How Does This New Developer Test Fit In? A Visualization to Understand Amplified Test Cases

Carolin Brandt
Delft University of Technology, The Netherlands
c.e.brandt@tudelft.nl

Andy Zaidman
Delft University of Technology, The Netherlands
a.e.zaidman@tudelft.nl

***Abstract*—Developer testing, the practice of software engineers programmatically checking that their own components behave as they expect, has become the norm in today's software projects. With the constantly growing size and complexity of software projects and with the rise of automated test generation tools, *understanding* a test case is becoming more and more important compared to writing test cases from scratch.**

This holds especially in the area of developer-centric test amplification, where a tool automatically generates new test cases to improve a developer-maintained test suite. To investigate how visualization can help developers understand and judge test cases, we present the TESTIMPACTGRAPH, a visualization of the call tree and coverage impact of a JUnit test case proposed for amplification. It empowers the developer to drill down into the behavior of a test case, as well as providing them a clear view on how the proposed test case contributes to the coverage of the overall test suite. In a think-aloud study we investigate which information developers seek from the TESTIMPACTGRAPH, how its features can support them in accessing this information, and observations regarding the coverage impact of test cases. We infer ten actionable recommendations on how developer tests can be visualized to help developers understand their behavior and impact.

***Index Terms*—Software Testing, Test Amplification, Test Review, Test Visualization, Test Understanding**

I. Introduction

Developer tests—xUnit test programs which developers use to check the behavior of their code [1]—have become a cornerstone in assuring the quality of today's software systems [2]. As test suites are growing in number and size, understanding test cases one has not written themselves is becoming more and more important, for example, (a) when trying to understand a failing test [3], (b) when using developer tests as a form of executable documentation [4]–[6], (c) when test cases are submitted for code review [7], or (d) when determining whether to add an automatically generated test case to the test suite, e.g., checking whether the captured behavior is correct [8]–[10].

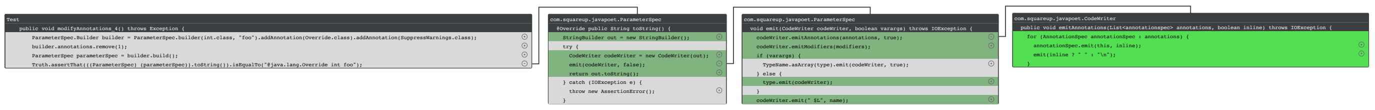
Point (d) is especially important in the area of *developer-centric test amplification* [10]. Test amplification is the process of improving an existing test suite with the help of automated tooling [11], in our case automatically generating new test cases that strengthen a manually written test suite. In *developer-centric* test amplification the goal is to partially relieve the developer's effort in writing test cases by generating ones that the developer subsequently takes over into their maintained test suite [10]. In this process, it is important that the developer understands the new test cases, and subsequently accepts or rejects them based on their understanding of the added value. We want to illustrate this with an example:

> Sara is a software developer who wants to improve her test suite with the help of an automated tool. The tool that she uses generates a few new test cases that supposedly improve the coverage of her test suite. Next, Sara browses through these test cases to determine whether they make sense and test behavior that is correct and relevant for the software under test. She does not only want to understand what the test case does, but also how it improves her current test suite. Even though the tool tells her in which lines new instructions are covered, she has to click and search through the called methods one by one to understand how the test case reaches these new instructions. Sara wishes that there was an easier, less time-intense way to understand the test cases.

To help developers such as Sara explore, understand, and judge test cases that amplify an existing test suite, this paper presents the TESTIMPACTGRAPH. It enables developers to drill down into the methods called by a test case without having to jump from file to file and risk loosing their mental context. A clear indication of where the test case contributes additional code coverage, helps the user judge whether including the test case improves their test suite.

With the help of the TESTIMPACTGRAPH we conduct a think-aloud [12] study to investigate what software developers expect from such a visualization of developer tests. In this paper, we present our results, focusing on the information developers seek from the TESTIMPACTGRAPH, features that help developers access this information and observations related to test coverage that arise from inspecting a test case through the TESTIMPACTGRAPH. We discuss how the TESTIMPACTGRAPH could be applied in further scenarios, like inspecting a proposed test from a pull request, and give ten actionable recommendations on how tools should visualize the behavior and impact of developer tests to aid the software developers exploring and understanding them.

This work was sponsored by the Dutch science foundation NWO through the Vici "TestShift" project (No. VI.C.182.032).



```
Test
  public void modifyAnnotations_4() throws Exception {
    ParameterSpec.Builder builder = ParameterSpec.builder(int.class, "foo").addAnnotation(Override.class).addAnnotation(SuppressWarnings.class);
    builder.annotations.remove(1);
    ParameterSpec parameterSpec = builder.build();
    Truth.assertThat(((ParameterSpec) (parameterSpec)).toString()).isEqualTo("@java.lang.Override int foo");
```

```
com.squareup.javapoet.ParameterSpec
  @Override public String toString() {
    StringBuilder out = new StringBuilder();
    try {
      CodeWriter codeWriter = new CodeWriter(out);
      emit(codeWriter, false);
      return out.toString();
    } catch (IOException e) {
      throw new AssertionError();
    }
```

```
com.squareup.javapoet.ParameterSpec
  void emit(CodeWriter codeWriter, boolean varargs) throws IOException {
    codeWriter.emitAnnotations(annotations, true);
    codeWriter.emitModifiers(modifiers);
    if (varargs) {
      TypeName.asArray(type).emit(codeWriter, true);
    } else {
      type.emit(codeWriter);
    }
    codeWriter.emit(" $L", name);
```

```
com.squareup.javapoet.CodeWriter
  public void emitAnnotations(List<annotationspec> annotations, boolean inline) throws IOException {
    for (AnnotationSpec annotationSpec : annotations) {
      annotationSpec.emit(this, inline);
      emit(inline ? " " : "\n");
    }
```

Fig. 1. An example TESTIMPACTGRAPH, visualizing the second test case in our study.

II. DEVELOPER-CENTRIC TEST AMPLIFICATION

Our work builds upon Brandt and Zaidman's proposal of *developer-centric test amplification* [10]. To generate test cases that are accepted by developers into their manually maintained test suite, they adapted Danglot et. al.'s [13] test amplification approach to produce simple, focused new test cases that improve the instruction coverage of a test suite. In addition, they prototyped a *test exploration tool* which facilitates the interaction between the developer and the automatic generation tool. The process of using a test exploration tool is illustrated in Figure 2. Brandt and Zaidman conducted semi-structured interviews to uncover what factors are important for their approach and the test exploration tool to be successful. They especially focused which information the developers looked for when judging whether it is worth to accept a test case.

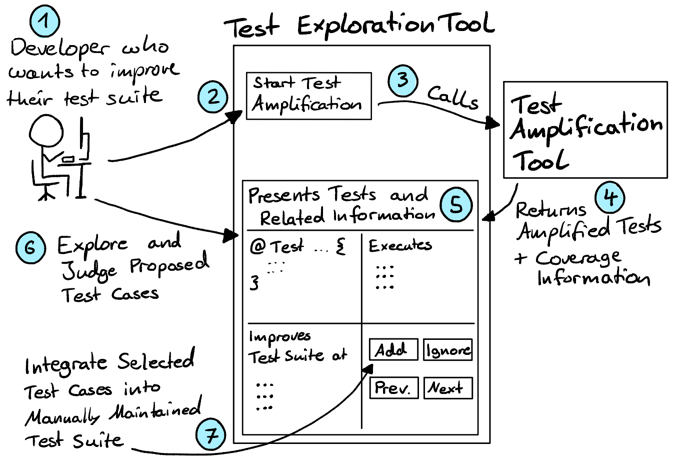

Test Exploration Tool
1
Developer who wants to improve their test suite
2
Start Test Amplification
3
Calls
Test Amplification Tool
4
Returns Amplified Tests + Coverage Information
Presents Tests and Related Information
5
@Test … {
}
Executes
Improves Test Suite at
Add
Ignore
Prev.
Next
6
Explore and Judge Proposed Test Cases
Integrate Selected Test Cases into Manually Maintained Test Suite
7

Fig. 2. An overview of using a test exploration tool [10].

Two key concerns for the study participants were the behavior and intent of the test case: What does it (aim to) test? Likewise, they wanted to know the test case's impact on the coverage of the test suite. During the interviews, the authors observed that the participants used the newly covered lines to infer the intent of the test case: The instructions that only the new test case covers must be what the test case is testing. In some cases, the methods with new coverage were not called directly by the test case, but only indirectly through changes in the input to other methods. The developers struggled to connect these test cases to the coverage impact they provided.

To augment this central interaction in developer-centric test amplification, we aim to develop a visualization that supports developers when inspecting the behavior and coverage of an amplified test case. The goal of this visualization is to:

- connect a test case to the methods it is executing,
- present which parts of the code are covered only by this test case, and with that
- effectively let the developer understand the behavior, intent and coverage of the test case.

Overall, the visualization could be part of Brandt and Zaidman's test exploration tool, serving as one of several components that help the developers browse and judge amplified test cases to decide which ones to take over into their test suite.

III. THE TEST IMPACT GRAPH

In this section, we present the design of the TESTIMPACTGRAPH, a visualization that supports developers in understanding the behavior and the coverage impact of a developer test. The graph consists of *nodes*, which represent the test case and the methods under test, as well as *edges* which represent method calls. The *default layout* helps developers directly focus on the additional coverage a test provides, while the *interactivity* lets them explore the behavior of the test case. Figure 1 shows an example of a TESTIMPACTGRAPH.

A. Method Nodes

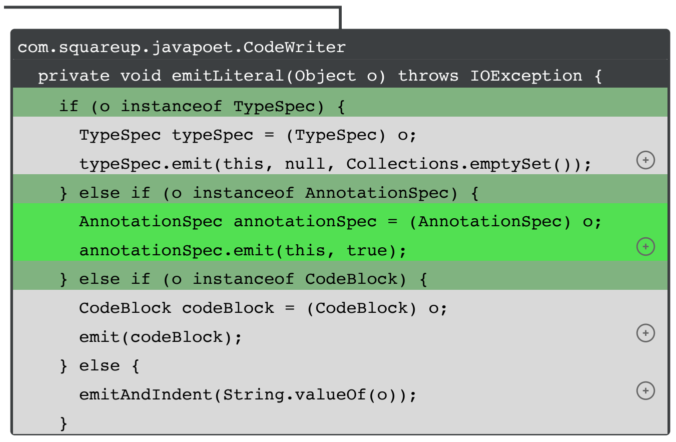
Each method, including the developer test which is visualized by the TESTIMPACTGRAPH, is presented as a node. A node consists of the fully qualified class name, the signature of the method and its source code. Figure 3 shows an example of a node presenting a method under test. The background of each source code line is colored depending on its coverage:

- **grey:** Not covered by this test case or belongs to the test code.
- **dark green:** Covered by this test case and already by another test case.
- **bright green:** Contains instructions only covered by this test case, which we call *additional coverage*.

With the term **additional coverage** we refer to those code elements that are covered by the inspected test case, but not by the other tests in the test suite.

B. Call Edges

For each line with one or more method calls, the TESTIMPACTGRAPH shows a small plus marker at the end. When the user clicks on this marker, they expand the edges connected to that line of code, showing all method nodes called by that line. The marker transforms into a minus icon, which lets the user collapse this part of the call tree again.



```
com.squareup.javapoet.CodeWriter
  private void emitLiteral(Object o) throws IOException {
    if (o instanceof TypeSpec) {
      TypeSpec typeSpec = (TypeSpec) o;
      typeSpec.emit(this, null, Collections.emptySet());
    } else if (o instanceof AnnotationSpec) {
      AnnotationSpec annotationSpec = (AnnotationSpec) o;
      annotationSpec.emit(this, true);
    } else if (o instanceof CodeBlock) {
      CodeBlock codeBlock = (CodeBlock) o;
      emit(codeBlock);
    } else {
      emitAndIndent(String.valueOf(o));
    }
```

Fig. 3. A node in the TESTIMPACTGRAPH.

Apart from opening and closing call edges, the user can freely rearrange nodes, as well as drag and zoom the canvas to explore the TESTIMPACTGRAPH. Initially, the nodes are presented in a hierarchical tree layout from left to right, with the inspected developer test as the root on the left.

C. Default Layout

As it is common for developer tests to execute a not-small number of methods [14], [15], the visualization we describe up until now can get quite large—and therefore overwhelming for the user. To clarify how the inspected test case improves the existing test suite, we want to let the developer focus on what distinguishes the test case from the rest of the test suite. We use *additional coverage* for this. By default, the TESTIMPACTGRAPH shows all methods under test that contain instructions that are covered by the inspected test case, but are not covered by the rest of the test suite. To give context on how these methods are called by the developer test, we also show all the method nodes on the call chains from the developer test to the methods with additional coverage. Figure 1 shows and example of this: the methods leading to the additional coverage are visible, while all other edges are collapsed.

D. Design Rationale

The design of the TESTIMPACTGRAPH is based on various well-established visualization metaphors. To ease the adoption by software developers, we apply as many familiar visual components as possible and strive for a simple design that can fit within an IDE environment.

From related works, we know that developers who inspect a test case during code review are interested in the code under test [7]. They rely on source code to understand the system under test [16], [17] and should be supported while building up the mental context between test code and code under test [18]. This is why we present the developer test alongside the code under test.

We choose to directly show source code to the user, as this provides the highest code proximity [19]. In their interviews, Brandt and Zaidman observed the developers navigating through the code using "jump-to-definition", a common strategy during code comprehension [20]–[23]. To let the developers keep the mental context of the methods they viewed, we show methods as rectangular nodes on a plane and use arrows to show calling relationships, similar to the Code Bubbles metaphor [24]. Just as Bragdon et al. observed with Code Bubbles, we want to support developers in *"understanding a call graph encompassing a handful of functions"* [25].

To visualize code coverage in an intuitively understandable way, we use the established notion of lines highlighted in green [26]. As many developer tests execute several methods [14], [15], presenting this large amount of information at once could be overwhelming for the user [19]. This is why we provide a default view focused on the most relevant methods—the ones with new coverage—and include interactive features [17], [27], letting the user zoom and pan to get an overview or a detailed look on items of interest. Markers to open and close branches indicate options for further exploration [19], [28], enabling the developer to access more details on demand or to filter out uninteresting elements.

E. Implementation

We implemented the TESTIMPACTGRAPH as an extension to the TestCube plugin[1], which generates JUnit tests with the help of the test amplification tool DSpot[2]. We collect the method calls to build the TESTIMPACTGRAPH based on a static analysis, using the IntelliJ PSI support[3]. The coverage information is provided by Jacoco[4] and obtained by DSpot during the test generation. The visualization itself is implemented with the G6 framework by antv[5] and can be found on GitHub[6] together with the exemplary graphs we used for our evaluation.

IV. THINK-ALOUD STUDY

To understand the current state of the TESTIMPACTGRAPH and collect feedback on what to improve and develop further to effectively help developers explore and understand test cases, we perform a preliminary think-aloud study. We aim to answer which information developers seek while they explore a proposed test case and judge whether it improves their current test suite (**RQ1**). Further, we want to know which existing and potential future features of the TESTIMPACTGRAPH help developers effectively and efficiently access the information they are seeking (**RQ2**). Finally, we conjecture that the novel view TESTIMPACTGRAPH provides at developer tests, will raise observations reflecting on the (additional) coverage of test cases (**RQ3**).

In summary, our preliminary think-aloud study intends to answer the following research questions:

[1]https://github.com/TestShiftProject/test-cube/tree/v1.0.3-tig.1
[2]https://github.com/STAMP-project/dspot
[3]https://plugins.jetbrains.com/docs/intellij/psi.html
[4]https://www.jacoco.org/
[5]https://g6.antv.vision/en
[6]https://github.com/TestShiftProject/test-impact-graph/tree/v0.1.0

RQ1: Which information do developers seek from the TestImpactGraph?
RQ2: Which features of the TestImpactGraph help developers access this information?
RQ3: What observations related to test coverage arise when inspecting a developer test through the TestImpactGraph?

A. Study Design

In our think-aloud study, we invite participants familiar with Java to inspect example test cases with the TestImpactGraph.

During the study, we go through three example test cases with each participant. The examples are amplified test cases which were generated by the test amplification plugin TestCube[7] to improve the test suite of the project `javapoet`[8]. We select this project for our study as we expect our participants to be familiar with its domain: generating Java source files. We aim to broadly explore the capabilities of the TestImpactGraph and therefore select three proposed test cases that show different patterns in terms of their additional coverage:

- The first proposed test case covers additional instructions in several lines, but not all lines, of a method that is directly called from the developer test. Its TestImpactGraph is shown in Figure 4.
- The second proposed test case (Figure 1) covers additional lines in a method that is three calls away from the developer test.
- The third proposed test case has the most complex additional coverage pattern. It covers additional instructions in a directly called method, but also several method calls further away and on more than one branch of the TestImpactGraph. The TestImpactGraph in Figure 5 shows how scattered the additional coverage of our third test case is.

All three of these example test cases can be found on GitHub[9], their TestImpactGraphs can be explored as part of our replication package[10].

Generating meaningful names for automatically generated test cases and the variables used in them is a challenging and actively researched topic [29]–[31]. As test names and variable identifiers play a big role in understanding code [32], we do not want our study to be influenced by the quality of the automatically generated names. Therefore, we choose to simplify the test names to the name of the original test case and a number and to simplify the variable names to lowercase variants of their class.

[7]https://github.com/TestShiftProject/test-cube
[8]https://github.com/square/javapoet
[9]L.68, L.92, and L.199 in https://github.com/lacinoire/javapoet/blob/2cbb3084c15a209d28fc8c5fd7472dd695c22591/src/test/java/com/squareup/javapoet/generated/ParameterSpecTest.java
[10]https://doi.org/10.5281/zenodo.6644723

B. Study Execution

We used convenience sampling to recruit our participants: Four PhD Students from the field of computer science and one industrial software developer, all with two to five years of experience in software development and testing. They were unfamiliar with the example project `javapoet`. Before each session, we asked the participant for informed consent according to the ethics guidelines of our university.

Then, we gave a short introduction about the aim and the features of the TestImpactGraph. The TestImpactGraph is built to help developers understand an amplified test case, and especially the value it adds to the existing test suite, similarly to Sara in Section I. This is why we ask the participants to answer the following task for each test case: *"What scenario is newly covered by this test case?"*.

While they browse through the visualization, we ask them to think aloud about their expectations and experiences. They stopped thinking aloud often during the experiment, as they sunk into understanding the code in front of them. After stimulating them with questions, they provided us with rich insights about the TestImpactGraph.

After the sessions, we analyzed the observer's notes using open and axial coding [33]. The results presented in Sections V, VI and VII are grouped along the resulting axial codes. All codes and groups, as well as which participant mentioned them, can be found in our replication package.

C. General Observations

The participants liked interacting with the TestImpactGraph and appreciated a tool that lets them dig deep into the behavior of one test case. Overall, most of them intuitively understood the different components of our visualization: method nodes, call edges, the coverage highlights, and the default view of all additionally covered lines. One participant reported that such a visualization would let them be more confident in automatically generated code because they could retrace its behavior.

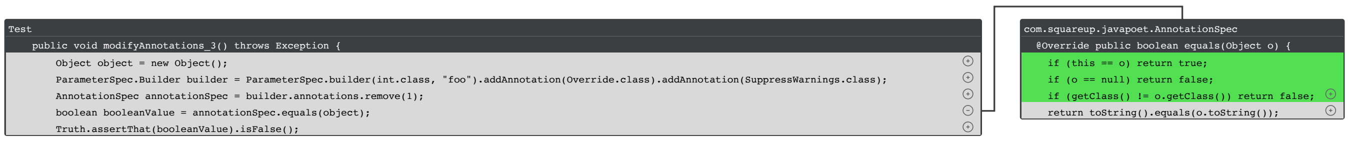
V. RQ1: Which Information Do Developers Seek from the TestImpactGraph?

In this section, we discuss our observations related to the kinds of information our participants sought while inspecting the developer tests to answer which additional scenario is covered by them.

A. What Does It Do? Understanding the Test Case

From observing our participants, we learned that different developers focus on different parts of a test case or its execution when they explore the test case and determine its behavior or its impact:

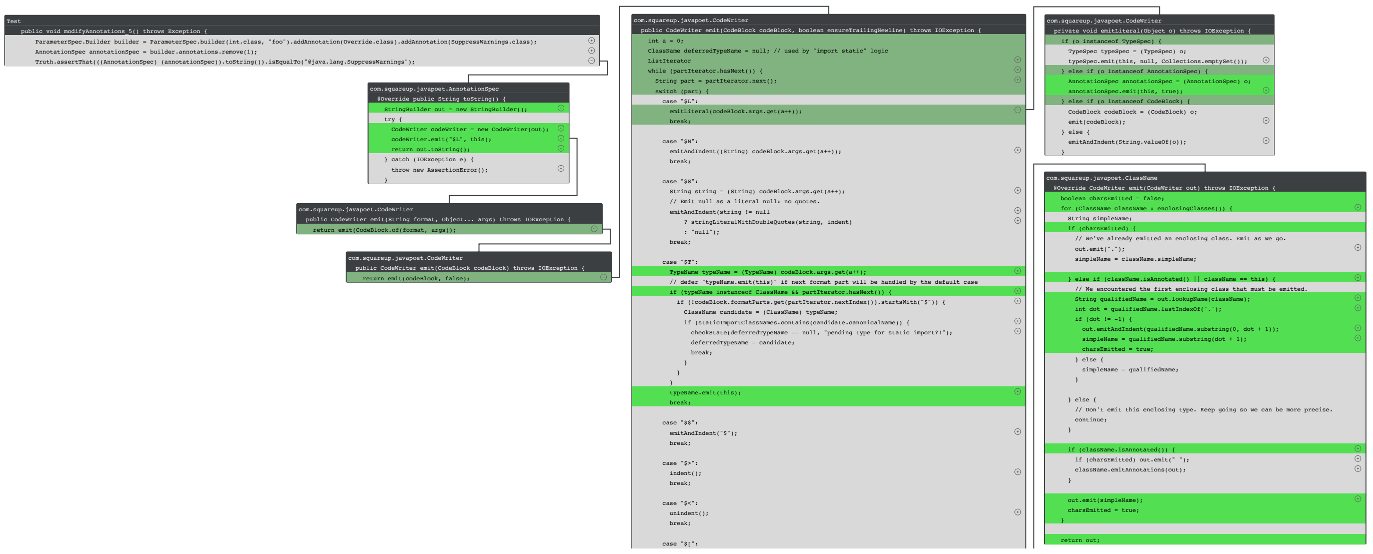
- **Test names:** Several participants based their judgment on the names of the test class or the test case. As we simplified the variable names (see Section IV-A) these could not be used as a source of additional information.
- **Test code:** Several times throughout the study, the participants studied the lines of code of the test case or



```
Test
    public void modifyAnnotations_3() throws Exception {
        Object object = new Object();
        ParameterSpec.Builder builder = ParameterSpec.builder(int.class, "foo").addAnnotation(Override.class).addAnnotation(SuppressWarnings.class);
        AnnotationSpec annotationSpec = builder.annotations.remove(1);
        boolean booleanValue = annotationSpec.equals(object);
        Truth.assertThat(booleanValue).isFalse();
```

```
com.squareup.javapoet.AnnotationSpec
  @Override public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null) return false;
    if (getClass() != o.getClass()) return false;
    return toString().equals(o.toString());
```

Fig. 4. The TESTIMPACTGRAPH for the first test case in our study.



```
Test
    public void modifyAnnotations_5() throws Exception {
        ParameterSpec.Builder builder = ParameterSpec.builder(int.class, "foo").addAnnotation(Override.class).addAnnotation(SuppressWarnings.class);
        AnnotationSpec annotationSpec = builder.annotations.remove(1);
        Truth.assertThat(((AnnotationSpec) (annotationSpec)).toString()).isEqualTo("@java.lang.SuppressWarnings");
```

```
com.squareup.javapoet.AnnotationSpec
  @Override public String toString() {
    StringBuilder out = new StringBuilder();
    try {
      CodeWriter codeWriter = new CodeWriter(out);
      codeWriter.emit("$L", this);
      return out.toString();
    } catch (IOException e) {
      throw new AssertionError();
    }
```

```
com.squareup.javapoet.CodeWriter
  public CodeWriter emit(String format, Object... args) throws IOException {
    return emit(CodeBlock.of(format, args));
```

```
com.squareup.javapoet.CodeWriter
  public CodeWriter emit(CodeBlock codeBlock) throws IOException {
    return emit(codeBlock, false);
```

```
com.squareup.javapoet.CodeWriter
  public CodeWriter emit(CodeBlock codeBlock, boolean ensureTrailingNewline) throws IOException {
    int a = 0;
    ClassName deferredTypeName = null; // used by "import static" logic
    ListIterator
    while (partIterator.hasNext()) {
      String part = partIterator.next();
      switch (part) {
        case "$L":
          emitLiteral(codeBlock.args.get(a++));
          break;

        case "$N":
          emitAndIndent((String) codeBlock.args.get(a++));
          break;

        case "$S":
          String string = (String) codeBlock.args.get(a++);
          // Emit null as a literal null: no quotes.
          emitAndIndent(string != null
              ? stringLiteralWithDoubleQuotes(string, indent)
              : "null");
          break;

        case "$T":
          TypeName typeName = (TypeName) codeBlock.args.get(a++);
          // defer "typeName.emit(this)" if next format part will be handled by the default case
          if (typeName instanceof ClassName && partIterator.hasNext()) {
            if (!codeBlock.formatParts.get(partIterator.nextIndex()).startsWith("$")) {
              ClassName candidate = (ClassName) typeName;
              if (staticImportClassNames.contains(candidate.canonicalName)) {
                checkState(deferredTypeName == null, "pending type for static import?!");
                deferredTypeName = candidate;
                break;
              }
            }
          }
          typeName.emit(this);
          break;

        case "$$":
          emitAndIndent("$");
          break;

        case "$>":
          indent();
          break;

        case "$<":
          unindent();
          break;

        case "$[":
```

```
com.squareup.javapoet.CodeWriter
  private void emitLiteral(Object o) throws IOException {
    if (o instanceof TypeSpec) {
      TypeSpec typeSpec = (TypeSpec) o;
      typeSpec.emit(this, null, Collections.emptySet());
    } else if (o instanceof AnnotationSpec) {
      AnnotationSpec annotationSpec = (AnnotationSpec) o;
      annotationSpec.emit(this, true);
    } else if (o instanceof CodeBlock) {
      CodeBlock codeBlock = (CodeBlock) o;
      emit(codeBlock);
    } else {
      emitAndIndent(String.valueOf(o));
    }
```

```
com.squareup.javapoet.ClassName
  @Override CodeWriter emit(CodeWriter out) throws IOException {
    boolean charsEmitted = false;
    for (ClassName className : enclosingClasses()) {
      String simpleName;
      if (charsEmitted) {
        // We've already emitted an enclosing class. Emit as we go.
        out.emit(".");
        simpleName = className.simpleName;

      } else if (className.isAnnotated() || className == this) {
        // We encountered the first enclosing class that must be emitted.
        String qualifiedName = out.lookupName(className);
        int dot = qualifiedName.lastIndexOf('.');
        if (dot != -1) {
          out.emitAndIndent(qualifiedName.substring(0, dot + 1));
          simpleName = qualifiedName.substring(dot + 1);
          charsEmitted = true;
        } else {
          simpleName = qualifiedName;
        }

      } else {
        // Don't emit this enclosing type. Keep going so we can be more precise.
        continue;
      }

      if (className.isAnnotated()) {
        if (charsEmitted) out.emit(" ");
        className.emitAnnotations(out);
      }

      out.emit(simpleName);
      charsEmitted = true;
    }

    return out;
```

Fig. 5. The scattered coverage highlights of the TESTIMPACTGRAPH for the third test case in our study. To show more within the figure, we rearranged the first four nodes and cropped the fifth node.

the methods under test to understand their behavior or determine the values of test objects. This connects to existing evidence that understanding the setup and input of a test case is central for developer comprehension [34].

- **Method calls:** The participants used the branches of the graph to inspect the behavior of statements that call methods by drilling deeper into the behavior of these methods.
- **Additional coverage:** Through the bright green highlighting and the initial layout showing all lines with additional coverage, some participants focused their analysis on these lines. They analyzed the proposed test case to infer how its statements and the methods they call lead to the execution of the highlighted lines.
- **Inner-method control flow:** Some participants tried to retrace the control flow inside of the methods under test. They made use of the coverage highlights to determine which lines were executed and inferred the outcome of conditional statements.
- **Actual values:** Our participants indicated that access to the actual values of variables and method parameters would help them retrace the behavior of the test execution even better, similar to debugging a test failure [16]. During our study, the participants manually went through the code of the test case and the methods under test to infer the actual values of variables.
- **Natural language explanation:** A participant wished that the tool should give a textual, more abstract description of the new input provided to the method under test, compared to the input other tests provide to the same method, e.g., "Other tests cover this method with an empty list, this test additionally executes it with a non-empty list".

We saw that different developers take different approaches to understand a developer test. A suitable visualization should provide information that supports many of these approaches.

Recommendation 1A

A developer test visualization should **provide access to the wide variety of information** sought by developers **to *understand* the inspected test case**. Under this fall the test name and the source code, the execution flow between and inside of methods, the values of variables and parameters, as well as the coverage and the high-level behavior of the test case.

B. Should I Test This? Provide Scope of Where Code Is From

Surprisingly, many of our participants used the TEST-IMPACTGRAPH in a very similar way to common code coverage tools, a popular means of developers to inspire new test cases [16]. They inspected which lines were covered by the test, but also focused on the lines not covered. Several of them pointed out grey lines, which the assumed to be not covered and stated that they would write additional tests for them.

In this context, it is important that the visualization does not mislead the developers, as the grey lines in the TESTIMPACTGRAPH are simply not covered by the proposed test case.

Recommendation 1B

A developer test visualization that presents coverage information to developers should **be careful when displaying code as not covered**, as developers might use the visualization to identify code to cover in additional test cases.

In addition, some participants asked if certain methods were part of a library or the code of the project itself. They considered it relevant to cover methods of the project itself, but not code that is part of a library.

Similar to this, one participant wished that the test method, i.e., the JUnit method which defines the developer test, as well as any helper methods are visually distinguished from the system code. Their reasoning was that while test helper methods are executed by the test case, they do not need to be covered and would not contribute to describing the additionally tested scenario.

Recommendation 1C

A developer test visualization should **distinguish the *origin* of presented code pieces**: the system under test, dependencies, or the test code. This helps developers judge if a certain piece of code should be tested and if the coverage of this code is relevant for the strength of the test suite.

C. Who Tests This Already? Support Exploring Other Tests

When presented with dark green highlighted lines in the methods under test, i.e., lines that are already covered by other tests in the test suite, several participants wondered *which other test case* was executing these lines already. They wished for functionality to inspect for each line the set of test cases that covers this line, a kind of line-based code-to-test traceability [35].

There were several reasons for the different participants to seek this kind of *reverse coverage* information [36]: looking at and understanding the other test cases which execute the same lines of code could help the users understand the developer test they currently inspect. When the dark green lines appeared in combination with bright green ones, i.e., indicating that the bright green code is only covered by the inspected test case, the developers wondered about the difference between the test case they currently inspect and the ones already covering the dark green lines. They were interested in why the current test case executed the additional instructions, why the other test cases did not. A different consideration was to minimize either the current or the other test cases, to not unnecessarily cover code parts twice. Similarly, Panichella et al. [3] found that developers would want to be notified if a part of the behavior of a generated test case is already checked in a previous method.

Recommendation 1D

A developer test visualization which indicates that code parts are already covered by another test, should **provide information on which *other* tests already cover a line under test**. This can help the developer to take refactoring decisions to improve the focus of their test cases.

VI. RQ2: WHICH FEATURES OF THE TESTIMPACTGRAPH HELP DEVELOPERS ACCESS THIS INFORMATION?

To answer our second research question, we collected observations and feedback from the participants to gauge which existing and potential features help them access the information they seek through the TESTIMPACTGRAPH.

A. Code Nodes: As Close to the IDE as Possible

We received several points of feedback concerning the method nodes in the TESTIMPACTGRAPH. The participants inspected the statements in several method nodes closely, e.g., to reconstruct the behavior of a method or determine the value of an object under test. When presenting source code, the users expected syntax highlighting as it would help them *"spot variables being reused throughout the method"*. The full signatures of the methods under test helped them to reconstruct the behavior of a method and the values returned. As our participants were unfamiliar with the example project from which they inspected test cases, they wished for access to the documentation of several methods they encountered in the TESTIMPACTGRAPH.

Recommendation 2A

Developers expect a **direct presentation of source code** to be **as similar as possible to their familiar IDE environment**. In our study, this included syntax highlighting, the whole signature and source code of the method and access to its documentation.

B. Default View: Provide Confidence to See Everything Relevant

Showing all code with additionally covered lines right away was a big success with the developers that participated in our study. They were glad not to have to search through the graph for more newly covered lines and that they could focus on what is presented at the beginning. After exploring the graph and opening many branches, a participant wished for an easy possibility to return to the default view, so they could refocus on the relevant code paths. Others proposed to highlight the methods leading to additional coverage, so they could be distinguished even while other nodes and branches are opened.

Recommendation 2B

A developer test visualization should help the developer **focus on the central and unique parts of the execution of a test case**. This prevents overwhelming the user with less relevant behavior details, e.g., details in the methods setting up test objects.

C. Where Was I? Providing and Keeping Context

Our participants were thankful for the flat visualization of all methods relevant to the execution of this test case. This let them focus on the methods under test, as well as their connection, better than in an IDE, where they have to switch back and forth between source files to inspect the methods under test.

During our evaluation, we saw that it was important to let the developers maintain their mental context of the nodes visualized on the screen. This includes maintaining the previous layout when new branches are opened and new methods are inspected, even if it required manual zooming and panning from the user to see the new nodes.

Recommendation 2C

The layout of the method nodes in a developer test visualization should **stay consistent while the developer interacts with the visualization**. This lets the user build up and maintain a mental context between the developer test and the code under test.

D. Where Does This Connect? Clarifying Edges

We also learned that the default layout of the graph contributes heavily to the interaction of the developer with the graph. In the case of our TESTIMPACTGRAPH prototype, some call edges were overlapping with each other or partially covered by unrelated method nodes, meaning the participants had to move the nodes around to identify which code line was calling which method node. Similarly, there were cases in which the method nodes overlapped, requiring interaction from the user to make all nodes they found relevant visible. The participants wished for a clearer layout that does not require their interaction, giving them more time to focus on the presented test case.

We discuss this and the previous aspects because they confirm existing results from the field of information visualization. In a study about visualizations for software exploration, Storey et al. [28] name the reduction of user effort in adjusting interfaces as an important design element. Guidelines for UML class diagrams [37], [38] recommend to avoid edge crossings or overlapping nodes, to support users in recognizing the presented objects.

We observed another visualization challenge more specific to the TESTIMPACTGRAPH. As the method calls are connected to a particular line, when multiple methods are called it was difficult to distinguish for a participant which method nodes correspond to which method call. One participant resorted to comparing the names of the methods, noticing that this falls short if two have the same name or one method is called, possibly with different parameter values. One way to address this could be the use of color on the method calls and edges, similar to the "smart step into" feature of IntelliJ[11].

Recommendation 2D

The layout of developer test visualization should **show clearly which method call and method node the ends of an edge connect to** and should not require the interaction of the user to clarify the presented information.

VII. RQ3: WHAT OBSERVATIONS RELATED TO TEST COVERAGE ARISE WHEN INSPECTING A DEVELOPER TEST THROUGH THE TESTIMPACTGRAPH?

Inspecting test cases through the TESTIMPACTGRAPH gives the developer the chance to take a novel look at the behavior and especially the additional coverage of a developer test. We want to report on a few interesting observations our participants made while using the TESTIMPACTGRAPH on the three examples we provided.

A. Should This Not Already Be Unit-Tested? Deep And Accidental Coverage

As shown in Figure 5, our third example shows a rather large TESTIMPACTGRAPH, because some of the additionally covered lines are multiple method calls away from the developer test. Two of our participants noted this and wondered whether these methods should not be covered more directly by a unit test, instead of by the more integration-style test they were inspecting. As there was also a whole new method covered directly through a call from the developer test, one participant even wondered if these further methods were covered "accidentally" by a test meant to test the directly called method. The participant said to not trust this accidental coverage and built their answer to what the test is newly testing solely on the directly called and newly covered method.

A participant also pointed to similar issues if a method would lead to additional coverage in several different, unconnected areas of the code under test. This would give them the impression that it is not clear what the test case is really intending to test.

[11] https://www.jetbrains.com/help/idea/stepping-through-the-program.html#smart-step-into Accessed: June 1st, 2021

Recommendation 3A

According to our participants, **additional coverage that is several method calls away could point to a lack of unit testing or be considered accidental**. Accidental coverage was deemed less relevant to determine the impact of a test case.

B. What Is Executed Here? Instruction Coverage Visualized Per Line

As described before, several of our participants approached understanding the methods under test by going through their statements one by one. Where possible, they made use of the line highlighting that indicates which lines were executed. However, the common practice of visualizing instruction coverage as a highlight of a whole line led to confusion in some cases. In the first example test case (see Figure 4), there are three one-line `if` statements with `return` statements in their bodies. All three lines are highlighted in bright green, i.e., indicating that there were additional instructions covered on these lines. The participants were confused how all of these `return` statements could be executed in one method call.

While there are newly covered instructions on these lines of code—namely the conditions of the `if` statements—the highlight's indication that the whole line is executed was confusing. A participant reflected, that it would be better to indicate that while additional instructions are covered, not all instructions on these lines are covered by the inspected test case. They added that this would also depend on the code style of the code under test, e.g., whether the `if` and `else` blocks of a conditional statement are on separate lines from the conditions.

We made a similar observation in the second example test case (Figure 1), where seemingly a whole new method is covered: all lines in the method are bright green. The method only consists of a `for` loop with statements in it. However, our participants wondered why the statement calling this method was dark green, indicating that it was already executed by another test case. Upon closer inspection, we determined that the previous callers executed the method in question with an empty list, therefore the first instruction of the `for` loop was executed. In the new, inspected test case, the method in question was called with a filled list, leading to the execution of the statements in the `for` loop, as well as the increment statement in the `for` loop's header. A participant noted that it would help to indicate that some of the instructions on the header line are already executed by other test cases.

Recommendation 3B

When instruction coverage is visualized at the line level, it should be indicated whether **instructions on a line were previously covered**, and whether **there are still uncovered instructions on a line**.

VIII. Discussion

Our exploratory think-aloud study resulted in a range of 10 actionable recommendations regarding the information developers seek from the TestImpactGraph, features to help them access this information and observations based on the new view angle on additional coverage of developer tests. Several of them confirm existing knowledge from information visualization (**2A**, **2C**, **2D**), extend existing intuitions for the area of developer test inspection (**1A**, **1C**, **2B**), and others point to novel challenges special to developer test inspection (**1B**, **1D**, **3A**, **3B**). In this section we discuss why our results can also be applied to the area of test code review, due to the large similarities between inspecting a test case for amplification and for test review. Furthermore, we illustrate how the TestImpactGraph can provide insights that lead to a more fine-grained coverage understanding related to test directness and redundancy.

A. Test Review

The review of test code during traditional code review has many parallels to the inspection of proposed test cases during test amplification. In both cases, developers judge whether a new developer test is adequate and should be included into the test suite. Our results also show these parallels to previous work on test review by Spadini et al. [7]. Similar to our observations that developers use the TestImpactGraph as a coverage tool to spot uncovered lines (Section V-B), a central concern in code review is to understand whether the test covers all paths [7]. None of our participants asked whether they could inspect the presented code in their IDE, a typical behavior of developers to gain a complete picture of the code during test review [7]. Therefore, we hypothesize that if test reviews are performed inside the TestImpactGraph tool, Spadini et al.'s recommendation to provide better navigation between the developer test and the code under test would be addressed.

B. Differential Code Coverage

A tool dedicated to surface coverage changes during code review is Codecov[12]. It provides high level, aggregated coverage information about the whole project, but also *differential coverage* introduced by a commit or a pull request. Their differential coverage indicates how much of the code diff is covered an how the change impacts the overall project coverage. In their source code view[13], developers can inspect the code under test enriched with highlights that show how the coverage of single lines change through the inspected commit or pull request. While Codecov provides a coverage diff for any kind of code change, i.e., to the test code and the code under test, the TestImpactGraph focuses on the addition of one test case, while the code under test stays constant. Furthermore, the TestImpactGraph is showing the method call connections between the developer test and the methods

[12]https://about.codecov.io/

[13]https://docs.codecov.io/docs/viewing-source-code

under test, letting the developer retrace the execution of the test case. This addresses an important point in test coverage evolution: helping developers understand better why a change in the code leads to a change in coverage [39].

When inspecting a test case that is proposed to amplify a test suite, or an added test case in traditional code review, a tool like Codecov would give quick feedback on how the test case impacts the coverage of the test suite, while the TESTIMPACTGRAPH would give more detailed insights to the developer on where and why the test case impacts the code coverage.

C. Refined Coverage Insights

From our observations about already covered code (**Recommendation 1D**), as well as deep and accidental coverage (**Recommendation 3A**), we see a chance for the TESTIMPACTGRAPH to give developer's a deeper insight into how their tests cover the code under test. The reflections of the participants show that coverage could be interpreted as more than just covered or not covered. Instead, the participants also considered how directly test cases are covering a specific method. Test directness is important to help developers pinpoint the fault in the code when a test is failing [40]. It also impacts how well a test case can serve as documentation of the code under test [15].

Whether these insights lead developers to adapt their test suite to be more direct, will depend on the needs of the software project and their testing culture, e.g., the relative value of unit and integration tests for the project. Independent from the testing culture of a project, the TESTIMPACTGRAPH can give the developers deeper insight into the coverage that single test cases contribute, enabling them to take informed decisions about the design of their test suites.

D. Relevance of Deep Coverage For Test Descriptions

Current techniques to automatically generate names for unit test cases use, beneath other information, the names of the methods executed by a test case as basis for the generated names [29]. The automatic test generation community is moving towards generating integration tests that check the interaction of multiple classes [41]–[43], and the amplified test cases from our study were also integration test that executed several methods. Based on the feedback we got regarding deep and accidental coverage (Section VII-A), the question becomes whether methods that are executed, but further away from the developer test in the call chain, are relevant to generate test names that are meaningful for developers. Similarly, we should investigate, whether or how deep coverage can be used for natural language test descriptions, such as those generated by Panichella et al. [3] or Roy et al. [30].

E. Threats To Validity

In the following, we discuss several threats to the validity of our results.

With regards to construct validity, the amplified test cases we chose for our study might be favorable for the TESTIMPACTGRAPH. Long, complex test cases that additionally cover a large part of the source code would be difficult to inspect with the current design. Similarly, for simple test cases it might feel unnecessary to use more than the tools available in an IDE. To mitigate this threat, we picked a variety of test cases that represents the types of coverage we see when amplifying test suites with TestCube. All the tools we used[14], as well as our experiment data[15] are openly available and we encourage others to explore them to retrace our observations.

We assume a broad target audience for the TESTIMPACTGRAPH, novices as well as experienced software developers. However, our participants might not match this audience, as we used convenience sampling to select participants. The reflective insights about test directness and accidental coverage might stem from the fact that our participants are mainly PhD students and therefore more aware about software quality than other software developers. Future work is needed to investigate how professional experience impacts developer's interaction with the TESTIMPACTGRAPH.

As we used convenience sampling, the researcher and the participants knew each other personally, however the participants had not been involved in the researcher's work before the study. The participants were motivated to participate to support the author's research work and could have been biased to give more positive feedback during the study. We mitigated this threat by emphasizing the value of critical and opinion-rich comments, and focusing our results on the concrete recommendations rather than ratings of the existing design.

A threat to the external validity of our study might be the underlying test amplification approach we used to generate the test cases for our study. The TestCube tool selects which test cases to propose based on whether they cover additional instructions in the code under test. One could choose other selection criteria, e.g., whether a new edge case is checked, which would call for another kind of visualization, e.g., because no additional instructions are covered. Another threat is the selection of the example project, or the test cases we inspected. We aimed for a middle-sized project and a variety of test cases with respect to their coverage characteristics. This lead to a variety of visualization in the TESTIMPACTGRAPH, and let us widely explore visualizations from an average Java project. We do not claim our findings to be applicable to every software project, and their relative importance will vary depending on the developer interacting with our tool.

IX. RELATED WORK: TEST VISUALIZATIONS

In this section we present related scientific works in the area of visualizing software test cases or associated metrics.

Visualizations are used to judge the quality of a whole test suite, visualizing test metrics, including code coverage, in conjunction with the system under test. Borg et al. [44] visualize historical test outcomes in a code city of the system under tests. Their goal is to help identify error-prone components and potential coverage holes, i.e., components that are not covered

[14]https://github.com/TestShiftProject/test-cube/tree/test-impact-graph, https://github.com/TestShiftProject/test-impact-graph

[15]https://doi.org/10.5281/zenodo.6644723

enough by the test suite. Balogh et at. [45] extend a different code city framework to visualize test-related metrics together with the components of the system under test. Perscheid et al. [46] use a variety of tree maps, presenting different quality aspects of a test suite. They aim to help developers pinpoint untested code and improve time or memory intensive test cases. Opmanis et al. [47] present a dashboard that visualizes the test results of large systems, helping managers and quality engineers to identify the origin of deteriorations or improvements. The TESTQ tool by Breugelmans and Van Rompaey [48] presents a tree-like overview over the structure of a whole test suite. For a closer inspection, singular test cases are presented as a hierarchical structure of the test components like fixture and test helpers, annotated with instances of test smells. A separate window provides an overview over all smells appearing in the test suite and lets the developer spot very smelly tests. While these visualizations aim to give an overview of the whole test suite, the TESTIMPACTGRAPH is geared towards visualizing the execution of a single test case and visualizing it coverage, focusing on the impact it makes on the coverage of the whole test suite.

Other approaches visualize the current coverage of a test suite to aid developers in achieving better coverage. Vanessa Peña Araya [49] proposes *Test Blueprints*, a hierarchical visualization of the structure of the code under test in conjunction with how often its methods are executed by the test suite. Lawrance et al. [26] highlight lines in the code under test to indicate the existing quality of a test suite and investigate whether this leads developers to create more effective tests. With a similar goal, Rahmani et al. [50] create a control flow graph showing the current branch coverage. Among our participants, we observed a similar, intuitive strive to write additional test cases for lines that were not marked as covered.

Previous work investigated visualizing the software components, methods or lines that are executed by a test case or an interaction with the system to aid understanding of the behavior of the system under test. Cornelissen et al. [51] visualize tests as abstracted scenario diagrams, Arthur-Jozsef Molnar [52] visualizes the components executed by a sequence of GUI interactions, and Gestwicki and Jayaraman [53] present the structure of live object and values as well as method invocations during the execution of a java program.

The research area of test-to-code traceability is similar to our effort to connect a developer test to the code it tests. The IDE extension EZUNIT by Bouillon et al. [54] creates links to jump from a test case to the methods under test. Aljawabrah et al. [55], [56] visualize the connection between unit tests and the code under test with the TCTRACVIS tool. Their tools shows traceability links in a hierarchical tree graph in both directions: from test to code and from code to test. While TCTRACVIS is built to support different methods to retrieve traceability links and visualizes the connection of high-level code structures like classes and methods, the TESTIMPACTGRAPH relies purely on coverage information and visualizes the connection of test and code under test on a line granularity. Furthermore, the TESTIMPACTGRAPH presents the source code directly to the developers, keeping the visualization as close as possible to the source code [19].

Vidaure et al. [57] use visualizations to shed light on the internal processes of automated test generation. Their tool TestEvoViz lets developers examine the process behind a genetic algorithm to see the impact their configuration has on the test generation. While the TESTIMPACTGRAPH also works with automatically generated test cases, its design is in principle independent from the specific test generation approach. The TESTIMPACTGRAPH steps in after the test cases are generated, with the aim to help the developer inspect one specific new test case at a time.

X. CONCLUSION

In this paper we present the TESTIMPACTGRAPH, a visualization of developer tests to aid understanding of their behavior and impact. Through an exploratory think-aloud study we identify a range of ten recommendations for future visualizations of developer tests.

We discuss how the TESTIMPACTGRAPH can give developers a better context of the code under test during code review, how it complements differential coverage and how developers can use it to gain a more fine grained understanding of the coverage their test cases provide. In future work we aim to apply the recommendations we presented to the TESTIMPACTGRAPH itself and study in-depth how much it helps developers explore automatically generated test cases, as well as test cases written by their colleagues. In addition, we intend to compare how understanding test cases with the specialized TESTIMPACTGRAPH compares to using standard debuggers, IDE features like *jump-to-definition*, or general-purpose visualizations of the tests' call graphs. We want to investigate other means to convey the behavior and impact of a test case, e.g., through test names or textual descriptions, leveraging our observations about deep and accidental coverage. Further, we will explore ways to help developers understand test cases that improve the test suite in other ways than instruction coverage, such as covering edge cases, reproducing known bugs or killing artificial mutants of the system under test.

In short, this paper contributes:

- The design and a prototypical implementation of the TESTIMPACTGRAPH, a visualization to help developers understand the behavior and impact of a test case generated to amplify an existing test suite.
- Ten actionable recommendations on how tools with a similar aim should visualize developer tests.

Our vision is that tools like the TESTIMPACTGRAPH will enable developers to dive deep into new test cases and help them understand how and why these test cases amplify the power of their test suite.

REFERENCES

[1] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Pearson Education, 2007.

[2] 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.

[3] 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 *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. Williams, Eds. ACM, 2016, pp. 547–558.

[4] D. Hoffman and P. Strooper, "API documentation with executable examples," *Journal of Systems and Software*, vol. 66, no. 2, pp. 143–156, 2003.

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

[6] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, H. Sharp and M. Whalen, Eds. IEEE / ACM, 2019, pp. 61–70.

[7] 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 *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 677–687.

[8] G. Fraser and A. Arcuri, "EvoSuite: On the challenges of test case generation in the real world," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 2013, pp. 362–369.

[9] 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 *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 2017, pp. 263–272.

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

[11] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[12] K. A. Ericsson and H. A. Simon, "How to study thinking in everyday life: Contrasting think-aloud protocols with descriptions and explanations of thinking," *Mind, Culture, and Activity*, vol. 5, no. 3, pp. 178–186, 1998.

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

[14] F. Trautsch, S. Herbold, and J. Grabowski, "Are unit and integration test definitions still valid for modern java projects? an empirical study on open-source projects," *J. Syst. Softw.*, vol. 159, 2020.

[15] J. Van Geet and A. Zaidman, "A lightweight approach to determining the adequacy of tests as documentation," *Proc. PCODA*, vol. 6, pp. 21–26, 2006.

[16] M. F. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Transactions on Software Engineering*, -.

[17] P. K. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, and P. Tulula, "Facilitating the comprehension of c-programs: An experimental study," in *IEEE Second Workshop on Program Comprehension, WPC 1993, Capri, Italy, 8-9 July 1993*. IEEE, 1993, pp. 55–63.

[18] M. P. Prado and A. M. R. Vincenzi, "Towards cognitive support for unit testing: A qualitative study with practitioners," *J. Syst. Softw.*, vol. 141, pp. 66–84, 2018.

[19] H. M. Kienle and H. A. Müller, "Requirements of software visualization tools: A literature survey," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2007, Banff, Alberta, Canada, June 25-26, 2007*, J. I. Maletic, A. C. Telea, and A. Marcus, Eds. IEEE Computer Society, 2007, pp. 2–9.

[20] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the elipse IDE?" *IEEE software*, vol. 23, no. 4, pp. 76–83, 2006.

[21] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, no. 12, pp. 971–987, 2006.

[22] M. Desmond, M. D. Storey, and C. Exton, "Fluid source code views," in *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*. IEEE Computer Society, 2006, pp. 260–263.

[23] T. Karrer, J. Krämer, J. Diehl, B. Hartmann, and J. O. Borchers, "Stacksplorer: call graph navigation helps increasing code maintenance efficiency," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, Santa Barbara, CA, USA, October 16-19, 2011*, J. S. Pierce, M. Agrawala, and S. R. Klemmer, Eds. ACM, 2011, pp. 217–224.

[24] A. Bragdon, R. C. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr., "Code bubbles: A working set-based interface for code understanding and maintenance," in *Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, Georgia, USA, April 10-15, 2010*, E. D. Mynatt, D. Schoner, G. Fitzpatrick, S. E. Hudson, W. K. Edwards, and T. Rodden, Eds. ACM, 2010, pp. 2503–2512.

[25] A. Bragdon, S. P. Reiss, R. C. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr., "Code bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 455–464.

[26] J. Lawrance, S. Clarke, M. Burnett, and G. Rothermel, "How well do professional developers test with code coverage visualizations? an empirical study," in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 2005, pp. 53–60.

[27] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Proceedings 1996 IEEE symposium on visual languages*. IEEE, 1996, pp. 336–343.

[28] M. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *J. Syst. Softw.*, vol. 44, no. 3, pp. 171–185, 1999.

[29] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 57–67.

[30] 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 *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 287–298.

[31] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. ACM, 2016, pp. 625–636.

[32] F. Salviulo and G. Scanniello, "Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals," in *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014*, M. J. Shepperd, T. Hall, and I. Myrtveit, Eds. ACM, 2014, pp. 48:1–48:10.

[33] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[34] C. S. Yu, C. Treude, and M. F. Aniche, "Comprehending test code: An empirical study," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 501–512.

[35] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009*, A. Winter, R. Ferenc, and J. Knodel, Eds. IEEE Computer Society, 2009, pp. 209–218.

[36] V. Hurdugaci and A. Zaidman, “Aiding software developers to maintain developer tests,” in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, T. Mens, A. Cleve, and R. Ferenc, Eds. IEEE Computer Society, 2012, pp. 11–20.

[37] D. Sun and K. Wong, “On evaluating the layout of UML class diagrams for program comprehension,” in *13th International Workshop on Program Comprehension (IWPC 2005), 15-16 May 2005, St. Louis, MO, USA*. IEEE Computer Society, 2005, pp. 317–326.

[38] C. Bennett, J. Ryall, L. Spalteholz, and A. Gooch, “The aesthetics of graph visualization,” in *3rd International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging, CAe 2007, Banff, AB, Canada, June 20-22, 2007, Proceedings*, D. W. Cunningham, G. W. Meyer, L. Neumann, A. Dunning, and R. Paricio, Eds. Eurographics Association, 2007, pp. 57–64.

[39] M. Hilton, J. Bell, and D. Marinov, “A large-scale study of test coverage evolution,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 53–63.

[40] 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.

[41] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, “Presentation abstract: Generating class integration tests using call site information,” in *18th Belgium-Netherlands Software Evolution Workshop (BENEVOL’19), Brussels*, 2019.

[42] M. Grechanik and G. Devanla, “Generating integration tests automatically using frequent patterns of method execution sequences,” in *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019*, A. Perkusich, Ed. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019, pp. 209–280.

[43] M. Pezzè, K. Rubinov, and J. Wuttke, “Generating effective integration test cases from unit ones,” in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 2013, pp. 11–20.

[44] M. Borg, A. Brytting, and D. Hansson, “An analytical view of test results using cityscapes,” in *Proc. of the Design and Verification Conf. and Exhibition United States (DVCON US)*, 2018.

[45] G. Balogh, T. Gergely, Á. Beszédes, and T. Gyimóthy, “Using the city metaphor for visualizing test-related metrics,” in *First International Workshop on Validating Software Tests, VST@SANER 2016, Osaka, Japan, March 15, 2016*. IEEE Computer Society, 2016, pp. 17–20.

[46] M. Perscheid, D. Cassou, and R. Hirschfeld, “Test quality feedback improving effectivity and efficiency of unit testing,” in *2012 10th*

[53] P. V. Gestwicki and B. Jayaraman, “Interactive visualization of java programs,” in *2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2002), 3-6*

International Conference on Creating, Connecting and Collaborating through Computing. IEEE, 2012, pp. 60–67.

[47] R. Opmanis, P. Kikusts, and M. Opmanis, “Visualization of large-scale application testing results,” *Baltic Journal of Modern Computing*, vol. 4, no. 1, p. 34, 2016.

[48] M. Breugelmans and B. Van Rompaey, “TestQ: Exploring structural and maintenance characteristics of unit test suites,” in *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer, 2008.

[49] V. P. Araya, “Test blueprint: An effective visual support for test coverage,” in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 1140–1142.

[50] A. Rahmani, J. L. Min, and A. Maspupah, “An evaluation of code coverage adequacy in automatic testing using control flow graph visualization,” in *2020 IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. IEEE, 2020, pp. 239–244.

[51] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman, “Visualizing testsuites to aid in software understanding,” in *11th European Conference on Software Maintenance and Reengineering (CSMR’07)*. IEEE, 2007, pp. 213–222.

[52] A. Molnar, “Live visualization of GUI application code coverage with GUITracer,” *CoRR*, vol. abs/1702.08013, 2017.
September 2002, Arlington, VA, USA. IEEE Computer Society, 2002, pp. 226–235.

[54] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, “EzUnit: A framework for associating failed unit tests with potential programming errors,” in *Agile Processes in Software Engineering and Extreme Programming, 8th International Conference, XP 2007, Como, Italy, June 18-22, 2007, Proceedings*, ser. Lecture Notes in Computer Science, G. Concas, E. Damiani, M. Scotto, and G. Succi, Eds., vol. 4536. Springer, 2007, pp. 101–104.

[55] N. Aljawabrah, T. Gergely, S. Misra, and L. F. Sanz, “Automated recovery and visualization of test-to-code traceability (TCT) links: An evaluation,” *IEEE Access*, vol. 9, pp. 40 111–40 123, 2021.

[56] N. Aljawabrah, T. Gergely, and M. Kharabsheh, “Understanding test-to-code traceability links: The need for a better visualizing model,” in *Computational Science and Its Applications - ICCSA 2019 - 19th International Conference, Saint Petersburg, Russia, July 1-4, 2019, Proceedings, Part IV*, ser. Lecture Notes in Computer Science, S. Misra, O. Gervasi, B. Murgante, E. N. Stankova, V. Korkhov, C. M. Torre, A. M. A. C. Rocha, D. Taniar, B. O. Apduhan, and E. Tarantino, Eds., vol. 11622. Springer, 2019, pp. 428–441.

[57] A. C. Vidaure, E. C. Lopez, J. P. S. Alcocer, and A. Bergel, “TestEvoViz: Visual introspection for genetically-based test coverage evolution,” in *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE Computer Society, 2020, pp. 1–11.