Global Overviews of Granular Test Coverage with Matrix Visualizations

Kaj Dreef
University of California, Irvine
kdreef@uci.edu

Vijay Krishna Palepu†
Microsoft
Vijay.Palepu@microsoft.com

James A. Jones
University of California, Irvine
jajones@uci.edu

***Abstract*—Existing IDE-based tools that are available to developers make understanding software testing difficult for a software system, for both granular tasks (*e.g.,* answering questions such as, "which test cases execute this method?") and global tasks (*e.g.,* answering questions such as, "what is the proportion of unit tests to system tests?"). IDE-based tools typically support local, file-based views of a project's test suite, and rarely offer a global overview. Global overviews can provide a larger context for a method's execution by test cases; help identify other similar, or related methods; and even reveal similarity between individual tests. This work approaches such challenges with a novel, interactive, matrix-based visual interface that provides a global overview of a software project's test suite, specifically in the context of the methods available in the project's codebase. Through a series of interactive functions to sort, filter, query, and explore a test-matrix visualization, we demonstrate how developers can effectively answer questions about their project's test suite, and the code executed by such tests. Our evaluations, performed on four real-world software systems, show that the interactive visualization assisted developers to answer questions about software tests and the code they execute. Further, the visualization consistently outperforms traditional development tools, both in accuracy and time taken to complete software-engineering tasks.**

***Index Terms*—Software Testing, Visualization, Software Comprehension, Software Test Comprehension**

I. Introduction

Consider some questions that engineers routinely ask of their test suite: "What code is tested?"; "Which components are untested or under-tested?"; "Does a test focus on specific methods, or the system as a whole?"; "Which methods do the failing tests execute?"; and "Are two tests executing the same methods, or testing the same/similar functionality?" Such questions and needs by developers to understand their test suites are validated by several prior studies (*e.g.,* [1], [2], [3], [4], [5], [6]). Such questions speak to: (a) how the tests themselves are organized, *i.e.,* the "form" of the test suite, and (b) the specific components and aspects of the product that the tests are designed to execute and verify, *i.e.,* the "function" of the tests.

Questions about the "form" of a test suite often require a global, or overarching, understanding of the entire suite. These questions help identify the existing or potential organization of a test suite. The tests may be organized in a number of ways. For example, tests can be organized by results: passing and failing. Or, tests can be organized into clusters, where each cluster represents a product behavior or functionality to be verified. And, perhaps a more common organization found in real-world test suites is to differentiate unit, integration, and system tests. In such cases, tests are organized into components that mirror the packages and methods that are directly verified by their respective tests. Organizing tests by different criteria may reveal different aspects of the suite as a whole, and help engineers navigate and understand their test suites. For instance, when organizing tests as unit/integration/system tests, an engineer can easily identify the batch of tests that directly test a method that the engineer is trying to refactor or modify. Similarly, when an engineering manager is trying to assess a product's testing effort, it can be useful to organize the tests by the tests' coverage (increasing to decreasing), or cluster them by the product behaviors they are trying to verify.

Unlike global overviews, the "function" of tests require a more localized consideration. A test's "function" pertains to questions about a desired behavior that the test is verifying, or the multiple methods and lines that it executes. Such questions help in evaluating the efficacy of a specific test. Conversely, engineers may also want to understand if, or how, a specific method is executed by multiple tests (perhaps to be selective about which subset of tests they want to re-run).

Indeed, questions about the global *form* of an entire test suite, and the localized *function* of individual tests are related, often inextricably. For instance, when asking, "which tests are executing my code?" it can be useful to know if those are unit, integration, or system tests; or which behavior such tests are verifying. Similarly, organizing an entire test suite by code coverage may actually highlight a specific method or component that remains entirely untested. As such, revealing the global form and the localized function of software tests in a unified way may reveal qualities that are beyond each individually.

In this work, we visualize software test-execution data in an interactive matrix visualization that we call Morpheus (see Figure 1) to support developer understanding, querying, discovery, and exploration of their test suites. Morpheus presents engineers with global overviews of their test suites in a visualization that captures granular data about a test suite's execution, to reveal patterns in the test suites and their executions. To seamlessly explore localized views of specific

†The opinions expressed in this publication are those of the author. They do not purport to reflect the opinions or views of Microsoft.

tests within global overviews, MORPHEUS enables user-driven interactions. Such interactions allow engineers to quickly filter test data in the visualization to a subset of tests and methods within a software project. Engineers would also be able to re-organize the data (tests and methods) to reveal patterns, both in local and global views.

The main contributions of this work are:

1) A novel application of the matrix-styled representation for presenting granular test execution data for a software project's test suite in global overviews;
2) A series of interaction capabilities, atop the matrix-based global overviews, to seamlessly explore a software test suite and answer questions about specific software tests and methods;
3) An open-sourced implementation of MORPHEUS and interactive demo [7] that supports Java programs and test suites written using JUnit or TestNG.
4) An evaluation of MORPHEUS when answering questions about test suites in real-world software projects, along with a replication package [7].

II. MOTIVATING EXAMPLES

Prior research studies have found that developers want better tools to understand, write, and query their test suites. Torkar and Mankefors [6] conducted a survey of 225 software developers and found, "One thing was consistent with all developers. They all wanted better tools for writing tests cases, especially when the code had been written by someone else." and "They, simply put, wanted some statistics on how well their tests were written and how well they tested a given function, class or code snippet, i.e. code coverage." Rafi *et al.* [2] also conducted a study of software engineers that elicited 115 responses. One of their findings was that "Test automation needs at least as much maintenance as the developed software with regards to the Technical Debt." And, several other research studies have found that developers have needs for understanding and answering questions about their test suites (*e.g.,* [1], [3], [4], [5]).

As a concrete example, consider the following question that a developer may ask about their test suite: *"what test cases execute a specific method, either directly or indirectly?"* The point of this question is to determine exactly which test cases execute a given method, whether the method is called directly from a test case, or it is called indirectly, by calling some chain of other methods, which then calls our given method. This is not an unreasonable or far-fetched question: we may want to determine the degree to which a given method has been tested. Simple coverage tools can answer whether or not a method was executed, but not by which (or how many) test cases. A profiling tool may answer how many times the method was invoked, but does not distinguish between "invoked N times in a loop by a single method" versus "invoked once by N methods." And, searching the test code will only reveal the test cases that called the method directly.

As an informal feasibility study, we asked three developers with 9, 12, and 24 years of Java-development experience this question. All three developers agreed that this was a useful question and expressed surprise that such a question is so simple and that it was not immediately obvious how to answer it. Before we reveal their answers, perhaps you, dear reader, can attempt to think of how you would solve this problem, using your own development tools. Here are the ideas that they came up with, often after hours of reflection and brainstorming:

1) Put print statements at the beginning of all test cases that print the name of the test case, and also put a print statement in the beginning of the specified method. Recompile, run the test suite, and log the output. Then, we can search for the specified method's print-statement output and correlate all matches with their preceding test case's print-statement output.
2) Put an artificial bug that forces a crash/uncaught exception at the beginning of the specified method, recompile, run the test suite, and then witness which test cases now fail due to the new artificial bug.
3) Put a breakpoint at the specified method, and step-and-continue, to look up the stack trace, one-by-one, for each test case for which the breakpoint is tripped.

As we can see from these possible solutions, this question *is* answerable with current tools, but it is far from obvious how to go about answering it. The current state of development tools around understanding such simple questions about the test suite is primitive and relies upon such tricks and cleverness.

Now, consider more difficult questions, such as "do most test cases execute the same or similar subsets of methods or components?" or "are my test cases primarily small, mostly single-method unit test cases, medium-sized integration test cases, or large-scale system tests (and what are the proportions of those categories)?"

Regarding the first of these questions of "same or similar" methods, Begel and Zimmerman [3] conducted a survey that elicited 607 responses from software engineers and found that 81% of testers said that answering questions such as "How should we handle test redudancy and/or duplicate tests?" was worthwhile. For the second of these questions, regarding the composition of our test suite in terms of unit, integration, and system tests, Begel and Zimmerman also found that their respondents asked "What is the cost/benefit analysis of the different levels of testing *i.e.,* unit testing vs. component testing vs. integration testing vs. business process testing?" Moreover, we might expect that some software be more heavily tested by unit tests (*e.g.,* a utility library that has little interaction between its methods) and other software to be more heavily tested by system tests (*e.g.,* a tool with a user interface in which all components of the system work together).

Existing developer tools cannot easily help to answer such questions. And, some research techniques (*e.g.,* dynamic slicing and change impact analysis) might only help to address a subset of them, and are not currently implemented in a way that developers can easily utilize them in practice. For each such question, we could imagine developing a specialized

Fig. 1: Morpheus Web Application UI, visualizing MAVEN test results

analysis technique to solve just that one problem (*e.g.,* using clustering on per-test-case execution data to identify groups of similar executions), but giving developers a way to view, query, discover, and explore their test execution behavior would allow for developers to form their own questions and answer them. Our goal with this work is to help to address these problems and to answer these questions, and more, with a query-able and interactive global-overview visualization of test-case execution.

III. MORPHEUS VISUALIZATION

One way to simultaneously comprehend test and product code is to trace and surface relations between them; like done in a test-coverage matrix (or simply, "test matrix"). A test matrix places test cases on one dimension/axis and the program entities to be covered or executed on the other dimension/axis. Consider the simple test matrix shown in Figure 2a that depicts the test cases on the vertical axis and the program methods to be executed on the horizontal axis, *i.e.,* each row represents a single test case, and each column represents a single method. The cells at the intersection of the rows and columns represent whether the test case (row) executes (*i.e.,* covers) the method (column). For example, the first row in Figure 2a shows the coverage for Test 0. Test 0 covers Methods A, E, and H. Similarly, Method A was executed by Tests 0, 3, and 9.

Simple test matrices such as this can concisely represent the code coverage of a program's test suite and reveal nuanced details of the test suite. However, test matrices for real-world test suites with thousands of test cases can be visually intractable and run into the information overload challenges. By itself, a test matrix is a static, unchanging data structure.

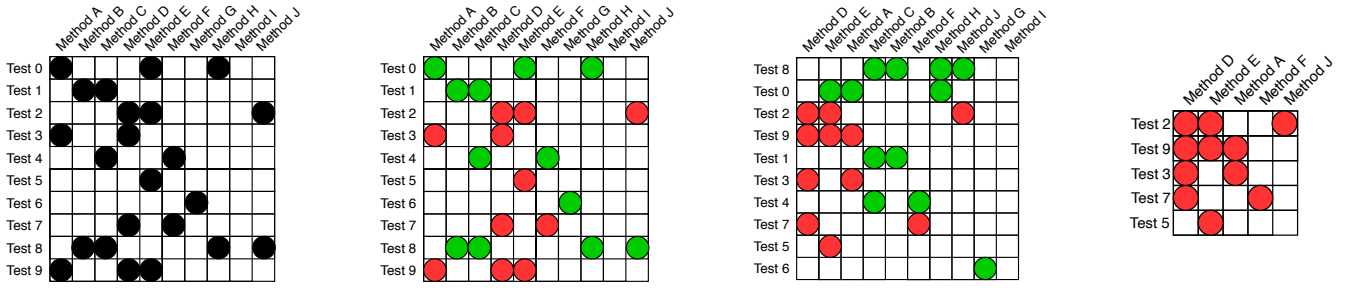
To address challenges of scale and scope, we re-imagine the test matrix as an interactive, dynamic visualization that reveals relations between tests and product code in a software project, at various levels of abstraction and detail. We refer to this interactive, visual rendition of the test matrix as the MORPHEUS VISUALIZATION. We next detail key visual and interactive elements of MORPHEUS: (1) Artifacts along Rows and Columns; (2) Color Overlays; (3) Juxtaposition of Artifacts via Sorting; (4) Drill-downs via Filtering.

Artifacts along Rows and Columns. The rows and columns of the test matrix can represent multiple artifacts. For example in Figure 2a, rows represent test cases and columns represent program methods. However, this mapping is arbitrary and could equivalently be reversed (*i.e.,* methods on rows and test cases on columns). Moreover, we envision that each of these dimensions can be configured to represent other artifacts. For example, one could have test cases on one axis and other granularities on the other axis: (a) Individual source-code lines; (b) Methods; (c) Source files; and (d) Modules or Packages.

Color Overlays. To link an artifact on a specific row and column, MORPHEUS shows a colored dot or node at the intersecting cell of the row and column. The matrix in Figure 2a shows a black colored dot at the intersection of Test 0 and Method A, to reveal that Test 0 executed Method A. MORPHEUS affords overlaying such intersecting dots with different colors to highlight additional information about a relation between two artifacts along rows and columns.

Consider the green- and red-colored intersecting dots in Figure 2b. Those colors show passing (green) and failing (red) tests. Such color overlays show that Test 0 (with green dots) is passing, while Test 2 (with red dots) failing. Moreover, we can also discern that Method A, with one green and two red dots in its column, is executed by both passing and failing tests; and thus, may be the source of a fault.

In our evaluation, we also use color overlays to highlight



(a) Test matrix presenting tests and methods as rows and columns; the intersection shows a dot if it was covered.

(b) Coverage colored according to pass (green) or fail (red).

(c) Sorting tests and methods by coverage.

(d) Filtering to only failing tests and the methods executed by them.

Fig. 2: Test matrices organized and presented in various ways.

test pass-or-fail results and test type (*e.g.,* unit, integration, or system tests). We envision overlaying other metrics using color, such as: performance data, code ownership, code churn, and distinct feature areas. Additionally, engineers would be able to define their color overlays to gain insights into specific problems of interest.

MORPHEUS also affords color overlays on artifact labels along rows and columns. Coloring the artifact labels themselves can reveal organizational patterns in the test and production code. For instance, if all methods along the columns were colored using their package names, engineers would discern which methods belong to specific modules; or how such modules are interspersed when their methods are split into failing and passing methods.

Juxtaposition of Artifacts via Sorting. Artifacts within a software project — tests, source lines, methods, files, packages — are typically related to each other. Latent dependencies snake across such artifacts to tie them together into a single software product. Therefore, users may make meaningful discoveries by reordering the rows and columns in the matrix visualization to juxtapose related artifacts. For instance, reordering rows to cluster tests that execute the same or similar methods or packages may be revealing. Scattering those test rows across 100s or even 1000s of other tests in the matrix does little to identify patterns in how such tests are different or similar to each other.

MORPHEUS presents a variety of sorting functions to organize the rows and columns such that related artifacts represented on those rows and columns can be bundled together. Specifically, we focus on the following sorting functions as part of the evaluation for this work.

1) Bundling tests based on their granularity: Unit, Integration and System;
2) Sorting tests and production artifacts based on their directory pathnames and filenames that they appear in on disk — this mimics the sorting that developers are accustomed to in IDEs;
3) Clustering production artifacts that are tested together;
4) Clustering tests that test common artifacts;
5) Sorting tests and code units (methods, lines, packages) by metrics such as coverage and suspiciousness, respectively.

To illustrate how such sorting could work, consider the matrix in Figure 2c that builds further upon Figure 2b. Both axes in Figure 2c are sorted by coverage. Methods executed by more tests are shuffled to the left, while tests executing more methods are placed at the top. Such sorting could reveal methods that are never or barely tested (*e.g.,* only Test 6 executes Method G, and Method I is untested altogether), or reveal test cases like Test 8 that executes a broad swath of the program (resembling an integration or system test) and Tests 5 and 6 that execute only a single method (like unit tests). Beyond preset sorting capabilities, we envision facilitating developer extensions in the future. Users would be able to plug-in custom sorting functions to comprehend interesting facets of their software project and tests.

Drill-downs via Filtering. When trying to understand how a test suite is verifying product behavior, developers want to focus on specific tests, instead of the whole test suite in one view. To support such detailed exploration, we provide the ability to filter down to tests or production artifacts of interest to developers. Specifically, we focus on the following filters as part of the evaluation for this work.

- **Filter by Name.** This allows developers to filter down to one or many tests, methods or modules by their name.
- **Filter to a Cluster.** This leverages the sorting and clustering capabilities enumerated in the previous section, allowing developers to inspect clusters one at a time, after initially sorting the given artifacts;
- **Filter by test levels.** This filter allows a user to specify a type of test (unit, integration, or system) or specify a specific range of number of test cases. This filter would particularly help in identifying gaps in a test suite's ability to verify product behavior and present opportunities for developers to expand on their testing efforts;
- **Filter by test result.** This filter allows users to choose to see only passing of failing test cases.

Figure 2d illustrates a filtered view of the test matrix in Figure 2c when filtered for failing test cases. We maintain the sort order of methods and tests along the axes, but we filter out passing test cases out and also remove methods that are not covered by failing tests. Such filtered views may be useful when debugging faults that are causing the test-case failures.

IV. Implementation

As shown in Figure 3, the MORPHEUS implementation comprises three components: (1) a per-test coverage data analyzer as a containerized component, (2) a RESTful API serving access to the coverage data, and (3) a web-based visualization of the coverage data. The authors have open sourced this implementation to facilitate reproducibility and extension by other researchers and practitioners [7].

Per-Test Coverage Data Collection. Code-coverage tools typically report aggregate coverage for an entire test suite, which is often composed of several test cases. Collecting code coverage for individual tests (*i.e.,* per-test coverage) provides traceability between those tests and the code that each test executes. We utilize a research tool called TACOCO [8] that collects per-test-case code coverage. Specifically, we use JACOCO's coverage instrumentation [9] within TACOCO's analysis framework. TACOCO discovers the compiled tests within a project and uses an appropriate test framework (*e.g.,* JUnit3/4/5, or TestNG) to run tests. TACOCO's event-based hooks determine the start and end of individual test case executions, which allows it to differentiate the coverage data — as reported by JACOCO — of one test case from the next.

RESTful Access to Coverage Data. The coverage data collected from the per-test analysis is stored in a database, and is exposed via a RESTful API as shown in Figure 3. We collect code coverage at line-level granularity, thus creating granular traceability between tests and product code. For MORPHEUS, we focus on a method-level granularity, and translate line-level coverage to method-level coverage by tracking the line-mappings (beginning- and ending-source line numbers) for methods within a project.

***Web-based* MORPHEUS VISUALIZATION.** Figure 1 shows the front-end of MORPHEUS that we implemented as an HTML5 application built using REACT and D3.JS [10]. The front-end consists of two parts: (1) the test-matrix visualization and (2) the toolbar. The visualization — implemented in D3.JS — shows the connections between methods and tests. The toolbar — built with REACT — provides a set of ways to filter, sort, and query the data. Since the visualization is built using web-based standards, it works with any modern web-browser.

This implementation affords users with the interaction capabilities of sorting, filtering, and color-based encoding of the coverage data, as detailed in Section III, which provides the basis upon which we conduct our evaluations of our test-matrix visualization to support answering developer questions.

V. Evaluation

Using the implementation of MORPHEUS that we presented in Section IV, we evaluated its effectiveness in aiding developers when answering detailed questions about a project's a test suite. We specifically ask the following research questions:

- RQ1 Does MORPHEUS help to reveal the function of individual test cases within a test suite?
- RQ2 Does MORPHEUS help to reveal the global form of a test suite?

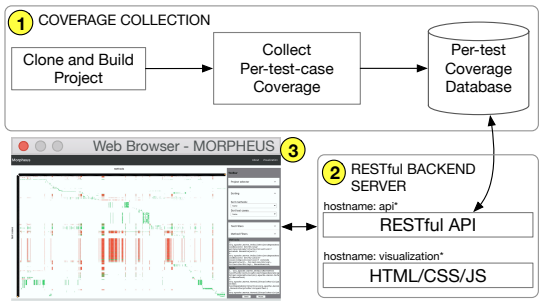


Fig. 3: MORPHEUS Implementation

- RQ3 Can MORPHEUS enable users to discern common or differing patterns across multiple projects?

To answer the research questions, we conducted a user study (composed of three tasks) and two case studies (a single-project and cross-project case study) that examine novel aspects of MORPHEUS that are designed to answer complex questions about a software project and its tests. The studies map to the research questions that they address as follows:

	User Study			Case Study	
	Task 1	Task 2	Task 3	Single-Project	Cross-Project
RQ1	X	X	X		
RQ2			X	X	X
RQ3					X

In terms of research questions, we define *function* of a test case as the functionalities that it tests, methods that it executes, and the similarities and overlaps with other tests; and we define *form* as the overarching patterns throughout a test suite and composition of tests within a test suite, as well as distinguishing similarities and differences across multiple test suites.

During this evaluation, we applied MORPHEUS on a set of four open-source Java projects: (a) COMMONS-CLI: a command-line-options parser library; (b) COMMONS-IO: a utility library for I/O functionality; (c) JSOUP: an XML parser library; and (d) MAVEN: a popular build system. Some statistics about the software projects are presented in Table I. Each of the columns in Table I provides statistics about the Java artifacts specifically (*i.e.,* number of Java files, number of Java lines of code, number of Java methods under test, and number of JUnit test methods). Further, just for the purposes of the evaluation, we define "unit tests" as tests that execute methods within a single class; "integration tests" are tests that execute methods across multiple classes in a single package; and "system tests" are tests that execute multiple methods that reside in at least two different packages.

TABLE I: Software Projects

Project	# Files	# LOC	# Methods	# Test Cases
Commons CLI	52	7066	247	355
Commons IO	360	40032	1468	1780
Maven	1033	91784	6626	682
JSoup	132	23181	1245	783

User Study. We conducted a user study to evaluate MORPHEUS in aiding real software engineers in the context of testing and engineering tasks such as debugging, code refactoring, and on-boarding to a new project. We presented participants

with the sources and builds for COMMONS-CLI and asked them specific questions about individual methods and tests in the project. We chose to use COMMONS-CLI among our software projects for the user study because of the limited ability of traditional tools to support answering questions, and the time limitations of our user study — COMMONS-CLI was suitably limited in size (in terms of number of tests and methods) to allow for the user study to be conducted in the time alloted (one hour per participant). In all, 20 software engineers participated in the user study, with a mean experience of 7.7 years of software development experience, 6.3 years in object-oriented programming, 5.9 years in Java programming, and 4.9 years in software testing. The 20 participants were largely composed of graduate (13) and undergrad (2) students from the department of the authors, as well as software professionals (5) working for large, reputable software-industry corporations.

The study was conducted as a series of one-on-one sessions, averaging around 45 minutes each, and consisted of two rounds: (a) "IDE" round; and (b) "Visualization" round. In both rounds, each participant performed three software-engineering tasks for the same software program: once with their own IDE or toolchain of their choice (*i.e.,* the "IDE" round), and once with MORPHEUS (*i.e.,* the "Visualization" round). Across both rounds, the participants performed similar software-engineering tasks, but focused on different methods and test cases. For example, if a participant was asked to identify integration tests for `MethodA` during the IDE round, then during the Visualization round, the participant was asked to identify integration tests for `MethodB`. Further, the participants were split into two groups — one group would perform the tasks for `MethodA` in the IDE round, and for `MethodB` in the Visualization round; the other group was asked to do the opposite to avoid any bias due to one method potentially making the task more difficult.

In the "IDE" round, the participants could use any tool in their developer toolkit and were asked to report the tools they used. The participants used a variety of tools: IntelliJ,VS Code, Eclipse, Vim, and `grep`. Most participants used IntelliJ (10 participants), Eclipse (6), or VSCode (2), while every other tool was used by at most one participant. We also helped the participants in setting up the project so they could, at minimum, run the test suite and obtain code coverage information using JACOCO [9].

Before the "Visualization" round, the users got a brief hands-on training with MORPHEUS, on a different, smaller program than the one used during the experiment. The training allowed participants to learn about the features of MORPHEUS.

User Study Tasks. In each task in the user study, we asked participants to answer questions about tests and methods in COMMONS-CLI. We framed those questions in scenarios that developers often run into, and would ask similar questions about their code and tests.

Developers often re-run tests after modifying code to check if the change caused any regression. However, current tools and IDEs offer limited support to trace how individual tests execute specific methods. Many IDEs have code-coverage tools, but it provides a file-centric view of the production code — showing which lines are covered, but not by which tests. So, we presented participants with this first task:

> *Task 1* List the set of tests that cover the following method:
> **Group A:** HelpFormatter.getOptPrefix()
> **Group B:** MissingOptionException.getMissingOptions()

Methods that fail together, *i.e.,* they are executed by the same failing test cases, may offer clues about a failure's root cause. To mimic such a scenario, we asked participants to complete this second task:

> *Task 2.1* List the set of methods that are also executed by the one or more of the same failing test cases that execute:
> **Group A:** Option.setArgName(String)
> **Group B:** Option.setValueSeparator(char)
>
> *Task 2.2* List the set of failing tests that are testing method:
> **Group A:** Option.setArgName(String)
> **Group B:** Option.setValueSeparator(char)

Developers often write multiple types of tests: unit, integration, and system tests. Understanding how the system is executed by various kinds of tests can give insight into a specific method's test plan and testability. As such, the participants' third task focused on understanding the composition of the test suite:

> *Task 3.1* List the set of unit tests for method:
> **Group A:** HelpFormatter.findWrapPos(String,int,int)
> **Group B:** HelpFormatter.renderWrappedText(StringBuffer,int,int,String)
>
> *Task 3.2* List the set of integration tests for method:
> **Group A:** HelpFormatter.findWrapPos(String,int,int)
> **Group B:** HelpFormatter.renderWrappedText(StringBuffer,int,int,String)

Empirical Results. Across both rounds, we tracked participants' performance in two ways: (1) time taken by a participant to complete each task (within a 5-minute time limit per task), and (2) the correctness of a participant's answer to each question. In Table II, we report "correctness" results, as the mean precision, recall, and f-score for each task performed by users. We report mean scores for both the IDE and visualization tasks. Figures 4a and 4b show the boxplots for the precision and recall for each task. Additionally, we compute Precision, Recall, and F-score as follows:

$$\text{Precision} = \frac{\#\text{Correct Answers}}{(\#\text{Correct Answers}) + (\#\text{Incorrect Answers})} \quad (1)$$

$$\text{Recall} = \frac{\#\text{Correct Answers}}{(\#\text{Correct Answers}) + (\#\text{Correct Answers Omitted})} \quad (2)$$

$$\text{F-score} = 2 \times \frac{(\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})} \quad (3)$$

For Equations 1 and 2, "Answers" are provided by the study participants in the form of a set of methods or test cases. For example, for Task 1, each participant provided their answers in the form of a list of test cases that executed a specified method, and for Task 3, each participant provided their answers in the form of a list of methods.

TABLE II: Mean Precision, Recall, and F-score results.

Tasks	Precision		Recall		F-Score	
	IDE	Vis.	IDE	Vis.	IDE	Vis.
Task 1	0.68	1.00	0.18	1.00	0.23	1.00
Task 2.1	0.25	1.00	0.01	0.97	0.02	0.98
Task 2.2	0.11	0.90	0.05	0.90	0.05	0.90
Task 3.1	0.60	1.00	0.41	1.00	0.39	1.00
Task 3.2	0.19	0.94	0.10	0.94	0.10	0.94

The row for Task 1 in Table II suggests perfect mean precision and recall scores of 1.0 for the Visualization rounds. Whereas, for the IDE rounds the mean precision and recall scores stand at 0.68 and 0.18, respectively. Notice, this trend continues for all tasks. The participants consistently performed better using MORPHEUS, in comparison to when using their own development environment.

The low mean precision scores suggest that with the IDE, when trying to report the correct set of methods or test cases for each task, the users consistently reported an incorrect set of methods and test cases. Moreover, the even lower mean recall scores indicate that they never reported many methods and test cases that they otherwise should have.

TABLE III: Mean time (secs.) taken by participants per task.

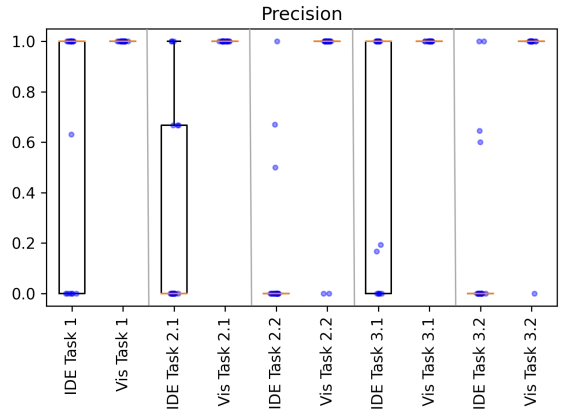
Rounds	Mean Time (seconds)		
	Tasks 1	Task 2.1 & 2.2	Tasks 3.1 & 3.2
IDE	189	279	285
Visualization	86	139	177

Next, Table III presents the mean time taken (in seconds) by a participant to finish each task. In Figure 5, boxplots show the timing results per task and for both rounds. During the IDE round, the participants made use of a variety of tools to get to an answer. The results suggest that the participants were faster in reporting answers with MORPHEUS.

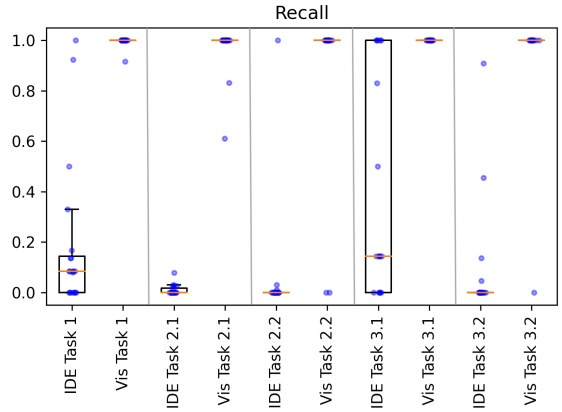
Finally, we also asked participants their level of satisfaction with each of the tool treatments, on a scale of 1 (least satisfied) to 10 (most satisfied). The mean satisfaction score for the IDE treatment (*i.e.,* using any tool available) was only 4.4, whereas the mean satisfaction score for MORPHEUS was 8.8.

COMMONS-CLI *Case Study.* To highlight to the reader the ability of MORPHEUS to reveal the overarching, global behavior of a test suite, we present a case study of a single software project (other projects are visualized in the next "Cross-Project-Comparison Case Study"). When discussing the global form of the test suite, we specifically expect to be able to answer the following questions: (1) "what is (and is not) tested?"; and (2) "what types of tests are present in the test suite?" Using COMMONS-CLI's test suite as a case study, we show how MORPHEUS can aid in answering such questions about that project.

Figure 6a shows how MORPHEUS presents the entire test suite for COMMONS-CLI to a developer. The initial view of the test matrix gives us an overview of all the methods (horizontal axis) and tests (vertical axis). The tests and methods are both sorted based on their names, and names of their enclosing packages and classes. Based on this view, we can make two observations. First, the horizontal strings of (green and red) test rows suggest that most tests seem to execute many methods. Second, the similarity between the horizontal (green and red) test rows shows that many tests execute common methods and are perhaps variations of one another.



(a) Boxplots for Precision of each task and tool.



(b) Boxplots for Recall of each task and tool.

Fig. 4: Precision and Recall of each task and round. (Higher is better)

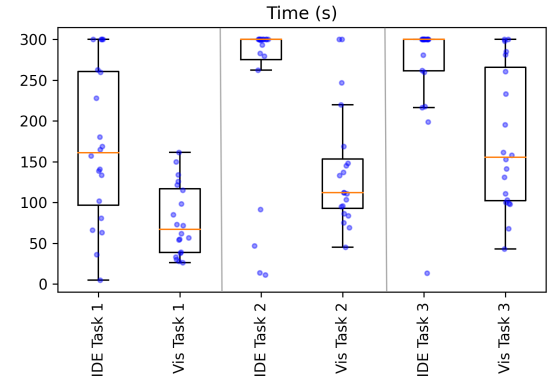
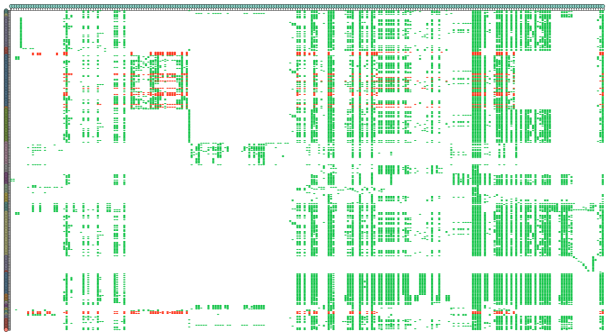
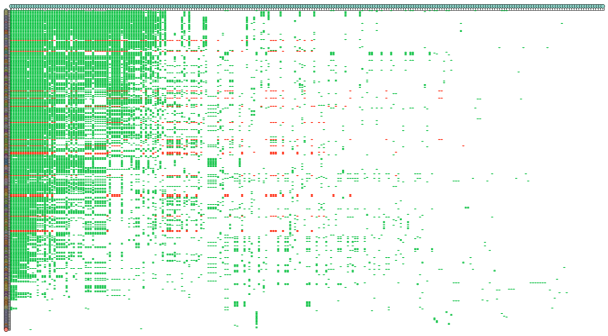


Fig. 5: Time (seconds) taken by participants to complete each round; separated per task. (Lower is better)

"Tested, or Not Tested". To determine what is (or not) tested, we sort the methods and tests by their coverage (*i.e.,* number of methods executed by a test; and number tests that a method is executed by). Figure 6b shows the sorted view of COMMONS-CLI's test suite, with frequently executed (and tested) methods



(a) COMMONS-CLI. All tests and methods sorted by name.



(b) COMMONS-CLI. Tests & Methods sorted by coverage.

Fig. 6: Sorting COMMONS-CLI's Methods and Test cases.

to the left, and tests executing the most number of methods at the top. By sorting the tests and methods, it becomes apparent that a selection of methods is very well-tested (on the left side), and the further you move to the right the sparser the coverage becomes, to the point of no coverage for a (small) group of methods. Developers can use this view to directly take steps on where the test suite can be improved.

"Type of Tests". Finally, MORPHEUS enables filtering and coloring to the type of test cases: *e.g.,* passing versus failing, or unit versus integration versus system. Figures 6a and 6b color the tests according to their pass/fail status: green denoting passing and red denoting failing. Alternatively, Figure 7c shows the colored result for each type of test: orange dots indicating integration tests and green dots indicating unit tests. From this view, we can clearly observe that COMMONS-CLI tests are primarily written as integration tests (326 tests), with limited amounts of unit tests (29 tests).

Case Study Inference. MORPHEUS provides multiple ways to better understand the test-suite composition, allowing engineers to sort through tested and untested methods and filtering to show the different types of tests. Combining filters, sorts, and coloring allows us to explore what is covered, what is tested together, and what types of tests are in the test suite.

Cross-Project-Comparison Case Study. So far, we focused on a single software project's test suite, and its form and function. We now compare and contrast visualizations from multiple projects, as a case study, to see if lessons can be learned about different test suites from differing software systems. Such inter-project comparisons may be useful to allow software engineers to assess if the degree and type of testing is appropriate for their type of program. For example, one may expect that an API-based utility library to be largely comprised of unit tests — each method in the library performs some function that can be independently tested with limited dependencies among those methods. Similarly, for an interactive system, one may expect to find test suites that have many more system and integration tests, in addition to unit tests, because the system itself relies upon multiple interacting components to perform its functionality. Figure 7 shows how MORPHEUS presents four projects — MAVEN, JSOUP, COMMONS-CLI, and COMMONS-IO — coloring according to test type: purple indicates system tests, orange indicates integration tests, and green indicates unit tests.

TABLE IV: Distribution of type of tests (in percentage).

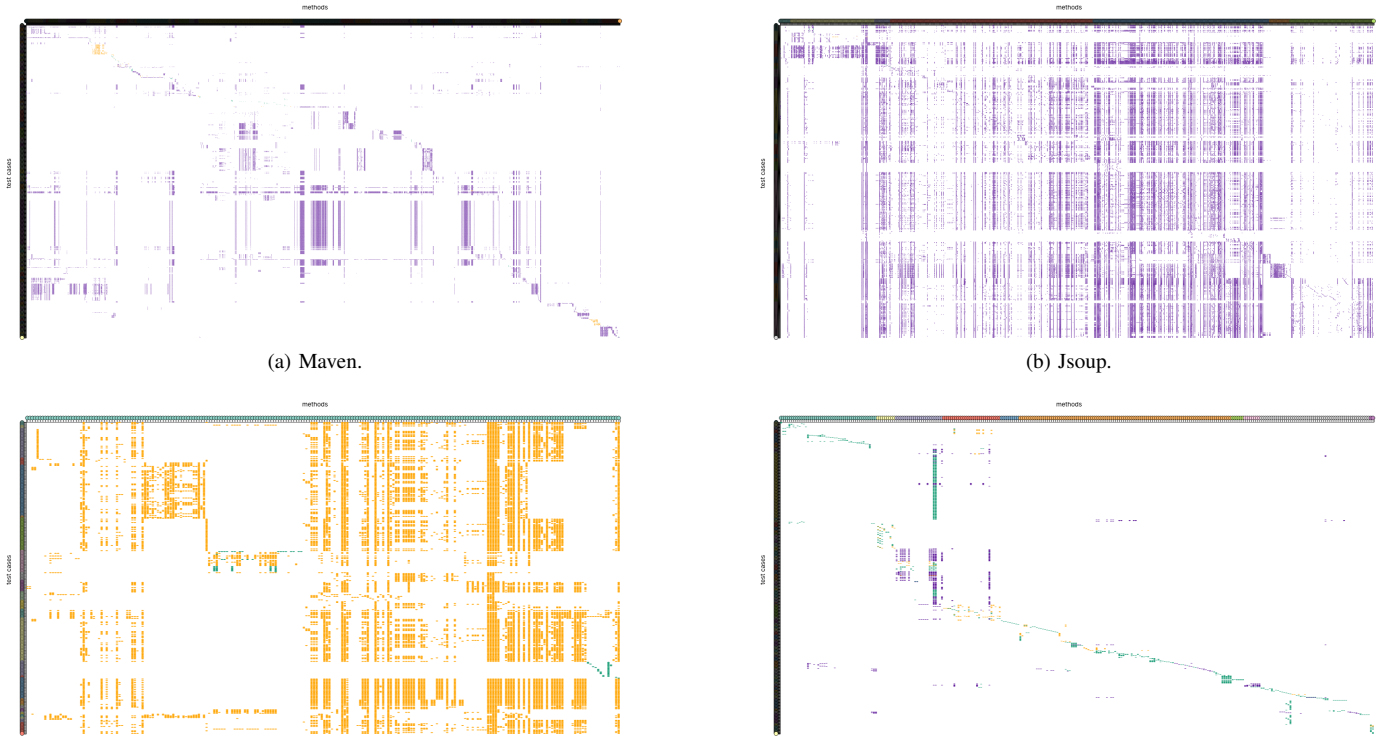
Test Type	Maven	Jsoup	Commons-CLI	Commons-IO
Unit Tests	10%	3%	8%	71%
Integration Tests	7%	0%	92%	9%
System Tests	82%	97%	0%	19%

The visualizations in Figure 7 make two aspects apparent, (1) long vertical lines, and (2) the sparseness of some matrices. The long vertical lines show us many tests cover the same or similar sets of methods. We see this happen mainly with JSOUP (Figure 7b), and COMMONS-CLI (Figure 7c), which can be attributed to the way tests are structured. MAVEN also contains some longer vertical lines, but it is not as apparent as JSOUP and COMMONS-CLI.

JSOUP structures their tests mainly around a small XML string that is being parsed by the library and finally, the results are verified. As a result, the majority of the tests are variations of each other, with each test covering different corner cases.

To test some parts of the COMMONS-CLI library, the project's developers perform three steps: (1) create a string array, (2) create for each test a command-line argument parser, and finally, (3) parse the string using the parser. Consequently, many of the tests make use of the same components, causing us to see the long vertical lines in MORPHEUS.

Now, consider the second aspect: difference in sparseness across the visualizations. Both JSOUP and COMMONS-CLI have denser visualizations, while MAVEN and COMMONS-IO are more sparse. This can be attributed, in part, to the composition of the test suites. Table IV shows the distribution of tests within the four projects. JSOUP is comprised almost solely of system tests, whereas COMMONS-CLI is comprised almost solely of integration tests due to all classes living in the same package. As mentioned before, MAVEN also exhibits long vertical lines, but not as much; as evident in its sparser test distribution in comparison to JSOUP and COMMONS-CLI. Finally, one sees that COMMONS-IO focuses more on unit tests, as reflected in the sparseness of MORPHEUS; and this result matches our expectation for a utility library that contains loosely coupled methods.



(a) Maven.

(b) Jsoup.

(c) Commons-CLI.

(d) Commons-IO.

Fig. 7: MORPHEUS visualizing the test suites for four projects: MAVEN, JSOUP, COMMONS-CLI, and COMMONS-IO (purple indicates system tests, orange indicates integration tests, and green indicates unit tests).

Case Study Inference. MORPHEUS is able to reveal patterns to us across projects, *e.g.,* composition of tests, and how developers test their system. The sparseness can indicate the presence of unit, integration, or system tests, while the vertical lines can point to commonly tested methods.

VI. DISCUSSION

Collectively, our user study and case studies reveal that MORPHEUS can aid software engineers in comprehending software tests and test suites. For **RQ1**, the user-study participants answered specific questions about individual tests and methods in a real-world system (Tasks 1, 2, and 3). Moreover, they did so with greater accuracy, and while using less time with MORPHEUS, than they did when using their own development tools. As such for **RQ1**, we are able answer:

> In a controlled user study, MORPHEUS aided experienced software engineers in correctly and efficiently understanding the function of test cases, by revealing how and the degrees to which individual test cases execute specific methods within a real-world software system.

For **RQ2**, the user-study participants needed to assess the composition of the test suite of unit, integration, and system tests for Task 3. Moreover, using COMMONS-CLI's test suite as a case study, we assess if MORPHEUS is able to highlight key aspects of the project's test suite. MORPHEUS's sort, filter, and coloring functionalities enabled us to breakdown COMMONS-CLI's tests into different types: 326 integration tests, and 29 unit tests; suggesting a set of highly interdependent methods that invoke together across a large swath of the project's tests. MORPHEUS was also able visualize methods with sparse, or no test coverage; revealing opportunities to expand test coverage in COMMONS-CLI. Finally, MORPHEUS was successful at highlighting overall form differences among multiple projects in our cross-project-comparison case study, which also demonstrates its ability to reveal global overviews of test behavior. As such for **RQ2**, we are able to answer:

> Across the user study and two case studies, MORPHEUS was able to reveal the overarching form and composition of real-world software test suites, especially in terms of the kind of tests composing the suites, the degree to which the suite executes the underlying software system, and the patterns of execution across multiple test cases.

For **RQ3**, we presented a cross-project-comparison case study, in which we applied MORPHEUS to analyze the global, overarching *form* of test suites, *across multiple projects*. This study represents a scenario in which engineers could assess if the overarching form and structure of their suite is suitable for the program under test. Engineers may do so by visually comparing the test-suite structures for independent software systems with similar architectures or features. For **RQ3** we conclude in the affirmative:

> MORPHEUS was able to highlight notable differences and similarities in the global structure (or form) for test suites across four independent real-world software systems.

In doing so, we were also able to gain insights about the architecture of the software systems themselves.

Our results with MORPHEUS suggest that software-test comprehension goes beyond improving code quality. We find that test comprehension reveals insights about a program's architecture. Understanding software tests may also help in forming meaningful questions about a program that aids in performing practical software engineering tasks, such as refactoring, debugging, or on-boarding a new contributor.

VII. THREATS TO VALIDITY

The main threats to validity in our studies arise from the generalizability of our results. Our study focused on Java systems that used the JUnit testing framework. Although other programming languages may be tested in different ways, the approach that we take in this work could easily be extended to those languages and testing frameworks, and we see nothing about the general approach that renders any of its conceived features more or less beneficial in other environments.

Also, our mean years of software-development experience for the participants in our study was almost eight years. As such, our participants were well versed in their development tools. An argument could be made that a developer with much more (or much less) experience may have performed better on the their development tools than our participants, and although that may be true to some extent, the extreme differences between the accuracy in the IDE and visualization treatments likely demonstrates that such differences would not change the general result.

Finally, our user study included participants who were not developers of the software projects. As such, we cannot generalize our results to developers who already have experience and knowledge of their test suite. However, the questions and tasks in the user study would likely be challenging even with experience in a project. Future work is planned to study the use of MORPHEUS by project contributors.

VIII. RELATED WORKS

Relationship between Test and Production Code. Prior works have explored using per-test-case coverage to aid developers for tasks such as fault localization [11]. Others focused on helping developers localize what has been tested and by what (*e.g.,* [12], [13], [14], [15], [16], [17], [18], [19]). MORPHEUS visualizes dynamically observed, per-test-case code coverage data to reveal traceability between test- and production-code as well. However, MORPHEUS does so in the global context of test cases and production code that house individual code-to-test relationships, enabling answers questions such as, "what other tests are failing when executing a given method?"

Synchronous co-evolution of tests and code has received prior study and investigation (*e.g.,* [20], [21]). This work also studies the relation between tests and code. However, instead of addressing questions about lineage, MORPHEUS helps uncover the composition and working of test cases at a given moment in the lifetime of a software project.

Dynamic Behavior Comprehension. Multiple prior works have studied comprehension of software behavior. Such works typically reveal relationships between different parts of a project's source code, typically using viualizations (*e.g.,* [22], [23], [24], [25]). Similarly, comprehension of software execution traces, aided with visualization also has received prior study (*e.g.,* [26], [27]). The main purpose of such works is to understand a single execution trace for a specific software program. Executions from test-runs can aid in understanding production code. Prior works looked to extract product use-case diagrams based on the behavior of a single test [28], [29].

MORPHEUS also reveals runtime execution data about a software project. However, MORPHEUS reveals such execution data in the context of a project's test suite, by highlighting relations between the project's tests and code.

Matrix-Based Visualizations. Prior works in information-visualization research have employed matrix-based visualizations for a wide variety of applications. Fernandez *et al.* [30] created CLUSTERGRAMMER., a tool to visualize high-dimensional biological data as a matrix visualization. Similarly, matrices have been used to visualize social networks [31], [32], [33]. Prior work has shown the advantage of matrix-based visualizations [34], [35], [36] to explore graph data at many levels. Ghoniem *et al.* [37] shows that the readability of matrix-based visualization outperforms node-link diagrams when graphs become bigger than twenty vertices. Our work too presents high-density information in a matrix-based visualizations. However, the information that we visualize is software test coverage data as applied specifically to the field of software engineering.

IX. CONCLUSIONS

Comprehending test suites for real-world software projects in relation to production code can be challenging for engineers. We approach such challenges using MORPHEUS — a visual tool that traces relations between test and production code. While MORPHEUS traces test-to-code relations in global overviews of a project's entire test suite, engineers can also use MORPHEUS to understand executions of specific tests and methods using exploration functionalities, *e.g.,* filter and sort.

We provide our implementation of MORPHEUS as open source, as well as an interactive demo and replication package with our user study questionnaire and database to facilitate repeating our user study [7].

Our evaluations show that MORPHEUS can provide insights into test suites of real-world systems, and that it consistently outperforms traditional development tools, both in accuracy and time taken to complete software-engineering tasks.

We envision MORPHEUS to evolve into an extensible framework for a variety of sort, filter, and exploration functions that aid software-test comprehension. As next steps, we will build such exploration functionalities by surveying owners and contributors of real-world systems about the typical questions they encounter about their software tests and testing strategies.

REFERENCES

[1] S. Vasanthapriyan, J. Tian, D. Zhao, S. Xiong, and J. Xiang, "An ontology-based knowledge sharing portal for software testing," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2017, pp. 472–479.

[2] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th International Workshop on Automation of Software Test (AST)*, 2012, pp. 36–42.

[3] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 12–23. [Online]. Available: https://doi.org/10.1145/2568225.2568233

[4] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211.

[5] L. Zhao and S. Elbaum, "Quality assurance under the open source development model," *Journal of Systems and Software*, vol. 66, no. 1, pp. 65–75, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412120200064X

[6] R. Torkar and S. Mankefors-Christiernin, "A survey on testing and reuse," 12 2003, pp. 164–173.

[7] "Morpheus," Morpheus Tool, Replication Package, and Code, 2020, https://spideruci.github.io/morpheus.

[8] J. Kim, V. K. Palepu, K. Dreef, and J. A. Jones, "Tacoco: Integrated software analysis framework," Github, https://github.com/spideruci/tacoco, 2015. [Online]. Available: https://github.com/spideruci/tacoco

[9] "Jacoco." [Online]. Available: https://www.eclemma.org/jacoco/

[10] M. Bostock, V. Ogievetsky, and J. Heer, "D^3 data-driven documents," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 12, pp. 2301–2309, 2011.

[11] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the International Conference on Automated Software Engineering*, November 2005, pp. 273–282.

[12] A. Tahir and S. G. MacDonell, "Combining dynamic analysis and visualization to explore the distribution of unit test suites," in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. IEEE, 2015, pp. 21–30.

[13] N. Koochakzadeh and V. Garousi, "Tecrevis: A tool for test coverage and test redundancy visualization," in *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, ser. TAIC PART'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 129–136.

[14] A. F. Otoom, M. Hammad, N. Al-Jawabreh, and R. A. Seini, "Visualizing testing results for software projects," in *Proc. of the 17th International Arab Conference on Information Technology (ACIT'16), Morocco*, 2016.

[15] M. Hammad, A. F. Otoom, M. Hammad, N. Al-Jawabreh, and R. Abu Seini, "Multiview visualization of software testing results," *International Journal of Computing and Digital Systems*, vol. 9, no. 1, 2020.

[16] T. Tamisier, P. Karski, and F. Feltz, "Visualization of unit and selective regression software tests," in *International Conference on Cooperative Design, Visualization and Engineering*. Springer, 2013, pp. 227–230.

[17] B. Van Rompaey and S. Demeyer, "Exploring the composition of unit test suites," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*. IEEE, 2008, pp. 11–20.

[18] N. Aljawabrah and A. Qusef, "Tctracvis: test-to-code traceability links visualization tool," in *Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems*, 2019, pp. 1–4.

[19] A. Rodrigues, M. Lencastre, and A. d. A. Gilberto Filho, "Multivisiotrace: traceability visualization tool," in *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2016, pp. 61–66.

[20] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[21] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani, "Chronotwigger: A visual analytics tool for understanding source and test co-evolution," in *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 2014, pp. 117–126.

[22] A. Kuhn, D. Erni, P. Loretan, and O. Nierstrasz, "Software cartography: thematic software visualization with consistent layout," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 191–210, 2010. [Online]. Available: http://dx.doi.org/10.1002/smr.414

[23] F. Deng, N. DiGiuseppe, and J. A. Jones, "Constellation visualization: Augmenting program dependence with dynamic information," in *Proceedings of International Workshop on Visualizing Software for Understanding and Analysis*, 2011, pp. 1–8.

[24] V. K. Palepu and J. A. Jones, "Revealing runtime features and constituent behaviors within software," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE, 2015, pp. 86–95.

[25] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow, "Cluster analysis of Java dependency graphs," in *Proceedings of the 4th ACM symposium on Software visualization*, ser. SoftVis '08. New York, NY, USA: ACM, 2008, pp. 91–94. [Online]. Available: http://doi.acm.org/10.1145/1409720.1409735

[26] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen, "Understanding execution traces using massive sequence and circular bundle views," in *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ser. ICPC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 49–58. [Online]. Available: http://dx.doi.org/10.1109/ICPC.2007.39

[27] Y. Feng, K. Dreef, J. A. Jones, and A. van Deursen, "Hierarchical abstraction of execution traces for program comprehension," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 86–96. [Online]. Available: https://doi.org/10.1145/3196321.3196343

[28] B. Cornelissen, L. Moonen, A. van Deursen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *2007 11th European Conference on Software Maintenance and Reengineering*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2007, pp. 213–222. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CSMR.2007.54

[29] S. M. Nasehi and F. Maurer, "Unit tests as api usage examples," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[30] N. F. Fernandez, G. W. Gundersen, A. Rahman, M. L. Grimes, K. Rikova, P. Hornbeck, and A. Ma'ayan, "Clustergrammer, a web-based heatmap visualization and analysis tool for high-dimensional biological data," *Scientific data*, vol. 4, p. 170151, 2017.

[31] N. Henry and J.-D. Fekete, "Matlink: Enhanced matrix visualization for analyzing social networks," in *IFIP Conference on Human-Computer Interaction*. Springer, 2007, pp. 288–302.

[32] J. S. Yi, N. Elmqvist, and S. Lee, "Timematrix: Analyzing temporal social networks using interactive matrix-based visualizations," *Intl. Journal of Human–Computer Interaction*, vol. 26, no. 11-12, pp. 1031–1051, 2010.

[33] N. Henry, J.-D. Fekete, and M. J. McGuffin, "Nodetrix: a hybrid visualization of social networks," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 6, pp. 1302–1309, 2007.

[34] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry, and J.-D. Fekete, "Zame: Interactive large-scale graph visualization," in *2008 IEEE Pacific Visualization Symposium*. IEEE, 2008, pp. 215–222.

[35] J. Abello and F. Van Ham, "Matrix zoom: A visual interface to semi-external graphs," in *IEEE symposium on information visualization*. IEEE, 2004, pp. 183–190.

[36] A. Abuthawabeh, F. Beck, D. Zeckzer, and S. Diehl, "Finding structures in multi-type code couplings with node-link and matrix visualizations," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2013, pp. 1–10.

[37] M. Ghoniem, J.-D. Fekete, and P. Castagliola, "On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis," *Information Visualization*, vol. 4, no. 2, pp. 114–135, 2005.