Shaken, Not Stirred. How Developers Like Their Amplified Tests

Carolin Brandt, Ali Khatami, Mairieli Wessel, and Andy Zaidman

Abstract—Test amplification makes systematic changes to existing, manually written tests to provide tests complementary to an automated test suite. We consider developer-centric test amplification, where the developer explores, judges and edits the amplified tests before adding them to their maintained test suite. However, it is as yet unclear which kind of selection and editing steps developers take before including an amplified test into the test suite. In this paper we conduct an open source contribution study, amplifying tests of open source Java projects from GitHub. We report which deficiencies we observe in the amplified tests while manually filtering and editing them to open 39 pull requests with amplified tests. We present a detailed analysis of the maintainer's feedback regarding proposed changes, requested information, and expressed judgment. Our observations provide a basis for practitioners to take an informed decision on whether to adopt developer-centric test amplification. As several of the edits we observe are based on the developer's understanding of the amplified test, we conjecture that developer-centric test amplification should invest in supporting the developer to understand the amplified tests.

Index Terms—Software Testing, Automatic Test Generation, Developer-Centric Test Amplification

1 Introduction

Automated testing has become central to ensure a high quality during software development [1], [2], [3]. Nevertheless, writing tests is seen as a tedious and time-consuming task [4], [5], [6]. This is where automatic test generation comes in by supporting developers and relieving them of the burden of writing tests [7], [8], [9], [10], [11].

State-of-the-art test generation tools are powerful in protecting against regressions [12], finding crashes [13], and reproducing crashes [14], [15]. However, they are rather difficult to adopt in day-to-day software engineering, in part due to the difficulty to understand the generated test scenarios [16], [17]. For developers it is crucial to understand a test when it fails and they have to localize the underlying fault [18], [19].

This is where *test amplification* shows promise: instead of generating completely new tests, e.g., with genetic algorithms (e.g., EvoSuite [9]), test amplification makes systematic changes to existing, manually written tests with the intent to provide tests that are complementary to the existing test suite [20]. In contrast to generated tests that are stored separately from manually written tests, e.g., when tests are regenerated after software evolution [21], [22], our focus is on *developer-centric amplified tests*. Developer-centric test amplification is a concept we coined in our previous work [23]. It proposes that developers adopt the amplified tests into their main test suite, potentially after manually adjusting the amplified tests. Developer-centric test amplification means (1) developers benefit from only having to validate amplified tests, instead of writing these tests manually, and (2) understanding the tests should be easier because they originate from manually written tests. To illustrate this more vividly we introduce an example use case of developer-centric test amplification:

> Adriana is a software developer in a project that is struggling with automated testing, as pressure for new features makes it hard to find time to write tests. She has some time left this sprint and decides to invest it into testing. To be quicker, she uses a developer-centric test amplification tool which generates compiling and passing tests that cover code that is not covered by the test suite. Adriana browses through the proposed tests, inspecting their behavior and new coverage contribution to judge which ones to include in the test suite. Whenever she decides to keep a test, she takes a look at its code and does some adjustment to make them easier to understand for her colleagues and fit better to their project's style and quality. After adding several new tests into the test suite of her project, she commits them all and prepares a merge request that describes the improvements to the test suite.

While several studies have investigated the shortcomings of generated and amplified tests from the developer's perspective [16], [23], [24], little is known about which kind of adjustments developers would make to an amplified test before including it in the test suite. Therefore, the goal of this paper is to better understand the effort that developers need to go through when (1) deciding whether to add an amplified test to the test suite, and (2) adjusting the amplified test before it can be added. To this end, we conduct a qualitative open-source contribution study [25], [26]: We amplify tests for 52 open-source projects and open 39 pull requests to contribute the amplified tests back to the projects. For the test amplification, we employ DSpot, which is the original, arche implementation of test amplification for Java created by Danglot et al. [20], [25]. Our qualitative investigation in

- *C. Brandt, A. Khatami and A. Zaidman are with the Delft University of Technology. E-mail: c.e.brandt@tudelft.nl*
- *M. Wessel is with Radboud University.*

This research was funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032).

this paper is steered by the following research questions:

RQ1: What deficiencies do we observe in DSpot amplified tests when preparing them for a pull request?
- **RQ1.1:** On which criteria do we select a candidate test to include in the test suite?
- **RQ1.2:** Which manual edits do we perform to improve the tests before submission?

RQ2: What feedback do we receive from the maintainers on the DSpot amplified tests?
- **RQ2.1:** Which changes are proposed during the pull request discussion?
- **RQ2.2:** What kind of information is requested by the maintainers during the pull request discussion?
- **RQ2.3:** How do the maintainers justify their judgment over the amplified tests during the pull request discussion?

Based on an existing dataset of buildable Java repositories [27], we try to amplify tests for 312 open source projects. We employ the developer-centric test amplification of DSpot [23], [28], together with a new automatic post-processing module that filters and simplifies the amplified tests. For each of the 52 projects where the test amplification succeeds, we manually select a candidate test to submit in a pull request. The criteria that emerge during this selection process answer **RQ1.1**. We manually edit the candidate tests to improve their quality before opening a pull request. Based on our experiences in this phase, we build a checklist of edits to expect, the answer to **RQ1.2**. To validate whether these edits would also be proposed by open source maintainers, we omit the manual editing for half of the projects.

We open pull requests for 39 projects with one amplified test each. To clarify our contribution to the project maintainers, we provide an automatically generated textual description of the amplified test. During the discussion, we incorporate any proposed changes and answer arising questions. 19 pull requests were accepted and 13 closed. We analyze the discussions on the completed pull requests to elicit the changes that the maintainers propose (**RQ2.1**), the information they request to understand the amplified tests (**RQ2.2**), and how they justified their judgment over the amplified tests (**RQ2.3**). As we manually selected which amplified tests to submit and manually edited half of them to improve their quality before submitting, the results for the second set of research questions more closely represent what amplified test are capable of with human intervention, or with automation advancing might be capable of in the future.

2 Developer-Centric Test Amplification

The technique of *test amplification* generates new tests by modifying test that were written by developers [20]. Our work is based on the developer-centric test amplification of DSpot [23], [25], which we introduce in this section.

To explore new behavior, DSpot mutates the setup and action phase of an existing test, called the *original test*, by changing the values of literals and removing or adding method calls to the objects under test. The old assertions are removed and replaced by new assertions. For the oracle, DSpot uses the current behavior of the system: it executes the test and observes returned values, which it uses as the expected value of the new assertion. This leads to all generated tests passing. The developer-centric variant of DSpot aims at generating concise and simple tests, so it adds one setup mutation and one assertion per test it generates. Lastly, only tests that execute instructions not yet covered by the test suite are kept and shown to the developers[1].

```java
// Together with all generated tests, reaches a mutation score of 0.8518
@Test public void test09() {
 InputStream.nullInputStream();
 PipedReader pipedReader0 = new PipedReader();
 Reader reader0 = Reader.nullReader();
 ByteArrayOutputStream byteArrayOutputStream0 = new ByteArrayOutputStream(17);
 ObjectOutputStream objectOutputStream0 =
    new ObjectOutputStream(byteArrayOutputStream0);
 BufferedOutputStream bufferedOutputStream0 =
    new BufferedOutputStream(objectOutputStream0);
 MockPrintStream mockPrintStream0 =
    new MockPrintStream(bufferedOutputStream0, true);
 CopyUtils.copy(reader0, (OutputStream) mockPrintStream0);
 Reader reader1 = Reader.nullReader();
 assertNotSame(reader1, reader0);   }
```

Fig. 1: Test generated by EvoSuite for apache/commons-io.

As the next step in developer-centric test amplification, a developer browses and inspects the new, amplified tests. They judge whether a test is valuable to include into their test suite, e.g., because of the additional coverage it provides. The developer can also edit the tests where they see fit, like adding meaningful names or explanatory comments. The goal is that they include the selected and edited tests into their test suite and keep maintaining them in the future.

Developer-centric test amplification is one instance of a variety of approaches to automatically generate xUnit tests. In comparison to, e.g., the widely studied search-based test generation of EvoSuite [9], it differs in these central points:

1) EvoSuite generally works without input of manually written tests, while DSpot mutates existing, manually written tests [25]. This introduces the assumption of more readable tests from the outset.
2) EvoSuite generally aims to generate a whole test suite at once [29], while DSpot's approach is closer to test augmentation: Complementing an already existing test suite with matching additional tests [30], [31].
3) The developer-centric variant of DSpot sees the developer judging and editing a test as a central component before adding the test to a maintained test suite. That is why it should always be combined with additional information and approaches to facilitate the communication between the test generation and the developer [23].

Fig. 1 and Fig. 2 illustrate the difference of tests generated by EvoSuite and developer-centric DSpot, respectively.

Recently, Roslan et al. [32] extended EvoSuite to support test amplification in combination with EvoSuite's powerful search-based test optimization. While they reported anecdotal evidence of less readability than DSpot-generated tests, all previous developer-involving studies with EvoSuite do not consider the test amplification approach. In Section 7 we connect and contrast our findings with those of the previous user studies of EvoSuite.

1. DSpot can select tests based on mutation score, the developer-centric variant selects on added instruction coverage for its easier explainability and better performance.

```java
// Covers new instructions in ByteArrayOutputStream.reset and
      ↪ AbstractByteArrayOutputStream.resetImpl
@Test public void testToByteArrayImplAndResetImpl() {
 InputStream in = new ByteArrayInputStream(inData);
 in = new ThrowOnCloseInputStream(in);
 final ByteArrayOutputStream baout = new ByteArrayOutputStream();
 final OutputStream out =
    new ThrowOnFlushAndCloseOutputStream(baout, false, true);
 final Writer writer = new OutputStreamWriter(out, StandardCharsets.US_ASCII);
 CopyUtils.copy(in, writer);
 writer.flush();
 baout.reset();
 Assertions.assertEquals("", baout.toString());   }
```

Fig. 2: Test generated by developer-centric DSpot.

Another approach that can be related to test amplification and search-based test generation is *fuzzing*, where random, but valid inputs are generated and iteratively mutated to test the robustness of a software system [33]. While the techniques overlap in their use of mutation and aim to improve the quality of the software under test, there are significant differences that make it difficult to apply the findings of developer-centered fuzzing studies [34], [35] to our work. Fuzzing focusses on highly structured test inputs and requires the use of fuzzing harnesses to call the system under test [36]. In comparison, test amplification and search-based test generation produce ready to use test structures leveraging xUnit frameworks [37], which developer-written tests also use. Furthermore, fuzzing primarily targets robustness, aiming to uncover crashes or unintended exceptions in the software under test [33]. Because of this, fuzzing is often used to address security and reliability concerns, where any fuzzer output that leads to an undesirable crash is relevant to be addressed [38]. In comparison, developer tests like the ones produced by test amplification typically have a functional oracle or assertion that checks that the code under test behaves as expected. Therefore, the tests generated by amplification and search-based approaches improve the quality of the functional test suite, which in turn improves the confidence in the correct behavior of the code under test. Beyond that, the developer test suite can also serve as documentation [3], [39], [40] and a starting point for developers to localize the root cause of a test failure [18], [19], two use cases where the understandability of the tests is crucial.

3 Automatic Post-Processing for Developer-Centric Test Amplification

We previously conducted an exploratory study to evaluate a test amplification plugin for the IntelliJ IDE [23]. The developers we interviewed mentioned several aspects they would change before accepting the amplified tests into their test suite. For example, removing unnecessary statements or changing cryptic identifiers to meaningful ones. The participants also pointed to methods that they found not relevant to test, e.g., simple getters. To automate these already known points, we design an automatic post-processing tool for developer-centric amplified tests: the *prettifier*. The prettifier is based on an existing module in DSpot and is run after the amplification described in Section 2. The aim of the prettifier is to make the resulting tests: (1) more concise, (2) easier to read, and (3) more relevant to developers.

The participants of the previous study spent a lot of their time *understanding the behavior* of an amplified test [23]. This understanding was the basis for their judgment on whether to accept a test into their test suite. Previous studies have shown that a natural language description helps developers to understand generated tests [18], [41]. To reduce the effort required by developers to understand an amplified test, we generate natural language descriptions of the behavior and impact of the test compared to the rest of the test suite.

In this section we will present our design for the prettifier and the description generation for amplified tests.

3.1 Prettifier module

To automate several of the post-processing steps indicated by our previous study [23], we extend Danglot et al.'s prettifier module for DSpot [28]. Our approach takes three steps: (1) minimizing the tests to make them faster to read, (2) renaming variables and the test methods to make them less cryptic and more expressive, and (3) filtering and prioritizing the tests according to their relevance to the developer.

3.1.1 Minimizer

To remove statements that were part of the original test, but are not relevant for the amplified test, we adopt Oosterbroek et al.'s approach [42]. They minimize amplified tests, while retaining the provided additional coverage. The approach works in increasingly conservative steps: a) remove all statements except the assertion and the ones needed for the code to compile, b) remove all statements that do not directly interact with the assertion, i.e., by setting variables used in there, or c) remove all statements that do not (in)directly interact with the assertion, i.e., by calling a method on the object involved in the assertion. When a step decreases the coverage or causes the test to fail, the next step is tried.

We also activate two existing minimizers of DSpot. One in-lines single use variables created by the DSpot amplification, the other removes redundant casts included by the amplification for safety.

3.1.2 Test and Variable Renamer

To make the tests easier to read and understand, we implement a simple variable renamer that hides DSpot's intermediate variable names (`__DSPOT_path_696`) with less cryptic, simple names (`String2`, pattern: `<Type><N>`). Further, we generate meaningful names for the amplified tests based on the additional coverage they provide using the NATIC approach [43]. NATIC identifies in which unique methods a test covers additional instructions, compared to the other amplified tests and the existing test suite. Similar to Daka et al.'s approach [17], we rank the methods according to how much additional coverage they contain, concatenate up to two of the method names and generate a unique test name such as "`testGetFileAndHasLength`".

3.1.3 Filter and Prioritize

One issue with automatic test generation can be the large number of tests produced. Specifically with developer-centric test amplification, some generated tests target methods that developers find irrelevant to test, such as simple getters, or `hashCode`. To reduce the number of tests not relevant to developers, we included a developer-centric filter in the prettifier. It removes tests that only contribute

coverage in simple getters or setters, i.e., one line methods starting with "get" or "set". The filter also removes tests that only add coverage in Java's `hashCode` method. Because exception handling code is commonly under-tested [44], [45], we explicitly keep any test that checks for an exception. The prettifier puts the test with the most additionally covered instructions first, so that the developers inspect the most impactful amplified test first.

3.2 Descriptions for Amplified Tests

In our previous study [23], we saw that a major step for the developers was understanding the behavior and intent of an amplified test. The developers studied the code of the test, compared it to the original test and inspected the newly covered code under test. To support the understanding of amplified tests, we design an approach for an automatically generated, natural language description for amplified tests. The description surfaces the behavior and impact of the test compared to the existing test suite. It is meant to be informative for the developer without having to read the code, e.g., as a description in a pull request that proposes an amplified test.

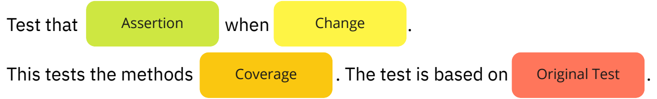

Fig. 3: The basis template for our description of amplified tests.

Similar to previous test description generators [18], [41], we use a template-based approach. It consists of four components, as presented in Fig. 3: (1) Describing the assertion, (2) describing the change to the setup of the test, (3) describing the additional coverage that is contributed, and (4) pointing to the original test. We fill these components based on information collected during the amplification process. Fig. 4 shows an example test and its corresponding description. In this case, the **assertion** is an expected exception, the **change** made by the amplification was to set the value of a literal method call parameter to an empty string. The description indicates that additional **coverage** is situated in the method `BuilderFactory.build`, and that the **original test** was `buildDouble`. The full templates and our implementation are open-source and shared as part of our replication package [46].

Test that a `java.lang.NumberFormatException` is thrown
when the parameter `data` is set to `""`.
This tests the method `BuilderFactory.build`.
This test is based on the test `buildDouble`.

```
/* Coverage improved at
redis.clients.jedis.BuilderFactory.build L. 8 +2 instr. */
@Test public void testBuild() throws Exception {
 try {
  Double build = DOUBLE.build("".getBytes());
  fail("testBuild should throw NumberFormatException"); }
 catch (NumberFormatException expected) {
  assertEquals("empty String", expected.getMessage()); }}
```

Fig. 4: An example amplified test and its generated description. The original test and the name of the changed parameter are not visible.

4 Open Source Contribution Study

The goal of this paper is to gain a clearer understanding of the changes that developers would make to amplified tests before including them into their test suite. To this end, we conduct a qualitative open source contribution study [25], [26], utilizing DSpot's developer-centric test amplification, our improved prettifier, and the automatically generated descriptions for amplified tests. The central step of the contribution study is to open pull requests with amplified tests to open source projects. However, it was crucial to us to not antagonize the project maintainers against us or the research community [47], [26]. Thus, we first carefully selected amplified tests that we believe are a valuable contribution to the project, and only opened a pull request if we found any. We document the criteria that arose during this selection process, including how often we applied each of them (**RQ1.1**). We also received feedback on the value of the submitted tests during the pull request reviews, which we analyze to answer **RQ2.3**.

As the maintainers of a software project are responsible to update the tests when the software evolves, their feedback is invaluable to understand which changes are necessary before including an amplified test in a maintained test suite. This is why analyzing the changes proposed during the pull requests is a central part of our study (**RQ2.1**). To keep the burden on the open source developers as minimal as possible, we manually edited and improved the amplified tests for half of the projects before submitting the pull requests. The other half we submitted without editing, to validate whether the edits we choose would also be proposed by a maintainer. To lead our editing, we created and continuously updated a checklist of potential edits, which we use to answer **RQ1.2**.

Another ambition of our study is to evaluate whether the automatically generated textual descriptions are helpful for understanding the behavior and value of amplified tests. Therefore, for a third of the projects we submitted the pull request with the generated description. For another third, we submitted the description and a question on whether the explanation was helpful, and for a third of the projects we submitted the pull request without any explanation of the amplified test. When analyzing the pull request discussions, we study what kind of information the maintainers requested, and the connection to whether an explanation was provided initially (**RQ2.2**).

Our qualitative study consists of five steps: First, (1) we select candidate projects for our study. Next, (2) we use the developer-centric test amplification of DSpot and our prettifier to generate the amplified tests and their descriptions. Then, (3) we manually select and improve the amplified tests, documenting our emerging criteria. After this, (4) we open pull requests with the amplified tests. Finally, (5) we analyze the feedback from the project maintainers during the pull request discussions. In the following, we will detail the separate steps of our study.

4.1 Repository Selection

Our first step is to find GitHub projects that are suitable for applying DSpot's test amplification. As our approach

requires building Java projects, and selecting coverage-improving tests with the JaCoCo[2] tool, we use Khatami and Zaidman's dataset [27], [48]. They tried to automatically build and calculate the code coverage of 1454 popular Java GitHub projects. We consider the 312 projects for which JaCoCo could successfully measure code coverage, and select one module per project[3].

4.2 Running the Test Amplification

We run DSpot on all selected project modules with a budget of 30 minutes on a desktop PC. For the exact configuration of DSpot and the hardware specification, we refer to our replication package [46]. We collect all test classes generated by DSpot. We also kept partial results, so if the amplification of all test classes would take longer than 30 minutes we consider all test classes that were completed within 30 minutes. Next, we apply the prettifier to simplify and filter the amplified tests and generate matching descriptions.

4.3 Manual Selection and Editing

We analyzed all amplified tests and created two checklists:

- How we select the best test to submit to the project.
- Which aspects we manually edit to improve the tests before proposing them to a project.

The first two authors reviewed all, the other authors a subset of the tests. Then we met up to come to a negotiated agreement [49] on the points for both checklists. During the selection and editing process of the study, performed by the first author, new points emerged. We validated them through discussions with other authors to mitigate bias and increase the reliability of the checklists [49].

For each project we selected one test to contribute in a pull request: a test we found the most valuable for the project, or a test where we were curious about the maintainer's reaction. For one half of the projects we manually edited the tests with the help of our checklist and own software engineering experience. To validate if such edits are necessary, and understand which edits are important to developers, we left the tests for the other half of the projects unedited. One goal of this study is to contribute to the open source community while learning from their feedback. It was crucial to us to only ask for the community's reviewing effort if we think a test is valuable for the project. If we did not find a test that seemed valuable, we excluded the project from the rest of the study.

4.4 Contributing Back the Tests

We opened pull requests for the resulting tests. The pull request description mentions that we want to add a test and the generated description. As Fig. 5 shows, we modified each mention of a method in the "Coverage" and "Original Test" parts to be a clickable link to the corresponding code on GitHub. The description contains a note that this pull request was part of a research study. However, we did not reveal that the tests were partially automatically generated.

2. https://www.jacoco.org/jacoco/index.html, visited August 2022.
3. Alphabetically the first. In trials we saw that the amplification not succeeding in one module of a project often means the same for other modules.

Fig. 5: An example pull request description from P14-PDM.

This is because we wanted to avoid negative backlash based on biases against automatic test generation. Before opening the pull request, we studied the contribution guidelines of the project and followed them, e.g., validating that a linter passes, or applying an auto-formatter. After opening the pull requests, we answered all questions by the maintainers and incorporated any changes they requested.

4.5 Data Analysis

We performed open and axial coding procedures [50] on the pull request discussions completed as of 19-02-2023. The first author analyzed the discussions by inductively applying open coding, wherein they identified discussion points on code changes, requests for information, judgment statements over the tests, and other possibly relevant characteristics of the pull request. They then performed an initial analysis to group the open codes, employing constant comparison [51] to the pull request discussions to validate our interpretation. To increase the reliability of the results and mitigate bias, the first and second authors refined the code set by merging codes together, updating code names, and identifying a different granularity level for a code. The authors discussed the emergent codes together with the original data and modified the codes until they reached a negotiated agreement [49]. The outcome was a set of higher-level categories as cataloged in our codebook [46]. The resulting higher-level categories are structuring the answers to our research questions in the following section, marked in **bold**. The lower-level codes captured the details that we use to illustrate the presented categories by giving concrete examples from the pull request discussions.

5 Results

In this section we discuss the results of our study: the test amplification, the manual preparation for the pull request, and our analysis of the discussions with the maintainers. To clarify in which projects each observation occurred, we use shorthand references in the style `P[n]-(E|P)(D|N)(M|C|O|D)`. The number uniquely identifies each project in our study, while the last three characters give a concise overview on the central dependent variables for the pull requests: (1) was the test **E**dited or **P**lain from the amplification tool, (2) did we provide the generated **D**escription or **N**ot, (3) the outcome of the pull request: **M**erged, **C**losed, n**O** reaction yet, under **D**iscussion. Projects where we did not select any test to contribute are

indicated as `P[n]-N--`. Table 1 gives an overview of all open source projects in our study, including the number of our pull request. We also report the project's size, total number of commits, number of contributors, number of pull requests, and the year the repository was created, showing that our study includes a diverse set of projects.

5.1 Running the Test Amplification

The base dataset [27] identified 312 repositories with in total 1821 Java modules that JaCoCo can automatically calculate coverage for. After selecting one module per repository, we in total tried to generate amplified tests for 312 modules. From the DSpot amplification, we obtained 238 classes with generated tests for 62 projects. For the other projects, DSpot crashed during the execution or could not produce any tests that improve the instruction coverage within the budget of 30 minutes. To these tests we apply the prettifier, resulting in 190 classes with 1297 generated tests for 52 projects. For the gap of 10 projects, all amplified tests were filtered out according to the criteria we explained in Section 3.1.3. Many projects only have a few tests generated (less than 5 generated test in 25 out of 52 projects), with a few large outliers (P51-PDC: 618 tests, P50-PDM: 123, P10-EDM: 96).

5.2 RQ1.1: On which criteria do we select a candidate test to include in the test suite?

For each project that we generated amplified tests for, we explored the new tests to choose a candidate test for the pull request. Initial exploration showed that there was a considerable number of unsuitable tests that could not be submitted for a variety of reasons. To transparently show the effort required to select the amplified tests in our study, we document our process of identifying the candidate test extensively. Through this process arose two checklists: one with *negative selection criteria* and one with *positive selection criteria*. With the negative criteria we identify tests that are not worth continuing with, e.g., because they would take so much effort to improve, that writing a new test from scratch felt easier. As we only submit one test per project, we used the positive criteria to pick which test of multiple possible candidates to choose for this study.

5.2.1 Negative Selection Criteria

We excluded amplified tests for the following reasons:
Coverage False Positive (P22-N–, P29-N–, P32-N–, P13-N–, P34-ENO, P43-N–): Appearing in six projects, the most-prevalent criterion to reject a test was a *coverage false positive*, i.e., tests where inspection revealed no additional coverage over existing tests. For example, the method calls leading to the additional coverage were in code taken over from the original test, that was not influenced by the amplified change (P22-N–, P29-N–, P32-N–). In three other cases, we browsed through the existing tests for the same object and found tests that are already calling the instructions the amplified test claims to newly cover (P13-N–, P34-ENO, P43-N–). We found that in three false positive cases mocking was used (P13-N–, P29-N–, P32-N–), pointing to missing support for mocks in DSpot's coverage calculation.
Simple Getters and Setters with Non-Standard Names (P11-N–): Tests only contribute coverage in simple getters and setters with non-standard names (not starting with 'get' or 'set'), which should have been filtered by the prettifier.
Could Not Find Class (P37-N–, P40-N–): We could not find the test class and the class under test (P37-N–), or the class with additional coverage (P40-N–).
Test Did Not Pass (P2-EDC): A test did not pass because the expected exception was not thrown. This and the last issue could be caused by the time difference between the commit at which we amplified the tests and the commit on which our pull request was based.
Assertion Unrelated to New Coverage (P20-N–, P22-N–, P29-N–, P32-N–)**:** In four projects, we found tests where the generated assertion does not check the behavior of the newly covered code. For example, the assertion is generated at a location before the call to the newly covered code (P20-N–), or the checked value is not influenced by the newly covered code (P23-N–, P42-PDM, P48-EDM). In both cases, while the test *covers* the code, we cannot claim that it *tests* the code.
No Explicitly Thrown Exception (P17-EDC, P19-N–, P23-N–, P24-PNC): In four projects, we found tests for `RuntimeException`s implicitly caused, e.g., in an unprotected call on a parameter that was set to null during amplification. As these exceptions did not seem to be part of the developer-intended behavior, we excluded these tests.
Change Unrelated to Assertion or New Coverage (P6-N–): We excluded tests where the amplified change did not influence the asserted value nor the additional coverage. The amplification process should check whether the amplified change is necessary for the additional coverage an amplified test is providing.
Readability and Understandability (P6-N–, P23-N–, P25-ENM, P38-ENC): A further negative selection criterion we used in four projects was that tests were not good to understand or not readable, because parts of them were cryptic, long, or verbose. For example, in P23-N– the original tests already contained complex configuration of mock behavior.
Unclear Connection between Test and Additional Coverage (P13-N–, P16-N–, P20-N–): In three projects, we encountered tests where it was unclear how the amplified change or the generated assertion leads to the new coverage reported by the amplification. In contrast to the coverage false positives, we did not find a test executing the same instructions, but we could not trace how the method calls in the new test would lead to execute the covered instructions.

5.2.2 Positive Selection Criteria

The positive selection criteria are divided into two groups: selecting the most valuable test, or one that we were curious about for our study. In seven projects, we did not need to apply any positive criteria, as there was only one test generated (P3-PDC, P9-EDM, P18-ENC, P21-EDM, P44-EDO, P47-ENC, P49-PDC). In 13 projects, the negative selection criteria already excluded all generated tests, we excluded these projects from the rest of our study (P6-N–, P11-N–, P13-N–, P16-N–, P19-N–, P20-N–, P22-N–, P23-N–, P29-N–, P32-N–, P37-N–, P40-N–, P43-N–).

We used the following criteria for the positive selection:
Most Additional Coverage (P3-PDC, P9-EDM, P18-ENC, P21-EDM, P44-EDO, P47-ENC, P49-PDC): In six projects, the

TABLE 1: The open source projects used in our study, including metrics to show their size, activity and age. Metrics were collected through the SEART GitHub search [52] on 2023-09-14. Each pull request can be accessed by the hyperlink in the third column, or via https://github.com/<project>/pull/<pr number>.

ID	Project	Our Pull Request	Lines of Code	Commits	Contributors	Pull Requests	Creation Year
P1-EDC	apache/commons-io	#358	55k	4.323	97	478	2009
P2-EDC	apache/curator	#418	57k	2.817	114	478	2014
P3-PDC	apache/guacamole-client	#731	118k	6.616	80	909	2016
P4-EDM	apache/httpcomponents-core	#349	82k	3.742	65	424	2009
P5-ENM	apache/unomi	#436	78k	2.608	44	645	2015
P6-N–	apache/zookeeper	–	182k	2.511	191	2.056	2009
P7-EDM	authme/authmereloaded	#2562	69k	4.131	111	740	2013
P8-PDM	axonframework/axonframework	#2244	158k	10.281	154	1.679	2011
P9-EDM	cloudbees-oss/zendesk-java-client	#480	15k	953	62	425	2013
P10-EDM	decorators-squad/eo-yaml	#504	15k	944	20	240	2016
P11-N–	dependencytrack/dependency-track	–	314k	3.946	94	963	2013
P12-PNC	digitalpebble/storm-crawler	#974	51k	1.815	39	344	2013
P13-N–	dius/java-faker	–	62k	834	83	515	2011
P14-PDM	eclipse/lemminx	#1228	511k	1.305	38	824	2018
P15-PNM	ff4j/ff4j	#571	71k	1.413	80	367	2013
P16-N–	firebase/firebase-admin-java	–	85k	447	42	610	2017
P17-EDC	gitlab4j/gitlab4j-api	#852	50k	2.169	145	362	2014
P18-ENC	glyptodon/guacamole-client	#470	118k	6.608	79	471	2013
P19-N–	hangarmc/hangar	–	66k	2.874	41	860	2020
P20-N–	hibernate/hibernate-tools	–	51k	3.177	16	4.415	2011
P21-EDM	hyperledger/fabric-chaincode-java	#244	17k	490	35	282	2017
P22-N–	jenkinsci/email-ext-plugin	–	21k	1.748	95	484	2010
P23-N–	jenkinsci/jira-plugin	–	15k	1.481	79	546	2010
P24-PNC	jqno/equalsverifier	#654	36k	2.884	31	542	2015
P25-ENM	jsqlparser/jsqlparser	#1568	52k	2.030	112	420	2011
P26-ENO	jtablesaw/tablesaw	#1124	1.182k	2.514	80	467	2016
P27-ENM	lukas-krecan/jsonunit	#530	14k	1.549	39	461	2012
P28-PNO	maven-nar/nar-maven-plugin	#389	42k	1.277	71	213	2009
P29-N–	mcmmo-dev/mcmmo	–	56k	6.627	165	631	2012
P30-EDM	miso-lims/miso-lims	#2680	342k	4.801	20	2.596	2012
P31-PDC	moquette-io/moquette	#680	20k	1.394	41	316	2014
P32-N–	mybatis/guice	–	16k	1.809	25	520	2013
P33-ENM	nats-io/nats.java	#663	56k	1.578	48	591	2015
P34-ENO	netflix/zuul	#1265	26k	1.512	54	1.080	2013
P35-PDC	nlpchina/elasticsearch-sql	#1179	145k	1.010	30	250	2014
P36-PDM	oblac/jodd	#788	36k	5.364	57	267	2012
P37-N–	open-metadata/openmetadata	–	639k	7.322	176	7.347	2021
P38-ENC	openhft/chronicle-queue	#1115	41k	7.516	58	705	2013
P39-PND	perwendel/spark	#1257	12k	1.067	124	528	2011
P40-N–	pwm-project/pwm	–	186k	3.063	41	293	2015
P41-EDO	qos-ch/logback	#574	74k	4.451	113	644	2009
P42-PDM	redis/jedis	#3019	70k	2.269	188	1.680	2010
P43-N–	redouane59/twittered	–	47k	701	24	278	2020
P44-EDO	rickfast/consul-client	#461	11k	556	72	255	2014
P45-PDM	rubenlagus/telegrambots	#1070	33k	1.050	91	474	2016
P46-EDO	spotify/dbeam	#486	6k	821	14	645	2017
P47-ENC	spring-projects/spring-data-couchbase	#1461	40k	1.210	48	589	2013
P48-EDM	synthetichealth/synthea	#1082	1.015k	4.662	68	728	2016
P49-PDC	teamnewpipe/newpipeextractor	#850	155k	2.479	64	642	2017
P50-PDM	wikidata/wikidata-toolkit	#691	44k	1.891	28	553	2014
P51-PDC	xerial/sqlite-jdbc	#741	30k	1.521	110	383	2014
P52-EDM	zsmartsystems/com.zsmartsystems.zigbee	#1333	165k	1.180	29	1.080	2017

test we selected covered the most additional instructions. This takes little effort, as the tests in each class are already sorted according to their additional coverage contribution.
Understandability (P12-PNC, P15-PNM, P25-ENM, P26-ENO, P28-PNO, P39-PND, P41-EDO, P46-EDO): In nine projects, we selected tests based on their understandability, as we expect an easy to understand test to more likely be accepted. For this, three criteria emerged that we used in conjunction: a) the coverage improvement is local to a few, closely related methods, b) the connection from the test to the additionally covered methods is clear from the methods called in the test, and c) the test is small and simple.

On several occasions, we choose a candidate test because we were ***curious about the developer's reaction***. In all these cases, we still only considered tests we believe to be a valuable contribution to the project. Non-valuable tests are identified by the negative selection criteria discussed before.
Exception Test (P10-EDM, P17-EDC, P24-PNC, P30-EDM, P34-ENO, P35-PDC, P38-ENC, P42-PDM, P51-PDC): In nine projects, we selected a test that checks for an exception.
Could Be Considered Not Worth Testing (P7-EDM, P8-PDM, P31-PDC, P36-PDM, P45-PDM, P52-EDM): In six projects, the test was contributing coverage in methods that developers could consider not valuable to test, such as a complex setters, `toString`, or `equals`.
Documentation Mismatch (P27-ENM): In P27-ENM we selected a test whose behavior did not match with the documentation of the method under test.
Improve Assertion Manually (P33-ENM): For P33-ENM, we were curious if we can improve an assertion that is not checking the newly covered code.
Uncommonly Large Coverage Increase (P50-PDM): In P50-PDM, one small method call lead to a lot of new coverage, more than what we saw throughout the study.

> **Answer to RQ1.1:** When selecting tests for the pull requests, we mainly excluded coverage false positives, tests with assertions that do not check the newly covered code, or tests that check for unintended runtime exceptions.

5.3 RQ1.2: Which manual edits do we perform to improve the tests before submission?

In this section we present the checklist that we created to guide our manual editing step before opening pull requests.
Align Assertion Style (P4-EDM, P5-ENM, P7-EDM, P9-EDM, P10-EDM, P17-EDC, P25-ENM, P26-ENO, P27-ENM, P30-EDM, P33-ENM, P34-ENO, P38-ENC, P41-EDO, P44-EDO, P47-ENC, P52-EDM): The edit we performed in the largest number of projects (17) was to align the assertion style with the other tests. Examples include: statically importing `assertEquals`, and unifying the assertion framework, e.g., transforming plain JUnit assertions to their Hamcrest versions. DSpot did not remove Hamcrest assertions, so we had to remove old, no longer matching assertions.
Remove Unnecessary Code (P2-EDC, P4-EDM, P5-ENM, P7-EDM, P10-EDM, P26-ENO, P34-ENO, P38-ENC, P41-EDO, P44-EDO, P47-ENC, P48-EDM, P52-EDM): The second

most prevalent edit (13 projects) was to remove variables and statements that were not relevant for the asserted behavior of the amplified test. These are left over from the original test, or temporary variables created by the test amplification and missed during their intended removal. In rare cases we also had to remove unnecessary casts or parentheses, introduced by the test generation for safety.

Adapt To Match Other Edits (P5-ENM, P7-EDM, P18-ENC, P21-EDM, P33-ENM): In five projects, we had to adapt the description of the test to match our manual edits. In P5-ENM we also adapted the test name and the expected value of the assertion to match the behavior that changed during our edits.

Apply IDE Recommendation (P2-EDC, P17-EDC, P52-EDM): In three projects, IntelliJ proposed a simplification through static analysis, e.g., reducing an always true condition.

Resolve Formatting and Linters (P8-PDM, P10-EDM, P26-ENO, P46-EDO): The contribution guidelines of projects sometimes state to apply auto-formatting (P8-PDM, P10-EDM, P46-EDO) or resolve all linter warnings (P10-EDM) before finalizing a pull request. In P26-ENO, we added line breaks to long lines to improve the readability.

Change Test Name (P25-ENM, P34-ENO, P52-EDM): We changed the test name to avoid duplication with existing tests (P25-ENM, P52-EDM), or make the test name fit the convention of the other test names in the class (P34-ENO).

Resolve Unrelated Amplified Change, Additional Coverage or Generated Assertion (P5-ENM, P21-EDM, P33-ENM): We encountered tests where the amplified change, additional coverage, or generated assertion were unrelated. In two cases, we changed the assertion to check the behavior of the newly covered code (P21-EDM, P33-ENM). In P5-ENM and P33-ENM the amplified change and the new assertion provided additional coverage, but they were not related to each other. We selected one test goal and adapted the rest of the test.

Move Test (P1-EDC, P7-EDM, P10-EDM, P52-EDM): In two cases (P1-EDC, P10-EDM), the object under test and the additional coverage were not related to the test class of the original test. We moved the tests to a better fitting class. In two other projects (P7-EDM, P52-EDM), we added our tests below other tests that were targeting the same method.

Simplify Literals (P7-EDM, P21-EDM, P46-EDO): For three tests, we simplified literal values in the test setup. For example, we removed extra clauses from a constructed SQL query that were not relevant for the new test (P46-EDO).

Make Compile (P17-EDC, P25-ENM): In two projects, we found parameters that no longer fit the signature of the called method. We adapted them, e.g., by copying over variable initializations from other tests (P25-ENM).

Answer to RQ1.2: When manually editing the amplified tests, we most often aligned the assertions' style to the test class and removed code unnecessary for the test scenario.

5.4 RQ2.1: Which changes are proposed during the pull request discussion?

Here we present the changes discussed by the maintainers on the pull requests with amplified tests, structured along the categories that emerged from our analysis.

Code Style Conventions (P1-EDC, P8-PDM, P14-PDM, P33-ENM, P42-PDM, P47-ENC, P50-PDM, P51-PDC): Most frequently, the maintainers proposed changes to let the code adhere to style conventions [53], [54], [55], [56]. This regarded aligning the static import of assertion methods (P1-EDC, P8-PDM, P14-PDM, P50-PDM) or used constants (P42-PDM) to the rest of the class, adding a blank line at the end of the file (P33-ENM), or listing our name among the authors of the file in the comment block (P47-ENC), resolving linter warnings (P1-EDC) to make the CI pass (P51-PDC), or adhering to variable naming conventions (P1-EDC). While these seem like conventions of the project, they were not explicitly stated in the contribution guidelines we examined before each pull request.

Remove Unnecessary Code (P12-PNC, P14-PDM, P49-PDC, P50-PDM): The next most frequently discussed change was removing unnecessary code. Three maintainers pointed to unused variables (P12-PNC, P49-PDC, P50-PDM). The test in P14-PDM saved the return value of a relevant method call in an unused variable. In P12-PNC the maintainer criticized a statement that had no impact on the test result, and in P49-PDC the reviewer pointed to unnecessary parentheses.

Change Test Name (P4-EDM, P8-PDM, P10-EDM, P14-PDM): In four pull requests the reviewers suggested changing the test name. The proposed names described the scenario of the method calls in the test (P4-EDM, P8-PDM, P10-EDM), or the exception expected by the assertion (P14-PDM). For P10-EDM, the maintainer explained their naming convention: *"all test names should follow the pattern xDoesSomething"*.

Practice Defensive Programming (P1-EDC, P4-EDM, P49-PDC): Over three projects we got five proposals related to defensive programming. The maintainer of P49-PDC suggested to not check for the complete message of an exception, which could fail if the code under test is refactored. The same reviewer asked to use interfaces instead of concrete implementations and to set variables as final where possible. The review of P4-EDM proposed to assert the return value of an intermediate call. The reviewer of P1-EDC advised to use the specialized try-with-resources when writing to an InputStream within a `try` environment.

Simplify Setup (P1-EDC, P8-PDM, P10-EDM, P14-PDM): The maintainers of four projects proposed to simplify the test setup. For example, in P8-PDM, we replaced a multiple times modified object with a fitting default instance. The reviewer of P14-PDM recognized that another call than one under test could throw the expected exception and proposed a change to avoid the tests passing because of the earlier thrown exception.

Choose More Powerful Assertion (P8-PDM, P10-EDM, P49-PDC): Three maintainers pointed to the benefit of using a stronger assertion method. For example, in P8-PDM they endorsed a change from `assertFalse(...equals())` to `assertNotEquals(..)`.

Merge or Extend Test (P3-PDC, P42-PDM, P48-EDM): Three

projects discussed merging the contributed test with other tests for the same method. P3-PDC and P42-PDM pointed to moving the assertion to an existing test. The maintainer for P48-EDM proposed to add an assertion to test a second scenario in the method under test and was open to keep both in the same test or split them up into two unit tests.
Use Meaningful Scenario (P7-EDM, P25-ENM, P47-ENC): Three maintainers proposed changing the test setup to a more meaningful scenario. For example, the test for P25-ENM used default initializations for SQL queries. The reviewer of P25-ENM criticized that the queries were not meaningful, and asked to *"craft an actual valid expression."*
Move Test (P12-PNC, P50-PDM, P47-ENC): Three reviews asked to move the test to another class as it tested a different object than the original test modified by the amplification.
Change Assertion Message (P42-PDM, P30-EDM): The maintainers of P42-PDM and P30-EDM both proposed to change the assertion message to explain why the code throws the exception that is expected by the test case.
Move Test Data (P1-EDC): For P1-EDC we moved the amplified test to another class, including globally defined test data. The maintainer asked us to move the test data into the test itself, as it was the only test using the data.
Test All Scenarios (P48-EDM): In P48-EDM the reviewer proposed to add a second assertion, to let the resulting test check for both the succeeding and failing scenario.

Answer to RQ2.1: The majority of changes proposed during the pull request reviews were focused on adhering to code style conventions and removing unnecessary code.

Fig. 6 looks closer at the connection between whether we manually edited a test and whether changes were proposed during the review. We observe that for both edited and not edited tests the maintainers were **more often proposing changes than not**. Three tests without edits were merged without any further changes, while in six projects the pull requests were closed even when changes were discussed. The latter happened, e.g., because through the discussion it became clear that the test is redundant to existing tests (P1-EDC), or the maintainers provided feedback on the code even though they already concluded to not accept the test (P49-PDC).

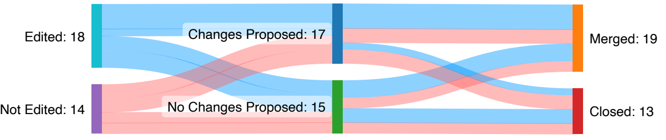

Fig. 6: Flow of editing tests, changes proposed during the pull request and pull request outcome.

5.5 RQ2.2: What kind of information is requested by the maintainers during the pull request discussion?

Next to proposing changes, the maintainers also requested different kinds of information during the discussions:
Purpose of the Pull Request / Test (P3-PDC, P12-PNC, P25-ENM, P27-ENM): Four reviewers asked to explain the purpose of the pull request or the test, such as *"I'm unsure what issue this is targeting at resolving"* (P3-PDC), or *"what problem exactly will this PR solve?"* (P25-ENM).
Added Value (P2-EDC, P25-ENM, P27-ENM, P51-PDC): In four cases, we were asked about the added value that the test is providing.
Coverage Increase (P1-EDC): One maintainer included a coverage tool, checking the coverage increased.
Description about the Test (P33-ENM): For P33-ENM we did not include the textual description at first, but we were asked to add a description about our test into the pull request.
Contribution Compared to Existing Tests (P1-EDC): The maintainer of P1-EDC asked what our test checks in comparison to existing tests for the same method.
Curiosity (P7-EDM, P14-PDM, P24-PNC, P50-PDM): Three reviewers asked questions out of curiosity, such as "how [our tool] generated the parameter input" (P7-EDM), which IDE and formatter we used (P14-PDM), and how we came to writing a test for this specific method (P24-PNC, P50-PDM).

Fig. 7 presents a closer analysis of the relationship between whether we provided a description in the initial pull request (such as in Fig. 5) and whether additional information was requested by the reviewers (excluding curious questions). We can see that **questions appeared just as often whether we provided the generated description or not** (4 projects each), and two pull requests without description were merged without requests for more information. In contrast, **curious questions** on the details of our process **were mainly asked for pull requests with a description**. When we provided a description, giving additional information never lead to a merged pull request (4 projects), while the majority of pull requests with a description were merged without further requests for clarification (14 projects).

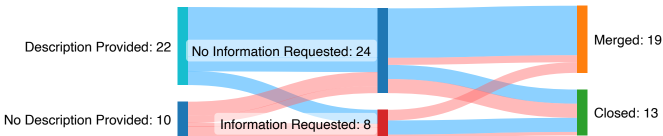

Fig. 7: Flow of description provided, information requested during the pull request and pull request outcome.

Answer to RQ2.2: The maintainers mostly asked for more information regarding the purpose and value of the contributed test.

5.6 RQ2.3: How do the maintainers justify their judgment over the amplified tests during the pull request discussion?

Another aspect we analyzed were the reasons that reviewers accepted or rejected our pull requests.
Completeness of Contribution (P3-PDC, P5-ENM, P31-PDC, P47-ENC, P49-PDC): Three reviews pointed out that the contribution was not complete enough. This was because all possible outcomes of a method should be tested (P31-PDC, P49-PDC), only a more comprehensive set of changes would be worth merging (P3-PDC), or an issue tracker entry (P5-ENM) needs to exist, and a discussion should happen before including a patch (P3-PDC).

Would Not Test (P2-EDC, P9-EDM, P49-PDC, P51-PDC): Three maintainers pointed out that the test was targeting methods they would not test, such as simple methods (P2-EDC, P49-PDC), classes taken from libraries (P51-PDC), or `toString` as it is used for debugging only (P9-EDM).
Test Untested Scenarios (P1-EDC, P9-EDM, P24-PNC, P27-ENM, P52-EDM): It was important to the reviewers that the proposed tests were testing yet untested scenarios. In P52-EDM and P9-EDM this was the rationale to merge the pull request, in P1-EDC and P24-PNC this was the reason to close the pull requests as the maintainers found other tests for the same scenarios. The reviewer of P27-ENM pointed out that *"ideally there should be some intention behind each test."*
Clear Test Scenario (P25-ENM, P27-ENM, P38-ENC): Three maintainers mentioned a meaningful scenario (P25-ENM) and clarity about what the test is testing (P27-ENM, P38-ENC).
Code Quality (P1-EDC, P31-PDC:) The reviewer of P1-EDC pointed out that the code should pass the linter. The maintainer of P31-PDC criticized that some code in the test is irrelevant for the method under test.

In several cases, we have no indication of the rationale for accepting or rejecting the pull request: Four projects merged (P15-PNM, P21-EDM, P36-PDM, P45-PDM) and two closed (P35-PDC, P17-EDC) our pull request without any comment.

Answer to RQ2.3: When verbalizing a rationale for their judgment on the amplified tests, the project maintainers mentioned the need for a comprehensive contribution of tests for meaningful, untested scenarios.

6 Discussion

In the previous section, we reported on the selection and manual edits we conducted before submitting the tests in pull requests, as well as the reactions of the maintainers concerning proposed changes, requested information, and rationale for their decisions to accept or reject the proposed tests. To connect our observations, we summarize the guidelines for developers to select and edit amplified tests in Table 2. Further in this section, we discuss the implications of our findings for developers that consider using developer-centric test amplification, and for test amplification researchers and tool designers. We also present threats to the validity of our study.

6.1 Guidelines for Developers to Select and Edit Amplified Tests

A strong take-away from our study is that the tests created by state-of-the-art test amplification tools still needed selection and editing efforts before they are incorporated into a maintained test suite. To summarize and connect the observations we made for our five research questions, we present guidelines for developers on what aspects they should consider when reviewing an amplified test. Here, selection and editing are put together and the decision which action to take is left to the developer. If an issue is too large, or it it is unclear how to resolve it, the developer might choose to exclude the test entirely. If they see an easy change to address the issue, they might choose to edit the test and include it in their maintained test suite. Table 2 gives an overview and explanation of each of our guidelines, as well as the observations from our study that it is based on.

We recommend, that a developer using developer-centric test amplification, should review each test individually and consider whether:

- the newly covered code is indeed not yet covered by any other test,
- the newly covered code or scenario is relevant to be tested in their maintained test suite,
- the test only contains code necessary for its behavior or understandability,
- the assertion in the test validates the behavior of the newly covered code,
- the test behavior and its impact on the test suite is understandable to them and their colleagues,
- the code style is adequate and adheres to their coding guidelines,
- the test is at an appropriate location and whether it should be merged or extended with another test.

6.2 Relation to existing Literature

Several of the edits to amplified tests we observed in our study are related to existing knowledge about high-quality tests and shortcomings in automatically generated tests. This section illustrates how each of our guidelines is supported by existing literature. However, to our knowledge, there is no research looking at what changes developers concretely make to generated or amplified tests before including them in a test suite.

6.2.1 Valid Coverage Improvement

Our first guideline is that the targeted code should not be covered by another test that might not have been considered by the coverage data used by the test amplification process. We observed something similar in an industrial study where developers considered code that was accounted for in other quality assurance practices or test suites to be not as relevant to test with a regression test [57]. In the concrete cases, the code blocks were covered by fuzzing, so the developers might have seen this robustness testing as sufficient. While improving an engineering goal such as coverage or mutation score is at the heart of the definition of test amplification [20], we see in this study that in practice we cannot always rely on the coverage data that test amplification tools use. This data might exclude other tests, higher-level test suites or other quality assurance practices.

6.2.2 Tests Relevant Code/Scenario in Project

When testing software, developers need to decide which code is worth testing with automated tests [58]. In other studies we conducted, we observed that not all code is relevant for developers to cover with regression tests [23], [57]. This is in line with the common recommendation to not aim for 100% code coverage [59], [60]. In interviews with developers, Kochhar et al. found that the judgment what to test is subjective, as participants disagreed whether it is useful to test simple things [40]. There can also be behaviors of code that should not be tested. Galindo-Guiterrez et al.

Concern in Amplified Test	Connected Codes / Observations (Source RQ)	Explanation
Valid Coverage Improvement	Test Untested Scenarios (2.3) Added Value, Coverage Increase, Contribution Compared to Existing Tests (2.2) Coverage False Positive (1.1)	Check that the targeted code is not tested by another test (which might not be considered by amplification tool or coverage data)
Tests Relevant Code/Scenario in Project	Use Meaningful Scenario (2.1) Would Not Test (2.3) No Explicitly Thrown Exception (1.1)	Check that the new coverage provided by the test covers code that is relevant to test with your test suite
Only Necessary Code	Change Unrelated to Assertion or New Coverage (1.1) Remove Unnecessary Code (1.2, 2.1) Resolve Unrelated Amplified Change, Additional Coverage or Generated Assertion (1.2)	Check that all code in the test is relevant for the test's execution or understandability
Checks Behavior of Newly Covered Code	Assertion Unrelated to New Coverage (1.1) Resolve Unrelated Amplified Change, Additional Coverage or Generated Assertion (1.2)	Check that the assertion of the test actually validates the behavior of the additionally covered code
Test Scenario and Impact are Understandable	Readability and Understandability (1.1) Simplify Literals (1.2) Simplify Setup (2.1) Change Assertion Message (2.1) Unclear Connection between Test and Additional Coverage (1.1) Clear Test Scenario (2.3) Change Test Name (1.2, 2.1)	Check that you can / your colleagues could understand the test and what it is testing
Good Code Style, Adhering to Guidelines	Code Style Conventions (2.1) Align Assertion Style (1.2) Apply IDE Recommendation (1.2) Resolve Formatting and Linters (1.2) Change Test Name (1.2, 2.1) Practice Defensive Programming (2.1) Choose More Powerful Assertion (2.1) Code Quality (2.3)	Check that the code is well written and adheres to your guidelines
Appropriate Scope and Location	Move Test (1.2, 2.1) Merge or Extend Test (2.1) Change Test Name (1.2, 2.1) Move Test Data (2.1) Test All Scenarios (2.1)	Check that the test is at an appropriate location and has the right granularity (move/merge/extend with other test otherwise)

TABLE 2: Guidelines to select and edit amplified tests

identified checking for `NullPointerException`s that are not explicitly thrown in the code under test as an undesirable behavior of EvoSuite-generated tests.

6.2.3 Only Necessary Code

Our third guideline recommends removing all code that is not necessary for the execution of the test. This code might be left over from the original test that was amplified or no longer needed after other changes to the test. Similarly, the test smell “General Fixture” [61] is based on unnecessary code in test setup methdos, and unnecessary code is also a problem in production code [62]. Panichella et al. [63] propose to use optimization heuristics like purification [64], carving [65] or slicing [66] to improve generated tests by focussing them on one, semantically coherent scenario.

6.2.4 Checks Behavior of Newly Covered Code

Our next guideline concerns the assertions of the amplified tests, which should check the behavior of the newly covered code. It is well known that structural coverage can give an indication whether a test suite is bad, but does not indicate error detection and prevention strength [40], [67], [68]. The ability to reveal faults in the targeted production code is a criterion in Grano et al.'s quality factors for unit tests [69]. A miss-match between the act and assert phase of a test was one of the quality issues Galindo-Gutierrez et al. detected in tests generated by EvoSuite [70]. To address these issues, we could employ more refined metrics to select the amplified tests, such as checked coverage [71], oracle adequacy [72], or mutation score [73]. However, one must way the trade-offs reagarding runtime, because such stronger metrics are generally more expensive to compute [74]. For mutation score, limiting the mutants to relevant lines [75], i.e., the additionally covered lines, could be an option to speed up computation. On the other hand, Zhang et al. [67] found that human-written assertions are stronger at detecting seeded faults than assertions generated by the tool Randoop [76].

6.2.5 Test Scenario and Impact are Understandable

The understandability of tests, or lack thereof, is mentioned in several user-involving studies on automatic test generation [16], [18], [24]. Code reviewers are concerned with the understandability of test that are contributed [77]. The understandability of a test is impacted by test names [17], [43], [78], variable identifiers [41], [79], [80], meaningful comments or summaries [18], [41], and the test data [81], [82], [83], [84], [85]. Lin et al. showed that the quality of identifier names is low in manually written and especially automatically generated tests [79]. The concern with readability of generated tests is a central motivation for the development of language model based test generation approaches [86], [87]. However, it was also shown that the judgment how

readable a test is differs per developer [85], and that experience influences the test comprehension process [82]. Daka et al. observed that developer-given test names could contain abstract knowledge about the test intent or scenarios, which was not the case for their generated names that focused on covered methods and asserted values [17].

6.2.6 Good Code Style, Adhering to Guidelines

Our guideline to ensure that the amplified tests have a good code style and adhere to the coding guidelines of a project, can also be observed in more general code review practices that require consistency of code style [88], [89]. Specifically for assertions, Zamprogno et al. found that developers prefer assertion statements that are consistent with the code style of the test suite [90]. While explicit guidelines on how to contribute to open source projects are more and more common [91], these documents often do not sufficiently reflect the whole process [92], [93] and especially lack information about not automatically checkable guidelines [94].

6.2.7 Appropriate Scope and Location

The final guideline in our list is to ensure that an amplified test has an appropriate scope and is in the right location within the code base. A too large scope, i.e., too much tested in one method, can be the test smell “Eager Test” [61] or a sign of lacking semantic coherence [63]. It also can make the test long, which negatively impacts understandability [40], [80]. Existing literature recommends that test code should be well-modularized and structured [40], [81]. Duplication of test setups over multiple tests is an indication of code clones hindering the maintainability of test code [70], which can be the motivation to merge an amplified test with an existing test from the test suite.

6.3 Implications for Practitioners

In this paper, we characterized the selection and editing steps developers are likely to conduct before incorporating amplified tests into their maintained test suite. For software developers and project managers, our results can be the **basis to take an informed decision on whether to adopt developer-centric test amplification**, by providing a realistic view on the kind of adjustments required by developers. We divide these efforts into two groups: (1) actions that could be automated by customizing the test amplification to a project, and (2) actions that highly benefit from the developer’s comprehension.

To the first category, we count the coverage false positives, additional coverage in not-test-worthy methods, adhering to code style guidelines, and using defensive programming constructs. If a software developer applies test amplification out of the box, without any further customization, they would run into these issues, such as we did during our study. However, if the project would commit to a longer use and invest the time in configuring and customizing the amplification tool for their project, such efforts can potentially be automated.

The other set of efforts require the software developers to understand the amplified test—which they aim for already before accepting the test. These efforts are about changing the scenario of the test to be simpler or more meaningful, removing left over code, moving or merging the test, or adding a clearer test name. With these, the test becomes easier to understand, therefore easier to maintain and more helpful when trying to locate the fault when the test fails. These changes have a large impact on the quality of the resulting test, addressing commonly observed shortcomings of automatically generated tests [16], [24], [80].

6.4 Implications for Researchers and Tool Designers

Previous user studies on test generation and amplification have shown that software developers find it important to understand the produced tests [10], [16], [23]. Understanding the tests was also necessary for us when selecting and editing the amplified tests, just as for the maintainers, who asked for additional clarification when the test or the pull request description were not clear enough. During this study, we elicited several adjustments to amplified tests that require an understanding of the behavior of the test. We conjecture, that such edits are much easier for developers to perform than for an automated tool. The next step for researchers would be to **investigate whether test amplification collaborating with the developer** for changes that require understanding **is an effective alternative to automating** them.

Because understanding is a prerequisite for the developer’s manual edits, we conjecture that it is crucial for developer-centric test amplification tools to **provide the information that developers need to understand and modify the amplified tests**. As we saw in Section 5.5, the descriptions we generate are one component that contributes here, pointing to the amplified changes and the additionally provided coverage. However, throughout our study we experienced that further information support is necessary. For example, visually connecting the methods called in the test with the additional coverage could help developers understand how the amplified test provides this coverage [95]. Developers would also benefit from knowing which other tests cover the same method [18], [96], to determine the difference to these tests, or to validate if all scenarios of a method are tested. When we performed changes to the test scenario, we were at times not sure whether the coverage reported by the test amplification tool is still provided. We hypothesize that a **close integration of test amplification and manual editing** would let the developer verify their changes in the terms of the test amplification tool.

While we plead to leverage the developer’s understanding and expertise to collaboratively produce valuable tests, our results also point to possible improvements of the automatic amplification process. During our selection we encountered tests where the generated assertion was not checking the behavior of the newly covered code. One could **apply local mutation analysis to verify that an assertion is really checking the additionally covered code**, similar to Ma et al.’s commit-aware mutation testing [75]. This means applying mutations only to the newly covered code and evaluating whether they cause the amplified test to fail. This approach would have a better performance than selecting amplified tests on mutation score directly, and we could still use the more widely understood instruction coverage when communicating the value of a test to the developer [97].

We encountered amplified tests that are based on complex, manually written tests whereas their tested scenario did not need this complexity. We propose to **improve test amplification by smartly selecting the original test** to modify, starting from simple tests and continuing to more complex ones. This way, the simple cases that can be tested through test amplification are caught with simple original tests, and the more complex original tests are only used if the amplification covers scenarios that need this complexity.

In the edits we conducted ourselves, as well as the ones proposed by maintainers, we moved tests to other classes, because the test target of the amplified test was no longer the same as the target of the original test. Clearly **identifying the target of an amplified test** would empower amplification tools to propose a better location for the produced test, and to communicate the intended impact of the amplified test clearer to the developer. From our observations, the tests were moved to test classes that are related to the additionally covered methods, or related to the methods directly called in the test.

6.5 Threats to Validity

There are several threats to the validity of our results:

6.5.1 Reliability of Results

To ensure the consistency and reliability of our qualitative analysis' findings, the first two authors revised the emergent codes throughout discussions until they reached a negotiated agreement [49]. We also employed constant comparison [51], whereby each interpretation and finding is compared with existing findings as it emerges from the data analysis to increase the construct validity. Especially for the manual selection and edits we conducted ourselves (adressing **RQ1**), the background of the researchers might have influenced which issues we identified in the amplified tests. Present are the threats of confirmation bias and experimenter bias, where our previous experience of issues with amplified tests leads to us overly focussing on these issues. Independent evaluators with a different background with regards to test generation might have identified other issues. Even when considering the presence of these biases, we deemed the manual selection and editing necessary to avoid antagonizing the open source maintainers by submitting tests that are clearly not ready to be merged. To mitigate the impact of our background, we carefully structured and documented our selection and editing process through the checklists that form the answers to **RQ1.1** and **RQ1.2** and invite other researchers and software engineering practitioners to replicate our study and compare their findings.

6.5.2 Construct Validity

The deficiencies we observed in the amplified tests are closely related to the current state of the test amplification tool DSpot. It is the state-of-the-art for test amplification in Java, and the archetypical implementation of test amplification that other tools are based on [32], [98], [99]. Still, the selection and edit efforts will change when the automation improves in the future. If efforts we observed are automated, developers might be willing to make new kinds of changes to improve the amplified tests. Because we manually selected the amplified tests to submit in pull requests and edited half of them to improve their quality before submitting, the results to **RQ2** do not directly reflect current amplified tests, but rather what test amplification might be capable of in the future. To mitigate this, we carefully document and report the selection and editing checklists we used in the answers to **RQ1** and pull our take-away recommendations on both our manual efforts and the maintainers' feedback in the pull request discussions.

6.5.3 Participant Bias

We did not reveal that the tests were at least partially automatically generated, and the maintainers' feedback might change if they were aware of this. The maintainers could also face a social desirability bias, answering in a way that they expect us or their surroundings to prefer. To mitigate this we did not reveal our exact research questions to them, and conducted the study in their familiar environment of pull request discussions.

6.5.4 Internal Validity

In most of the pull request discussions the maintainers did not communicate their rationale for accepting or rejecting the pull request. We hypothesize that such a judgment is based on a plethora of factors, e.g., the code quality or the coverage contribution. As visible in Table 1, our study includes a diverse set of projects, whose individual size, contributors, or general interaction with pull requests, might influence the acceptance of a pull request. To mitigate the threat of inferring too much from the pull request outcome, we focused our analysis on the concrete discussion comments from the project maintainers.

6.5.5 Generalizability

The threat of internal generalizability concerns whether the sampled study objects are representative of our population of interest: open-source Java projects. We only considered projects where DSpot did not fail during execution, and produced amplified tests within 30 minutes. Other software projects might need a considerably higher up-front effort to adapt the test amplification tool before they can apply it and show a different set of deficiencies in the produced tests. Projects that need more than 30 minutes to build or use external tools that cannot be set up with DSpot's plain Maven or Gradle support, might show a unique set of selection criteria and change wishes from the developers. To mitigate this threat, we focus on providing an overview of the possible selection and change efforts that developers can encounter. While we specify how often each of them occurred in our study, we refrain from hypothesizing how likely they would appear in any project.

With respect to external generalizability and external validity, we acknowledge the need for replication studies with other programming languages, test frameworks or project settings. The feedback from maintainers of less active projects could differ, and industrial projects could have different requirements for automated tests.

7 Related Work

In another open source contribution study [25], Danglot et al. showed that DSpot is able to provide valuable additions

to existing test suites by amplifying tests. They amplified 40 test classes of 10 projects and opened 19 pull requests of which 13 were accepted. Compared to their study, we focus on comprehensively documenting which kind of manual adjustments are necessary before submitting an amplified test, conduct our study on a larger number of repositories and pull requests, and present a detailed analysis of the feedback from the open source maintainers. We previously conducted an exploratory study evaluating an IntelliJ plugin to facilitate developer-centric test amplification from within the developer's IDE [23]. While we gathered a broad variety of feedback through interviews with developers, this work focuses on the concrete changes that maintainers and code owners would make to the amplified tests, independently of IDE tooling.

There have been several studies of search-based test generation with EvoSuite that involved users [10], [16], [24]. Our findings corroborate several results from these studies, such as the importance of readability for the developers [16], [24], that the quality of a test is strongly connected to how easy it is to elicit its behavior [24], and a diversity of preferences for tests between different developers [24]. Daka et al. [80] established identifiers, line length and constructor and method calls as important features of the readability of a test. We go further into analyzing what a developer would change to obtain a satisfactory test from a, potentially less readable, generated one. Similar to us, Almasi et al. [16] asked the participants of their industrial case study what they would change in the generated tests to keep them. Our findings corroborate their results that developers would change the test data, or scenario, and the assertions to more meaningful ones. In contrast to their study, our open source contribution study spreads over a larger variety of projects. We point to a greater diversity of concrete changes that were important to the projects we contributed to, such as aligning with code style conventions, or moving and merging tests.

In a large-scale, manual study of EvoSuite generated tests, Galindo-Gutierrez et al. [70] identified 13 new quality issues in automatically generated test cases, which are not covered by the previous definitions of test smells [61], [63], [100]. While our study is based on a different test generation approach and tool, several of their quality issues coincide with the deficiencies we observe in DSpot amplified tests, and which we recommend developers to consider when selecting and editing amplified tests. They name three quality issues concerning a mismatch between the act and assert sections of the test case, which correspond to our observations of unrelated amplified change, additional coverage or generated assertion (**RQ1**). Our filter criterion "No Explicitly Thrown Exception" is also present in their list of quality issues. A set of their collected issues does not apply to the approach of test amplification, where one test is generated and then integrated into an existing test suite. These issues concern code and test scenarios that are redundant between the many tests EvoSuite generates, or violate the stricter unit testing paradigm aimed at by EvoSuite, i.e., only testing behavior directly in the class under test.

Incorporating the developer's expertise into the test amplification process, is also central in interactive search-based test generation [101], [102], [103]. In contrast to this field, we do not ask the developer to provide specific types of judgments to improve the search process, but instead they customize the amplified test to its final state for their test suite.

Several previous works investigated generating descriptions for automatically generated [18], [41], [104] and manually written tests [105], [106] and have shown that these descriptions help developers understand the tests [18], [41], [106]. Similar to us, these approaches leverage the called and covered methods to describe the intention of the test case. Our description is specialized for amplified tests, focussing on the amplified change and new assertion, while referring to the original test, leading to a shorter description.

8 Conclusion

In this paper, we manually analyzed the amplified tests of 52 projects, and discussed them through 39 pull requests with their open source maintainers. In a nutshell, we contribute:

- Insights into the selection and manual editing we performed to prepare the amplified tests for a pull request.
- Insights into the proposed changes, requested information and judgment of open source maintainers towards developer-centric amplified tests.
- Improvements to the test suites of 19 open source projects through our accepted pull requests.

Throughout the whole study we repeatedly observed that amplified tests need to be understood by developers before they consider including the tests into their maintained test suite. This understanding was also the basis for several kinds of edits we made and changes that were proposed by the maintainers, opening up a fundamental question for researchers working on developer-centric test amplification:

> Should we focus on further automating test amplification or focus on supporting developers in understanding the amplified tests, leaving some edits to them?

Therefore, the next steps in this line of research are to investigate this tradeoff and to develop tools that support developers with information and actionable recommendations while editing amplified tests. Further, we want to improve the state-of-the-art of test amplification by automating the now manual efforts and by sharpening the quality of the amplified tests through local mutation analysis. We encourage researchers to validate whether our results hold for other programming languages and test generation tools.

References

[1] K. L. Beck, *Test-Driven Development - By Example*, ser. The Addison-Wesley signature series. Addison-Wesley, 2003.

[2] M. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Trans. Software Eng.*, vol. 48, no. 12, pp. 4925–4946, 2022.

[3] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1100–1125, 2014.

[4] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" in *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE CS, 2015, pp. 559–562.

[5] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 179–190.

[6] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the IDE: patterns, beliefs, and behavior," *IEEE Trans. Software Eng.*, vol. 45, no. 3, pp. 261–284, 2019.

[7] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 742–762, 2010.

[8] L. Baresi and M. Miraz, "TestFul: Automatic unit-test generation for java classes," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 281–284.

[9] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and European Software Engineering Conference (ESEC)*. ACM, 2011, pp. 416–419.

[10] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? A controlled empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 23:1–23:49, 2015.

[11] M. Swillus and A. Zaidman, "Sentiment overflow in the testing stack: Analysing software testing posts on stack overflow," *J. Syst. Softw.*, vol. 205, 2023.

[12] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE CS, 2011, pp. 23—32.

[13] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for java," *Softw. Pract. Exp.*, vol. 34, no. 11, pp. 1025–1050, 2004.

[14] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Botsing, a search-based crash reproduction framework for java," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1278–1282.

[15] P. Derakhshanfar, X. Devroey, A. Zaidman, A. van Deursen, and A. Panichella, "Good things come in threes: Improving search-based crash reproduction with helper objectives," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 211–223.

[16] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE CS, 2017, pp. 263–272.

[17] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 57–67.

[18] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 547–558.

[19] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.

[20] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *J. Syst. Softw.*, vol. 157, p. 110398, 2019.

[21] M. Nassif, A. Hernandez, A. Sridharan, and M. P. Robillard, "Generating unit tests for documentation," *IEEE Trans. Software Eng.*, 2021.

[22] STAMP, "Use cases validation report v3," https://github.com/STAMP-project/docs-forum/blob/master/docs/, 2019.

[23] C. Brandt and A. Zaidman, "Developer-centric test amplification," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 96, 2022.

[24] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated unit test generation during software development: A controlled experiment and think-aloud observations," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 338–349.

[25] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus, "Automatic test improvement with DSpot: A study with ten mature open-source projects," *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 2603–2635, 2019.

[26] C. Brown and C. Parnin, "Sorry to bother you: Designing bots for effective recommendations," in *International Workshop on Bots in Software Engineering (BotSE)*. IEEE/ACM, 2019, pp. 54–58.

[27] A. Khatami and A. Zaidman, "State-of-the-practice in quality assurance in open source software development—replication package," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6563549

[28] B. Danglot, M. Monperrus, W. Rudametkin, and B. Baudry, "An approach and benchmark to detect behavioral changes of commits in continuous integration," *Empir. Softw. Eng.*, vol. 25, no. 4, pp. 2379–2415, 2020.

[29] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[30] R. Bloem, R. Koenighofer, F. Röck, and M. Tautschnig, "Automating test-suite augmentation," in *International Conference on Quality Software*. IEEE, 2014, pp. 67–72.

[31] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2010, pp. 257–266.

[32] M. F. Roslan, J. M. Rojas, and P. McMinn, "An empirical comparison of EvoSuite and DSpot for improving developer-written test suites with respect to mutation score," in *International Symposium on Search-Based Software Engineering (SSBSE)*, ser. LNCS, vol. 13711. Springer, 2022, pp. 19–34.

[33] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023.

[34] O. Nourry, Y. Kashiwa, B. Lin, G. Bavota, M. Lanza, and Y. Kamei, "The human side of fuzzing: Challenges faced by developers during fuzzing activities," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, nov 2023.

[35] S. Plöger, M. Meier, and M. Smith, "A usability evaluation of AFL and libfuzzer with CS students," in *Conference on Human Factors in Computing Systems (CHI)*. ACM, 2023, pp. 186:1–186:18.

[36] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2676–2692.

[37] M. Olsthoorn, A. van Deursen, and A. Panichella, "Generating highly-structured input data by combining search-based testing and grammar-based fuzzing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1224–1228.

[38] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Softw.*, vol. 38, no. 3, pp. 79–86, 2021.

[39] D. Hoffman and P. Strooper, "API documentation with executable examples," *J. Syst. Softw.*, vol. 66, no. 2, pp. 143–156, 2003.

[40] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE/ACM, 2019, pp. 61–70.

[41] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "DeepTC-Enhancer: Improving the readability of automatically generated tests," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 287–298.

[42] W. Oosterbroek, C. Brandt, and A. Zaidman, "Removing redundant statements in amplified test cases," in *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021, pp. 242–246.

[43] N. Nijkamp, C. Brandt, and A. Zaidman, "Naming amplified tests based on improved coverage," in *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021, pp. 237–241.

[44] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *J. Syst. Softw.*, vol. 106, pp. 82–101, 2015.

[45] S. Sinha and M. J. Harrold, "Criteria for testing exception-handling constructs in java programs," in *International Conference on Software Maintenance (ICSM)*. IEEE CS, 1999, p. 265.

[46] Anonymous Authors, "Replication package for "Shaken, not stirred. How developers like their amplified tests"," https://doi.org/10.5281/zenodo.7034924, 2023.

[47] R. Lakshmanan, "Minnesota university apologizes for contributing malicious code to the linux project." [Online]. Available: https://thehackernews.com/2021/04/minnesota-university-apologizes-for.html

[48] A. Khatami and A. Zaidman, "State-of-the-practice in quality assurance in java-based open source software development," *Software: Practice and Experience (SP&E)*, 2024, to appear.

[49] D. R. Garrison, M. Cleveland-Innes, M. Koole, and J. Kappelman, "Revisiting methodological issues in transcript analysis: Negotiated coding and reliability," *The internet and higher education*, vol. 9, no. 1, pp. 1–8, 2006.

[50] A. L. Strauss and J. M. Corbin, "Basics of qualitative research: Techniques and procedures for developing grounded theory," *SAGE Publications*, 1998.

[51] B. G. Glaser and A. L. Strauss, *Discovery of Grounded Theory: Strategies for Qualitative Research*. Routledge, 2017.

[52] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 560–564.

[53] G. Gousios, A. Zaidman, M. D. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE CS, 2015, pp. 358–368.

[54] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 430–448, 2009.

[55] M. Beller, A. Bacchelli, A. Zaidman, and E. Jürgens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 202–211.

[56] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE CS, 2013, pp. 712–721.

[57] C. Brandt, M. Castelluccio, C. Holler, J. Kratzer, A. Zaidman, and A. Bacchelli, "Mind the gap: What working with developers on fuzz tests taught us about coverage gaps," in *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2024.

[58] M. Aniche, *Effective Software Testing: A Developer's Guide*. Simon and Schuster, 2022.

[59] M. Gittens, K. Romanufa, D. Godwin, and J. Racicot, "All code coverage is not created equal: A case study in prioritized code coverage," in *Conference of the Centre for Advanced Studies on Collaborative Research*. IBM, 2006, pp. 131–145.

[60] B. Marick, J. Smith, and M. Jones, "How to misuse code coverage," in *International Conference on Testing Computer Software*, 1999, pp. 16–18.

[61] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.

[62] S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K. Prommer, "How much does unused code matter for maintenance?" in *International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2012, pp. 1102–1111.

[63] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Test smells 20 years later: Detectability, validity, and reliability," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 170, 2022.

[64] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 52–63.

[65] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Trans. Software Eng.*, vol. 35, no. 1, pp. 29–45, 2009.

[66] S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. C. Briand, "Log-based slicing for system-level test cases," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021, pp. 517–528.

[67] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 214–224.

[68] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan, "Code coverage and postrelease defects: A large-scale study on open source projects," *IEEE Trans. Reliab.*, vol. 66, no. 4, pp. 1213–1228, 2017.

[69] G. Grano, C. D. Iaco, F. Palomba, and H. C. Gall, "Pizza versus pinsa: On the perception and measurability of unit test code quality," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 336–347.

[70] G. Galindo-Gutierrez, M. N. Carvajal, A. F. Blanco, N. Anquetil, and J. P. S. Alcocer, "A manual categorization of new quality issues on automatically-generated tests," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023.

[71] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2011, pp. 90–99.

[72] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 391–400.

[73] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 402–411.

[74] C. Brandt, D. Wang, and A. Zaidman, "When to let the developer guide: Trade-offs between open and guided test amplification," in *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 231–241.

[75] W. Ma, T. Laurent, M. Ojdanic, T. T. Chekam, A. Ventresque, and M. Papadakis, "Commit-aware mutation testing," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 394–405.

[76] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE CS, 2007, pp. 75–84.

[77] D. Spadini, M. F. Aniche, M. D. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in *IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 677–687.

[78] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 625–636.

[79] B. Lin, C. Nagy, G. Bavota, A. Marcus, and M. Lanza, "On the quality of identifiers in test code," in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2019, pp. 204–215.

[80] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 107–118.

[81] D. Winkler, P. Urbanke, and R. Ramler, "What do we know about readability of test code? - A systematic mapping study," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 1167–1174.

[82] C. S. Yu, C. Treude, and M. F. Aniche, "Comprehending test code: An empirical study," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 501–512.

[83] A. Deljouyi and A. Zaidman, "Generating understandable unit tests through end-to-end test scenario carving," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 107–118.

[84] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "What factors make SQL test cases understandable for testers? A human study of automated test data generation techniques," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 437–448.

[85] P. Delgado-Pérez, A. Ramírez, K. J. Valle-Gómez, I. Medina-Bulo, and J. R. Romero, "InterEvo-TR: Interactive evolutionary test generation with readability assessment," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2580–2596, 2023.

[86] N. Rao, K. Jain, U. Alon, C. L. Goues, and V. J. Hellendoorn, "CAT-LM training language models on aligned code and tests," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 409–420.

[87] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[88] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 181–190.

[89] W. Zou, J. Xuan, X. Xie, Z. Chen, and B. Xu, "How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3871–3903, 2019.

[90] L. Zamprogno, B. Hall, R. Holmes, and J. M. Atlee, "Dynamic human-in-the-loop assertion generation," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2337–2351, 2023.
[91] A. Khatami and A. Zaidman, "Quality assurance awareness in open source software projects on GitHub," in *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 174–185.
[92] M. Guizani, A. Chatterjee, B. Trinkenreich, M. E. May, G. J. Noa-Guevara, L. J. Russell, G. G. C. Zambrano, D. Izquierdo-Cortazar, I. Steinmacher, M. A. Gerosa, and A. Sarma, "The long road ahead: Ongoing challenges in contributing to large OSS organizations and what to do," *Proc. ACM Hum. Comput. Interact.*, vol. 5, no. CSCW2, pp. 407:1–407:30, 2021.
[93] Z. Zhang, O. Sievi-Korte, U. Virta, H. Järvinen, and D. Taibi, "An investigation on the availability of contribution information in open-source projects," in *47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021, Palermo, Italy, September 1-3, 2021*. IEEE, 2021, pp. 86–90.
[94] O. Elazhary, M. D. Storey, N. A. Ernst, and A. Zaidman, "Do as I do, not as I say: Do contribution guidelines match the github contribution process?" in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 286–290.
[95] C. Brandt and A. Zaidman, "How does this new developer test fit in? A visualization to understand amplified test cases," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022, pp. 17–28.
[96] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE CS, 2012, pp. 11–20.
[97] A. Arcuri, "An experience report on applying software testing academic results in industry: We need usable automated test generation," *Empir. Softw. Eng.*, vol. 23, no. 4, pp. 1959–1981, 2018.
[98] M. Abdi, H. Rocha, S. Demeyer, and A. Bergel, "Small-amp: Test amplification in a dynamically typed language," *Empir. Softw. Eng.*, vol. 27, no. 6, p. 128, 2022.
[99] E. Schoofs, M. Abdi, and S. Demeyer, "AmPyfier: Test amplification in python," *J. Softw. Evol. Process.*, vol. 34, no. 11, 2022.
[100] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software Evolution*. Springer, 2008, pp. 173–202.
[101] B. Marculescu, R. Feldt, R. Torkar, and S. M. Poulding, "Transferring interactive search-based software testing to industry," *J. Syst. Softw.*, vol. 142, pp. 156–170, 2018.
[102] A. Ramírez, J. R. Romero, and C. L. Simons, "A systematic review of interaction in search-based software engineering," *IEEE Trans. Software Eng.*, vol. 45, no. 8, pp. 760–781, 2019.
[103] A. Ramírez, P. Delgado-Pérez, K. J. Valle-Gómez, I. Medina-Bulo, and J. R. Romero, "Interactivity in the generation of test cases with evolutionary computation," in *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2021, pp. 2395–2402.
[104] M. Kamimura and G. C. Murphy, "Towards generating human-oriented summaries of unit test cases," in *IEEE International Conference on Program Comprehension (ICPC)*. IEEE CS, 2013, pp. 215–218.
[105] D. Gaston and J. Clause, "A method for finding missing unit tests," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 92–103.
[106] B. Li, C. Vendome, M. L. Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically documenting unit test cases," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE CS, 2016, pp. 341–352.

Carolin Brandt Carolin Brandt is a Ph.D. student at the Software Engineering Research Group of the Delft University of Technology. She received her bachelor' degree from the Technical University of Munich and in 2020 completed the Elite Graduate Program Software Engineering at the Technical University of Munich, the University of Augsburg and the Ludwigs-Maximilians-University of Munich. The focus of her research is the interaction of software developers with automated tools that are designed to support their work. Her goal is to embed the developer's expertise into automatic test generation tools to create test cases that the developers can directly use to improve their test suites and the quality of their software.

Ali Khatami Ali Khatami is a PhD candidate in the Software Engineering Research Group at Delft University of Technology, the Netherlands. He received his MSc degree in Software Engineering from Sharif University of Technology, in 2021. Currently, he is part of the TestShift project under the supervision of Andy Zaidman. His research interests lie in the intersection of software quality assurance (QA) practices and software analytics, conducting both quantitative and qualitative research in this area, with a focus on software engineers' awareness of QA within their projects and exploring ways to improve QA in open-source software projects.

Mairieli Wessel Mairieli Wessel is an assistant professor at the Department of Software Science, Radboud University, The Netherlands. In 2021, she received her Ph.D. degree in Computer Science from the University of São Paulo, Brazil. Mairieli's research interests include software engineering and computer-supported cooperative work, focused on software bots and open source development.

Andy Zaidman Andy Zaidman is a full professor in software engineering at Delft University of Technology, The Netherlands. He received the MSc and PhD degrees in Computer Science from the University of Antwerp, Belgium, in 2002 and 2006, respectively. His main research interests include software evolution, program comprehension, mining software repositories, software quality, and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013 he was the laureate of a prestigious Vidi mid-career grant, while in 2019 he received the most prestigious Vici career grant from the Dutch science foundation NWO.