, and organize PUT information may be an opportunity for future research.

TestLoop may help to identify steps in which users struggle to make forward progress. For instance, Table 8 and Figure 4 show that Hypothesis users in our study followed a path similar to Jest and NaNofuzz users for steps S3–S6. However, Hypothesis users backtracked from S6 (Understand test results) to S4 (Update test suite) 60% of the time vs. 6% for NaNofuzz and 21% for Jest. The elevated rate of backtracking observed for Hypothesis may indicate that Hypothesis users struggled to cross the Gulf of Execution (U2) and had difficulty successfully encoding their desired test scenarios into Hypothesis' required format and, therefore, required repeating S4–S6—without

[12] A notable exception is proptest.ai [110], which attempts to extract property tests from a PUT's documentation.

planning a new testing scenario (S1–S3). In addition to reducing efficiency, backtracking to correct such errors may reduce a user's satisfaction as well as the effectiveness of the generated test suite.

Tests that users write can also contain bugs [10, 72]: buggy tests may pass outputs that should fail or vice-versa. To cross The Gulf of Evaluation (U3) in Understand Test Results (S6), users need feedback from the tool in order to assess whether their tests had the expected outcome. For instance, we were surprised that no users identified the bug caused by an incorrect initialization[13] in task 1 of the Hypothesis study given, we thought, that the bug's presence was obvious from inspecting the program's output. However, Hypothesis does not provide feedback about the results or outputs of passing tests by default. The Gulf of Evaluation provides a lens to see that Hypothesis' lack of feedback can make it easy for users to overlook some types of critical bugs.

But to what extent does showing more information about passing tests—including outputs—impact users' efficiency? With our previous caveat about Hypothesis timing data being imprecise (see Section 5.1), TestLoop can provide data to help us address this question. The mean time users spent on each S6 step instance for each tool may be calculated by dividing the S6 step's total user time (Table 10) by the number of S6 instances for that tool (Table 8). With this calculation, we see that Hypothesis users spent a mean 14 seconds per S6 instance ($1,094\ seconds/78\ steps$), Jest users spent 10 seconds ($161\ seconds/16\ steps$), and NaNofuzz users spent 14 seconds ($196\ seconds/14\ steps$). We might also consider *how many* results each tool returned to the user to read. Hypothesis by default returned 0–1 result(s). Jest returned 1–5 result(s) in the sampled sessions. NaNofuzz by default returned 1,000 results. Hypothesis and NaNofuzz users surprisingly spent a similar amount of time on each S6 instance despite NaNofuzz providing many more test results, including that of passing tests and the passing tests' actual outputs. This analysis indicates that providing more feedback to users in S6, if coherently organized, may not necessarily degrade user efficiency and may, instead, help users be more effective at finding bugs.

Did my new idea *actually* help users or did it simply move the work to another step? Determining whether a new feature *actually* helps users is an important reason to perform a user study [26, 40]. However, improving one step might introduce unintended work or problems in other steps [13]; therefore, it can be useful to identify *which* steps are affected by a new feature. If we consider Fraser et al. [40]'s user study of the EvoSuite TUG, its authors were surprised to find that EvoSuite did not *actually* help users find more bugs despite EvoSuite's high level of automation in steps S3 (Choose scenarios to test) and S4 (Update test suite). Without a way to describe user study sessions, the EvoSuite authors relied on users' post-survey responses to posit that the negative result *might* be due to EvoSuite generating obtuse test cases that were hard to read. The EvoSuite authors' suggestion led to many subsequent studies by other researchers, who hoped to reverse EvoSuite's negative results by improving test case readability in various ways (e.g., [45, 73, 82, 94, 97]). TestLoop provides a new way to describe user interactions with TUGs and helps us posit that EvoSuite's design shifted user effort from S3 and S4 to S7 (Choose interesting test results) by requiring that the user identify incorrect test cases without providing any affordances to support this task. This suggests that rather than focusing subsequent research only on improving the readability of test cases, researchers might also focus on ways that TUGs can support users who need to identify and debug incorrect test cases.

[13]In their own study, Duran and Ntafos [33] rejected programs with initialization errors because such errors were "too easy" to detect. The PUT for Task 1 of our Hypothesis study looked correct, but no participant detected the initialization error that resulted in identify matrices that contained all 1s due to every row of the matrix pointing to the same underlying row.

6.3 Can TestLoop be used to improve an existing tool?

Our previous user study [27] showed that NaNofuzz users out-performed Jest users in terms of effectiveness, efficiency, and confidence. But we lacked a vocabulary to describe what NaNofuzz users were doing differently than Jest users. TestLoop filled this descriptive gap and helped us elicit ideas for an improved version of NaNofuzz, which we called TerzoN [31]. Below, we outline three ideas inspired by TestLoop that we incorporated into TerzoN.

Idea 1: Allow the user to assert correctness in various ways. Users have difficulty writing property-based tests [41, 42, 114], and we can see evidence of user' varying difficulty crossing The Gulf of Execution (U2) in the TestLoop data by dividing the total S4 (Update test suite) time in Table 10 by the number of S4 steps in Table 8 for each tool to calculate mean time per S4 step. Thus, NaNofuzz is 8.1 seconds/step (164 seconds/20 steps), Jest is 28.8 seconds/step (460 seconds/16 steps), and Hypothesis is 40.1 seconds/step (4,168 seconds/104 steps).[14] We can see that users worked more quickly with the low-effort implicit-based oracle (NaNofuzz), needed more time with the example-based oracle (Jest), and expended the most time with the property-based oracle (Hypothesis) such that a user's invested time might be expected to increase as the oracle's expressive power increased. However, many bugs may be found with an implicit-based oracle, and we know that finding every bug does not require writing a high-effort property-based test. Unlike other tools that expect the user to assert correctness using one particular type of oracle, TerzoN allows the user to gradually escalate from a low-effort implicit-based oracle to an example-based oracle and then generalize those examples using a property-based oracle, if necessary. By allowing the user the flexibility of asserting correctness in various ways, we expected that users might avoid unnecessary effort in cases where a simpler oracle might sufficiently achieve the user's goals.

Idea 2: Minimize typing and boilerplate code. In addition to expressive power, the design of the tool may also influence step times. We can see in Table 10 that Jest and Hypothesis users spent the largest portion of their session time performing S4 (Update test suite), which for those tools involves a lot of typing and boilerplate code. Given that we intended for TerzoN to support property-based oracles, we knew that users would need to write *some* code, but we wanted to minimize the amount of code that the user needed to type. Consequently, TerzoN included three features intended to reduce the need for typing. First, TerzoN generates one set of inputs for all property tests so that the user may avoid typing up and maintaining a separate input generator for every property test. Second, TerzoN's user interface includes a prominent "+" button that generates any boilerplate code required for a new property test so that the user only needs to type up the unique aspects of the test. Third, TerzoN provides the output of the PUT to the property test as a parameter so that the user does not have to also type up (or copy/paste) a call to the PUT in every property test.

Idea 3: Display and organize all property-based test results. Analysis via TestLoop's Gulf of Evaluation (U2) indicated that users' difficulty with task 1 in the Hypothesis user study stemmed from inadequate feedback during S6 (Understand test results). We posited that displaying the Program Under Test's (PUT) incorrect output in Hypothesis might have enabled users to detect the bug. Therefore, TerzoN differs from the standard behavior of many property-based testing tools in how it presents test results in two significant aspects. First, TerzoN organizes and presents the details of all tests—whether they passed or failed—and crucially, shows the PUT's output for each test. This is unlike many property-based testing tools that, by default, do not show the results of successful tests. Second, TerzoN continues executing tests even after encountering the first failure and reports all results, thus offering users a comprehensive view with multiple execution examples. In contrast, most property-based testing tools stop at the first failure and only display that single failing case.

[14] Our Hypothesis session timing data may be imprecise due to that study's use of a think-aloud protocol (see Section 5.1).

We think this approach was successful: in our recent randomized controlled trial of professional software engineers, participants using TerzoN outperformed participants using the popular industry property-based testing tool for TypeScript, called fast-check [32], in terms of effectiveness (by a factor of 2) and efficiency (by 16%).[15] We describe more details about TerzoN and its evaluation in our recent conference paper [31]. However, without a way to describe the steps that users actually performed—and the gulfs they actually crossed—to generate test suites, we would have built a property-based testing tool that was more traditional and very different than TerzoN. Instead, our TestLoop research played an important role in our design discussions because it allowed our design decisions to be based on what users were *actually* doing rather than on our own subjective guesses.

7 FUTURE WORK

The Gulf of Expectation (U1) may apply to domains *other* than test suite generation. Determining what to expect may be relevant when using an application programming interface (API), designing a traditional or LLM-based system, interacting with an unfamiliar device, planning a complex course of action, filling out government forms, and so on. Users often do not have a perfect understanding of the outcome they expect to achieve, what outcomes might be normative, or how they might evaluate whether an outcome is "correct" or "optimal." In this article, we identified and described this new gulf specifically within the context of test suite generation; however, more research is needed that focuses on the ways in which this new gulf might be useful in other domains.

A surprising test result might lead a user to, e.g., debug or repair a program [9, 10, 16]. Our study did not attempt to observe processes *related* or *adjacent* to test suite generation (see Section 3). Further studies would be needed to understand how these processes might interact so that tools can be designed to support users as they transition among various steps of related processes.

Using TestLoop's abstract steps to describe how users interact with testing tools, oracle types, or user testing processes that our study did not observe may identify contexts in which TestLoop may not be well-suited. More studies would be needed to determine the boundaries of TestLoop's applicability as well as the ways in which TestLoop might be refined to expand its applicability. Our study did not aim to identify what various "units of progress" might be expected for each Complete Loop Iteration, and more studies would be needed to understand the relation among the user's progress toward a goal, a TUG's design, and the number of Complete Loop Iterations.

In this article, we aimed to describe the steps of a single user generating a test suite to function test a single function. However, industrial-scale software testing may involve many users generating a test suite involving many functions. Consequently, further studies would be needed to evaluate and, where appropriate, extend TestLoop to *also* describe these other contexts.

8 THREATS TO VALIDITY

We do not know how our sample of software engineer participants relates to those of the general population. However, the purpose of our study was not to generalize to a population or to capture all possible user processes but rather to describe the steps a user might perform to generate a test suite for a single function. We are also unable to describe how the testing process of the small number of participants sampled in Section 4 relates to that of the general population.

To reduce barriers to participation in our study by professional software engineers, we limited the maximum time for each task. It is possible that longer session lengths might have resulted in additional or different steps. It is possible that the tasks selected from GitHub, Rosetta Code, and Stack Overflow were not representative. However, our study followed Ko et al.'s [58] recommendation to use "found" tasks to improve task realism. In addition, 86% (24/28) of NaNoFuzz study

[15] Confidence was numerically higher among participants using TerzoN, but this measure was not statistically significant.

participants and 100% (5/5) Hypothesis study participants reported that the study tasks were similar to tasks they might encounter while programming outside the study.

The Hypothesis study used a think-aloud protocol that had the important benefit of providing more detail about what the participants were thinking, which helped us to confirm some of our findings. The trade-off of using a think-aloud protocol is that the timings of the Hypothesis Step Summaries (R8) might not be reliable in ways that are not possible to quantify. However, the important finding in this study is that we could use the abstract steps to describe Hypothesis sessions.

Our thematic analysis resulted in a series of steps intended to answer our particular research questions, and other interpretations are possible. Braun and Clarke explain that it is not possible to entirely remove the researcher's biases from the inductive thematic analysis process [20]. Consequently, our discussion of prior work in Section 3 and the context of our research described in Section 1 are intended to help the reader understand the background and perspectives of the authors, who conducted the inductive thematic analysis described in Section 4.

We used TestLoop to describe sessions in which a user generated a test suite for a single function using Jest (an example-based human-guided TUG), NaNofuzz (an implicit-based random-guided TUG), or Hypothesis (a property-based random-guided TUG). Test suite generation may occur in various *other* contexts, such as when multiple users generate a test suite for an entire system or with various types and combinations of tools that we did not evaluate. Importantly, our study has the limitation that it did not identify the particular boundaries of TestLoop's applicability. Therefore, further studies would be needed to evaluate and possibly extend TestLoop for use in other contexts and with other user testing processes, tools, and oracles that were not observed in our study.

Any process model must elide *some* details. Card et al. [21] argued that simplicity is necessary for models to be useful as have researchers in other fields such as Robinson [90] in economics. Easterbrook et al. [35] emphasizes that "real-world phenomena are simply too rich and complex to study without" filtering away many details. While it is possible that TestLoop elides important details that would have changed our results, our aim was to fill an important gap in knowledge that other researchers might evaluate in their own studies as well as build upon. We did not investigate models of processes *related* or *adjacent* to test suite generation. Therefore, we do not know if these related processes affect test suite generation in important ways that would have changed our results.

9 CONCLUSION

We presented TestLoop, a process model of a single user generating a test suite to function test a single function, which builds upon prior work, such as Norman's Seven Stage Model of Action [77, 78] and the oracle problem [15]. TestLoop systematizes a user's test suite generation process into 7 Abstract Steps and 3 Gulfs that we extracted from empirical recordings of individual professional software engineers generating test suites using one of two different testing tools. In our evaluation of TestLoop's 7 Abstract Steps and 3 Gulfs, we showed that TestLoop helped us to answer otherwise hard-to-answer questions and to describe user test suite generation activity in ways that may complement traditional qualitative analysis of user feedback, such as that used in the NaNofuzz [27] and EvoSuite [40] user studies. Within this study, we also presented the new Gulf of Expectation, which represents the user's need to determine what to expect from the system and which—similar to Norman's Gulfs of Execution and Evaluation—may apply to *other* domains beyond software testing. Taken together, TestLoop's steps and gulfs may help researchers describe, explain, and improve upon the benefits that testing tools offer to users; thus, we shared our experiences using TestLoop to improve upon an existing tool, called NaNofuzz, in order to create a new tool, called TerzoN, that evaluated favorably compared to a popular industry tool in a recent randomized controlled trial [31]. We hope that this work spurs much-needed further research into testing tool usability as well as interventions that might help users generate test suites more effectively and efficiently, along with appropriate evaluations of their success.

10 DATA AVAILABILITY

We provide as supplementary material the data and materials necessary to reproduce our results [29], with the exception of the video recordings (D1, D2), which we are unable to share due to participant privacy and IRB restrictions. The data and materials to reproduce the original NaNofuzz user study [27] are provided as supplementary material for that study [28].

11 APPENDICES

The following sub-section is adapted from Section 5.3 of Davis et al. [27].

11.1 Participant Details for the NaNofuzz User Study

We recruited professional software engineers into the user study via LinkedIn, Mastodon, Twitter, and e-mail. The recruiting posts described the study and included a link to the screener survey, which screened for participants: (i) in the United States or Canada (as required by our IRB), (ii) who were over 18, (iii) had at least one year of professional programming experience, and (iv) had programming experience with TypeScript. Participants were offered a $30 Amazon gift card. No bonuses were offered. Recruited participants were asked to also recruit others they thought might be open to participating; however, we provided no incentives for participants to do so.

From November 4, 2022 to January 6, 2023, the screener survey received 552 responses and automatically classified 99 responses as likely being eligible, of which 35 were humans that scheduled sessions, and 28 completed the study. Four consented participants were excluded from the data set: one was unable to access their GitHub account and was unable to start the study, another did not follow protocol, and two more had to leave unexpectedly without finishing the study. In the screener survey, potential participants self-reported: gender; professional software engineering experience; hours of coding per week; and experience with testing tools, TypeScript, Jest, and Visual Studio Code. The screener included timed questions recommended by Danilova [25] to eliminate non-programmers. The screener included TypeScript and Jest questions, which we used to identify bots. The final page of the screener allowed the participant to choose a time slot.

Participants were assigned to groups A and B using matched pair random assignment based on a physical coin flip and participants' self-reported professional coding experience: 1–5 years, 6–10 years, and 11+ years. When a participant scheduled a session, they were assigned a participant number, and a researcher checked to see if a previous participant with the same experience level was awaiting a match. If no participant with the same experience level was awaiting a match, the researcher flipped a physical coin to determine the participant's group, and the participant was flagged as needing a match. When the next participant with the same experience level scheduled a time slot, the new participant would be matched to the previous one such that one participant would be randomly assigned to group A and the other randomly assigned to group B.

Participant demographics are shown in Table 11 and were as follows: 5 participants identified as female, 21 as male, and 2 did not disclose; 4 participants had 10+ years of professional experience, 4 had 6–9 years, and 20 had 1–5 years; 11 participants reported spending 30+ hours coding per week, 13 reported spending 10–29 hours per week, and 4 reported 5–10 hours per week.

11.2 Participant Details for the Hypothesis User Study

We recruited participants using announcements to: Mastodon, a mailing list of Carnegie Mellon University Professional Software Engineering Masters students, and the Slack for a PhD-level software engineering course in which Hypothesis was taught. The recruiting messages to Mastodon

Table 11. NaNofuzz User Study Participants ($n = 28$). IDs were assigned at the time an appointment was made. This table does not show: P01–11, P17 (pilots); P28, P38, P39, P41 (excluded); P13, P29, P40 (cancellations). This table is adapted from the supplemental material of Davis et al. [27].

ID	Group	Professional Experience	Coding Hrs/Week	Self-Reported Expertise with: TypeScript	Jest	VS Code	Gender
P12	A	1–5 years	30–40	Advanced	Intermediate	Advanced	Male
P14	B	6–10 years	10–20	Advanced	None	Advanced	Male
P15	B	1–5 years	20–30	Intermediate	None	Beginner	Undisclosed
P16	A	6–10 years	30–40	Beginner	Beginner	Beginner	Male
P18	B	1–5 years	5–10	Beginner	Beginner	Intermediate	Female
P19	A	1–5 years	20–39	Intermediate	Beginner	Advanced	Undisclosed
P20	B	6–10 years	30–40	Advanced	None	Expert	Male
P21	A	6–10 years	5–10	Intermediate	Beginner	Intermediate	Male
P22	A	1–5 years	20–30	Intermediate	Beginner	Advanced	Male
P23	B	6–10 years	10–20	Intermediate	Intermediate	Intermediate	Female
P24	B	1–5 years	30–40	Advanced	Beginner	Advanced	Male
P25	A	1–5 years	30–40	Advanced	Beginner	Advanced	Female
P26	B	11+ years	10–20	Beginner	None	Advanced	Male
P27	B	1–5 years	10–20	Intermediate	Beginner	Advanced	Male
P30	B	1–5 years	10–20	Intermediate	Beginner	Advanced	Male
P31	B	1–5 years	20–30	Intermediate	Beginner	Expert	Male
P32	B	1–5 years	20–30	Intermediate	None	Expert	Female
P33	A	6–10 years	30–40	Advanced	Advanced	Expert	Male
P34	A	1–5 years	40+	Intermediate	Intermediate	Advanced	Male
P35	B	1–5 years	30–40	Beginner	None	Advanced	Male
P36	A	1–5 years	10–20	Advanced	Intermediate	Advanced	Male
P37	B	1–5 years	30–40	Intermediate	Beginner	Advanced	Male
P42	B	1–5 years	10–20	Expert	Advanced	Expert	Male
P43	A	1–5 years	30–40	Beginner	None	Advanced	Female
P44	A	11+ years	5–10	Beginner	None	Advanced	Male
P45	A	1–5 years	5–10	Advanced	Advanced	Advanced	Male
P46	A	1–5 years	10–20	Intermediate	Beginner	Advanced	Male
P47	A	1–5 years	40+	Beginner	Beginner	Intermediate	Male

Table 12. Hypothesis User Study: Participants ($n = 5$).

ID	Professional Experience	Coding Hrs/Week	Self-Reported Expertise with: Python	Property Testing	VS Code	Gender
H01	0 years	5–10	Beginner	Beginner	Intermediate	Female
H02	1–5 years	20–30	Intermediate	None	Intermediate	Male
H03	1–5 years	40+	Advanced	None	Expert	Male
H04	1–5 years	20–30	Intermediate	None	Intermediate	Male
H05	1–5 years	10–20	Beginner	None	Advanced	Female

and the mailing list included a link to a screener survey, which screened for participants who were: (i) in the United States or Canada (as required by our IRB), (ii) over 18 years old, and (iii) had programming experience with Python. The recruiting message to the course Slack included a direct scheduling link, and we verified participant eligibility and collected demographics within the study session. Qualified participants were offered a $30 Amazon gift card, and no bonuses were offered.

From May 8, 2024 to May 19, 2024, the screener survey received 6 responses and automatically classified all 6 responses as likely being eligible, and all 6 were humans that scheduled sessions. 4 of these participants attended the sessions they scheduled and completed the study. We followed up with the 2 potential participants who did not attend the sessions they scheduled and received no response. One additional participant scheduled a session in response to the announcement on the PhD-level course Slack. All consented participants completed the study, and none were excluded from the data set. In the screener survey, potential participants self-reported: gender; professional software engineering experience; hours of coding per week; and experience with Python, property-based testing tools like Hypothesis, and Visual Studio Code. The screener included timed questions recommended by Danilova [25] to eliminate non-programmers as well as property-based testing questions, which we included to help us identify bots. The final page of the screener included a Google Calendar link that allowed the participant to schedule a time slot.

Participant demographics are shown in Table 12 and were as follows: 2 participants identified as female and 3 as male; 4 participants had 1–5 years of professional programming experience, 1 reported 0 years; 3 participants reported spending 30+ hours coding per week, 1 reported spending 10–29 hours per week, and 1 reported 5–10 hours per week.

ACKNOWLEDGMENTS

This work was supported in part by a CyLab seed funding award and by NSF grants 1910264, 2150217, and 2339775. Any opinions, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

REFERENCES

[1] 2016. *The American Heritage Dictionary of the English Language* (sixth [revised] ed.). Houghton Mifflin Harcourt, Boston, MA.

[2] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. 2020. Software documentation: the practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 590–601. https://doi.org/10.1145/3377811.3380405

[3] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1199–1210. https://doi.org/10.1109/ICSE.2019.00122 ISSN: 1558-1225.

[4] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05)*. Association for Computing Machinery, New York, NY, USA, 98–109. https://doi.org/10.1145/1040305.1040314

[5] Bilal Amir and Paul Ralph. 2018. There is no random sampling in software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: companion proceeedings*. 344–345.

[6] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing*. Cambridge University Press.

[7] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. *SIGPLAN Not.* 37, 1 (Jan. 2002), 4–16. https://doi.org/10.1145/565816.503275

[8] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001. https://doi.org/10.1016/j.jss.2013.02.061

[9] Mauricio Aniche. 2022. *Effective Software Testing*. Manning Publications. https://learning.oreilly.com/library/view/effective-software-testing/9781633439931/

[10] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2022. How Developers Engineer Test Cases: An Observational Study. *IEEE Transactions on Software Engineering* 48, 12 (Dec. 2022), 4925–4946. https://doi.org/10.1109/TSE.2021.3129889 Conference Name: IEEE Transactions on Software Engineering.

[11] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* 23, 4 (2018), 1959–1981.

[12] Leif Azzopardi, Diane Kelly, and Kathy Brennan. 2013. How query cost affects search behavior. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval (SIGIR '13)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/2484028.2484049

[13] Gregg "Skip" Bailey. 1993. Iterative methodology and designer training in human-computer interface design. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*. Association for Computing Machinery, New York, NY, USA, 198–205. https://doi.org/10.1145/169059.169163

[14] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering* 27, 4 (April 2022), 94. https://doi.org/10.1007/s10664-021-10072-8

[15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[16] Boris Beizer. 1983. *Software testing techniques.* Van Nostrand Reinhold Co., USA.

[17] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering* 45, 3 (2017), 261–284.

[18] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. https://doi.org/10.1145/3477132.3483540

[19] John Bransford and Barry S. Stein. 1984. *The ideal problem solver : a guide for improving thinking, learning, and creativity.* New York, NY : W.H. Freeman. http://archive.org/details/idealproblemsolv00bran

[20] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[21] Stuart K. Card, Thomas P. Moran, and Allen Newell. 1983. *The Psychology of Human-Computer Interaction.* CRC Press, Boca Raton. https://doi.org/10.1201/9780203736166

[22] Stephen Cass. 2023. The Top Programming Languages 2023 - IEEE Spectrum. https://spectrum.ieee.org/the-top-programming-languages-2023

[23] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[24] Sasha Costanza-Chock. 2020. *Design Justice: Community-Led Practices to Build the Worlds We Need.* The MIT Press. https://library.oapen.org/handle/20.500.12657/43542 Accepted: 2020-12-15T13:38:22Z.

[25] Anastasia Danilova. 2022. *How to Conduct Security Studies with Software Developers.* Ph. D. Dissertation. Universitäts- und Landesbibliothek Bonn.

[26] Matthew C. Davis, Emad Aghayi, Thomas D. Latoza, Xiaoyin Wang, Brad A. Myers, and Joshua Sunshine. 2023. What's (Not) Working in Programmer User Studies? *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 120 (jul 2023), 32 pages. https://doi.org/10.1145/3587157

[27] Matthew C. Davis, Sangheon Choi, Sam Estep, Brad A. Myers, and Joshua Sunshine. 2023. NaNofuzz: A Usable Tool for Automatic Test Generation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1114–1126. https://doi.org/10.1145/3611643.3616327

[28] Matthew C. Davis, Sangheon Choi, Sam Estep, Brad A. Myers, and Joshua Sunshine. 2023. Reproduction package for article "NaNofuzz: A Usable Tool for Automatic Test Generation". https://doi.org/10.1145/3580413

[29] Matthew C. Davis, Sangheon Choi, Amy Wei, Sam Estep, Brad A. Myers, and Joshua Sunshine. 2024. Reproduction package for article "TestLoop: A Process Model Describing Human-in-the-Loop Software Test Suite Generation". https://github.com/nanofuzz/testloop-2024-analysis

[30] Matthew C. Davis, Amy Wei, Sangheon Choi, and Sam Estep. 2024. NaNofuzz: a fast and easy-to-use automatic test suite generator for Typescript that runs inside Visual Studio Code. https://github.com/nanofuzz/nanofuzz. [Online; accessed 2024-12-19].

[31] Matthew C. Davis, Amy Wei, Brad A. Myers, and Joshua Sunshine. 2024. TerzoN: Human-in-the-Loop Software Testing with a Composite Oracle. Association for Computing Machinery. https://cmumatt.github.io/assets/TerzoN_2025_preprint.pdf Pre-print; in submission.

[32] Nicolas Dubien. 2024. fast-check official documentation | fast-check. https://fast-check.dev/

[33] Joe W. Duran and Simeon Ntafos. 1981. A report on random testing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)*. IEEE Press, San Diego, California, USA, 179–183.

[34] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive Load Theory in Computing Education Research: A Review. *ACM Transactions on Computing Education (TOCE)* 22, 4 (2022), 1–27.

[35] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer, London, 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

[36] Eduard Enoiu and Robert Feldt. 2021. Towards Human-Like Automated Test Generation: Perspectives from Cognition and Problem Solving. In *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 123–124. https://doi.org/10.1109/CHASE52884.2021.00026 ISSN: 2574-1837.

[37] Eduard Enoiu, Gerald Tukseferi, and Robert Feldt. 2020. Towards a Model of Testers' Cognitive Processes: Software Testing as a Problem Solving Approach. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 272–279. https://doi.org/10.1109/QRS-C51114.2020.00053

[38] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.

[39] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (Feb. 2013), 276–291. https://doi.org/10.1109/TSE.2012.14 Conference Name: IEEE Transactions on Software Engineering.

[40] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (2015), 1–49.

[41] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639581

[42] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. 2022. Some Problems with Properties. In *Proc. Workshop on the Human Aspects of Types and Reasoning Assistants (HATRA)*.

[43] Google. 2023. random number generator. https://www.google.com/search?q=random+number+generator [Online; accessed 2023-06-27].

[44] John D. Gould. 1975. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies* 7, 2 (March 1975), 151–182. https://doi.org/10.1016/S0020-7373(75)80005-8

[45] Giovanni Grano, Simone Scalabrino, Harald C Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 348–3483.

[46] David P. Hale and Dwight A. Haworth. 1991. Towards a model of programmers' cognitive processes in software maintenance: A structural learning theory approach for debugging. *Journal of Software Maintenance: Research and Practice* 3, 2 (1991), 85–106. https://doi.org/10.1002/smr.4360030204

[47] Joanne E Hale, Shane Sharpe, and David P Hale. 1999. An evaluation of the cognitive processes of programmers engaged in software debugging. *Journal of Software Maintenance: Research and Practice* 11, 2 (1999), 73–91.

[48] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. 2019. Reducing Oracle Cost in Search Based Test Data Generation. (2019). http://web4.cs.ucl.ac.uk/staff/S.Yoo/papers/Harman2010ys.pdf Retrieved 2023-07-26.

[49] John R. Hayes. 1989. Cognitive Processes in Creativity. In *Handbook of Creativity*, John A. Glover, Royce R. Ronning, and Cecil R. Reynolds (Eds.). Springer US, Boston, MA, 135–145. https://doi.org/10.1007/978-1-4757-5356-1_7

[50] Amber Horvath. 2024. *Meta-Information to Support Sensemaking by Developers.* thesis. Carnegie Mellon University. https://doi.org/10.1184/R1/26880229.v1

[51] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A. Myers. 2022. Understanding How Programmers Can Use Annotations on Documentation. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3491102.3502095

[52] Amber Horvath, Andrew Macvean, and Brad A. Myers. 2024. Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3613904.3642676

[53] Amber Horvath, Brad A. Myers, Andrew Macvean, and Imtiaz Rahman. 2022. Using Annotations for Sensemaking About Code. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3526113.3545667

[54] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct manipulation interfaces. *Human–computer interaction* 1, 4 (1985), 311–338.

[55] ISO. 2018. Ergonomics of human-system interaction—Part 11: Usability: Definitions and concepts ISO 9241–11: 2018 (en).

[56] Barbara A. Kitchenham, Guilherme H. Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F. Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. 1999. Towards an ontology of software maintenance. *Journal of Software Maintenance: Research and Practice* 11, 6 (1999), 365–389. https://doi.org/10.1002/(SICI)1096-908X(199911/12)11:6<365::AID-SMR200>3.0.CO;2-W

[57] Amy J. Ko, Htet Aung, and Brad A. Myers. 2005. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th international conference on Software engineering (ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 126–135. https://doi.org/10.1145/1062455.1062492

[58] Amy J. Ko, Thomas D LaToza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.

[59] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.

[60] Amy J. Ko and Brad A. Myers. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1569–1578.

[61] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[62] Jeshua S Kracht, Jacob Z Petrovic, and Kristen R Walcott-Justice. 2014. Empirically evaluating the quality of automatically generated and manually written test suites. In *2014 14th International Conference on Quality Software*. IEEE, 256–265.

[63] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU '10)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/1937117.1937125

[64] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering (ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 492–501. https://doi.org/10.1145/1134285.1134355

[65] Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and Software* 7, 4 (Dec. 1987), 325–339. https://doi.org/10.1016/0164-1212(87)90032-X

[66] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers.. In *USENIX Security Symposium*. 2777–2794.

[67] Henry Lieberman. 1997. The debugging scandal and what to do about it. *Commun. ACM* 40, 4 (April 1997), 26–30.

[68] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Transactions on Software Engineering and Methodology* 23, 4 (Sept. 2014), 31:1–31:37. https://doi.org/10.1145/2622669

[69] David R. MacIver, Zac Hatfield-Dodds, and many other contributors. 2019. Hypothesis: A new approach to property-based testing. https://doi.org/10.21105/joss.01891 original-date: 2013-03-10T13:51:19Z.

[70] Brad A. Myers. 1993. *Why are human-computer interfaces difficult to design and implement?* Carnegie-Mellon University. Department of Computer Science.

[71] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2019. Human-Centered Methods to Boost Productivity. In *Rethinking Productivity in Software Engineering*, Caitlin Sadowski and Thomas Zimmermann (Eds.). Apress, Berkeley, CA, 147–157. https://doi.org/10.1007/978-1-4842-4221-6_13

[72] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing*. John Wiley & Sons. Google-Books-ID: GjyEFPkMCwcC.

[73] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P Robillard. 2021. Generating unit tests for documentation. *IEEE Transactions on Software Engineering* (2021).

[74] Allen Newell. 1972. *Human problem solving*. Englewood Cliffs, N.J.: Prentice-Hall.

[75] Sebastian P Ng, Tafline Murnane, Karl Reed, D Grant, and Tsong Yueh Chen. 2004. A preliminary survey on software testing practices in Australia. In *2004 Australian Software Engineering Conference. Proceedings*. IEEE, 116–125.

[76] Jakob Nielsen. 1994. *Usability Engineering*. Elsevier Science & Technology, Chantilly, UNITED STATES. http://ebookcentral.proquest.com/lib/cm/detail.action?docID=1190977

[77] Don Norman. 2013. *The Design of Everyday Things : Revised and Expanded Edition*. Vol. Revised and expanded edition. Basic Books, New York.

[78] Donald A Norman. 1988. *The Psychology of Everyday Things*. Basic books.

[79] T. J. Ostrand and M. J. Balcer. 1988. The category-partition method for specifying and generating fuctional tests. *Commun. ACM* 31, 6 (June 1988), 676–686. https://doi.org/10.1145/62959.62964

[80] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[81] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[82] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. 2016. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th international conference on software engineering*. 547–558.

[83] David L. Parnas and Sergiy A. Vilkomir. 2007. Precise Documentation of Critical Software. In *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*. 237–244. https://doi.org/10.1109/HASE.2007.70 ISSN: 1530-2059.

[84] Mauro Pezzè and Michal Young. 2008. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons.

[85] Marllos Paiva Prado and Auri Marcelo Rizzo Vincenzi. 2018. Towards cognitive support for unit testing: A qualitative study with practitioners. *Journal of Systems and Software* 141 (2018), 66–84. https://doi.org/10.1016/j.jss.2018.03.052

[86] Jean E. Pretz, Adam J. Naples, and Robert J. Sternberg. 2003. Recognizing, defining, and representing problems. In *The psychology of problem solving*. Cambridge University Press, New York, NY, US, 3–30. https://doi.org/10.1017/CBO9780511615771.002

[87] Darrel A. Regier, William E. Narrow, Diana E. Clarke, Helena C. Kraemer, S. Janet Kuramoto, Emily A. Kuhl, and David J. Kupfer. 2013. DSM-5 Field Trials in the United States and Canada, Part II: Test-Retest Reliability of Selected Categorical Diagnoses. *American Journal of Psychiatry* 170, 1 (Jan. 2013), 59–70. https://doi.org/10.1176/appi.ajp.2012.12070999 Publisher: American Psychiatric Publishing.

[88] Martin P. Robillard. 2021. Turnover-induced knowledge loss in practice. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1292–1302. https://doi.org/10.1145/3468264.3473923

[89] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (Dec. 2011), 703–732. https://doi.org/10.1007/s10664-010-9150-8

[90] Joan Robinson. 1969. *The Economics of Imperfect Competition*. Palgrave Macmillan UK, London. https://doi.org/10.1007/978-1-349-15320-6

[91] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *2012 34th International Conference on Software Engineering (ICSE)*. 255–265. https://doi.org/10.1109/ICSE.2012.6227188 ISSN: 1558-1225.

[92] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2015. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 international symposium on software testing and analysis*. 338–349.

[93] Robert Rosenthal and Ralph L Rosnow. 2008. *Essentials of behavioral research: Methods and data analysis*.

[94] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 287–298.

[95] Ian Ruthven. 2008. Interactive information retrieval. *Annual Review of Information Science and Technology* 42, 1 (2008), 43–91. https://doi.org/10.1002/aris.2008.1440420109

[96] Joseph M. Scandura. 1977. *Problem solving: a structural/process approach with instructional implications*. Academic Press, New York.

[97] Novi Setiani, Ridi Ferdiana, and Rudy Hartanto. 2022. Understandable Automatic Generated Unit Tests using Semantic and Format Improvement. In *2022 6th International Conference on Informatics and Computational Sciences (ICICoS)*. 122–127. https://doi.org/10.1109/ICICoS56336.2022.9930600

[98] Teresa M. Shaft and Iris Vessey. 1995. Research Report—The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research* 6, 3 (Sept. 1995), 286–299. https://doi.org/10.1287/isre.6.3.286 Publisher: INFORMS.

[99] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. 2015. To Document or Not to Document? An Exploratory Study on Developers' Motivation to Document Code. In *Advanced Information Systems Engineering Workshops*, Anne Persson and Janis Stirna (Eds.). Springer International Publishing, Cham, 100–106. https://doi.org/10.1007/978-3-319-19243-7_10

[100] Janet Siegmund and Jana Schumann. 2015. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering* 20, 4 (Aug. 2015), 1159–1192. https://doi.org/10.1007/s10664-014-9318-8

[101] James Somers. 2023. What if writing tests was a joyful experience? https://blog.janestreet.com/the-joy-of-expect-tests/. [Online; accessed 2023-01-22].

[102] Facebook Open Source. 2023. Jest - Delightful Javascript Testing. https://jestjs.io/. [Online; accessed 2024-07-08].

[103] Hari Subramonyam, Roy Pea, Christopher Pondoc, Maneesh Agrawala, and Colleen Seifert. 2024. Bridging the Gulf of Envisioning: Cognitive Challenges in Prompt Based Interactions with LLMs. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/3613904.3642754

[104] E. Burton Swanson. 1976. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering (ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 492–497.

[105] The JUnit Team. 2023. JUnit. https://junit.org/ [Online; accessed 2023-06-11].

[106] Priyadarshi Tripathy and Kshirasagar Naik. 2011. *Software testing and quality assurance: theory and practice.* John Wiley & Sons.

[107] Gias Uddin and Martin P. Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (July 2015), 68–75. https://doi.org/10.1109/MS.2014.80 Conference Name: IEEE Software.

[108] Iris Vessey. 1986. Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *IEEE Transactions on Systems, Man, and Cybernetics* 16, 5 (Sept. 1986), 621–637. https://doi.org/10.1109/TSMC.1986.289308 Conference Name: IEEE Transactions on Systems, Man, and Cybernetics.

[109] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? [preprint; accessed: 2023-07-26]. http://arxiv.org/abs/2307.04346

[110] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Proptest AI. https://proptest.ai/

[111] Anneliese von Mayrhauser and A.M. Vans. 1993. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of 6th International Workshop on Computer-Aided Software Engineering*. 230–239. https://doi.org/10.1109/CASE.1993.634824 ISSN: 1066-1387.

[112] Anneliese von Mayrhauser and A.M. Vans. 1993. From program comprehension to tool requirements for an industrial environment. In *[1993] IEEE Second Workshop on Program Comprehension*. 78–86. https://doi.org/10.1109/WPC.1993.263903 ISSN: 1092-8138.

[113] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. 1997. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice* 9, 5 (1997), 299–327. https://doi.org/10.1002/(SICI)1096-908X(199709/10)9:5<299::AID-SMR157>3.0.CO;2-S

[114] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2021. Using Relational Problems to Teach Property-Based Testing. *The art science and engineering of programming* 5, 2 (2021).

[115] Andreas Zeller. 2009. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. https://learning.oreilly.com/library/view/why-programs-fail/

[116] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, Ying Zou, and Ahmed E. Hassan. 2021. An Empirical Study of Obsolete Answers on Stack Overflow. *IEEE Transactions on Software Engineering* 47, 4 (April 2021), 850–862. https://doi.org/10.1109/TSE.2019.2906315