UNIVERSITY OF CALIFORNIA,
IRVINE

Visualizing the Form and Function of Test Suites

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Kaj Dreef

Thesis Committee:
James A. Jones, Associate Professor, Chair
André van der Hoek, Professor
Cristina Videira Lopes, Professor

2020

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

As I conclude my time at UCI, there are many I would like to thank for the impact they have had on me and the support they have provided me.

I would like to thank my advisor, James Jones. During my time at UCI, you have always been there to support and guide me. I have learned so much from you and could not have wished for a better advisor and mentor. I am grateful for all your advice over the years, and am look forward to continuing to work with you in the future.

A sincere thank you to Vijay Krishna Palepu, whose love for research and software engineering is contagious. You have constantly challenged me to think deeper and I have appreciated your time advising me this past year. You have made me a better software engineer and critical thinker.

To my committee: André, thank you for always having an open door, supporting me through my years at UCI, and making sure my Dutch isn't rusty. Crista, you are an incredible software engineer who I have learned so much from and I am immensely thankful for the chance to work alongside you as your TA.

To my parents, Birgitta and Frank: Long distance has extended from 65 km away in Delft to 9,000 km away in California, but your support (and faces) are never more than a FaceTime call away. Thank you for being with me every step of the way and for encouraging me to step outside my comfort zone once in a while.

To my "zusje", Camilla: Thank you for your periodic reminders to go outside and feel the sun on my face, and to not just sit in a room staring at my screen.

To my wife, Samantha: Thank you for your support (and the 24/7 proofreading you provided) during the stressful weeks and sleepless night, and for daily, step-away-from-research walks with you and Luna to clear my head. Your support, for both me and my research, has meant the world to me.

ABSTRACT OF THE THESIS

Visualizing the Form and Function of Test Suites

By

Kaj Dreef

Master of Science in Software Engineering

University of California, Irvine, 2020

James A. Jones, Associate Professor, Chair

The test suite of a software project can be characterized by several meaningful questions, such as "does the suite contain unit, integration or system tests?"; "which methods are covered by the tests?"; "how often does a method get executed by the test suite?"; "is there an opportunity to write additional tests?"; and "are multiple test cases testing a common, or related set of methods?"

Answering such questions can help engineers understand the overall nature of a test suite and its ability to test its associated software project. However, the existing IDE-focused tools, available to developers today, make finding answers to such seemingly basic questions challenging. Moreover, any IDE-based tooling typically shies away from providing a global overview of a project's test suite. Without a global overview, it can particularly challenging to establish the overall context in which a method is executed by a fleet of various test cases, how such a method may relate to other methods in how it is executed by different tests, and similarly how test cases may relate with each other in how they execute a shared, or related set of test cases.

In an effort to overcome such challenges, this thesis presents a novel interactive visual tool that provides a global overview of the tests available in a project's test suite, specifically in the context of the methods available in the project's codebase. Through a series of inter-

active functions to sort, filter, query, and explore a test-matrix visualization, I demonstrate how developers can effectively answer questions about their project's test suite, and how such interactive visualizations can aid in the overall testing effort of the code that they are developing.

The evaluations, performed on four real-world software systems, consist of two components: (1) three case studies presenting how the interactive visualization can provide insights into the test suite, and (2) a user study that shows the visualization consistently outperforms the participants' development environment, both in precision/accuracy and time it takes to complete the tasks.

Chapter 1

Introduction

1.1 Importance of Testing and Understanding Testing Efforts

A test suite in a software project is often as elaborate and meaningful as the software product that it is testing. It is not uncommon to come across a project where each method is tested by hundreds of test methods (*e.g.,* Guava, Apache Commons). In such cases, the scale and complexity of a software product is often matched by its own suite of tests.

Further, software tests supply a meaningful blueprint of how a software program is expected to behave. When verifying the actual behavior of a program, tests need to independently establish the desired or expected behavior of the software product. Thereby, software tests also aid in software comprehension, wherein they help engineers understand not just how the product actually behaves, but how it is supposed to behave.

While tests themselves are rarely ever deployed or packaged as part of the software product to its end users, they are an important tool for a software engineer. Understanding the form

and function of a test suite within a software project is critical for any software engineer.

1.2 Understand Form and Function of the Test Suite

Consider some questions that engineers routinely ask of their test suite, such as "Which components are tested?"; "Which components remain untested or under-tested?"; "Which tests exercise individual units of code, or the product/system as a whole?"; "Which methods do the tests execute when failing?"; and "How similar are two tests in the methods that they execute?" Such questions speak to: (a) how the tests themselves are organized, *i.e.,* the "form" of the test suite, and (b) the specific areas and aspects of the product that the tests are designed to exercise 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 how a test suite is, or can be organized, which can be informed by the code coverage, both each test case individually and comparisons among test cases within the suite. Tests can also be organized into clusters, where each cluster represents a common behavior to be verified. A more common organizing principle found in real-world software test suites is to distinguish between unit, integration, and system tests. In such cases, the tests are organized into components that mirror the packages and components in the production code that are directly verified by their respective tests.

Organizing the test suite in different ways often reveals different aspects of the suite as a whole, and helps 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 re-factor or modify. Similarly, when an engineering manager is trying to assess the overall state of a product's testing

effort and strategy, it can be useful to organize the tests by their coverage (increasing to decreasing), or cluster them by the product behaviors they are trying to test.

While the "form" of a test suite addresses questions of a global flavor, questions about the "function" of tests require a more localized consideration. The "function" of a test deals with the specific desired behavior that the test is testing for, and the specific components and methods that it executes when doing so. Developers often have questions about how an individual test works, especially in the methods that it tests and covers. Such questions help in evaluating the efficacy of the test in question. Conversely, developers often want to know which tests can possibly execute a given method, so they can re-run those tests only rather than the entire test suite.

Indeed, questions about the global form of an entire test suite, and the localized function of individual tests are often inextricably related. For instance, when asking, "which tests are executing my software component?" it is often useful to understand how those tests are categorized as unit, integration, and system tests, or which behavior clusters might include such tests. Similarly, organizing the entire test suite by code coverage of individual tests may actually highlight a specific method or component that remains entirely untested. As such, it becomes necessary to pool together the data that simultaneously answers questions about both the global form and local function of software tests on a single unified canvas.

1.3 My Solution: Interactive Visualization to Support Understanding

Test suites in real world software projects offer high volumes of data and information that together help answer meaningful questions for engineers. In this work, I present such high-volumes of software test data in the form of an interactive matrix-styled visualization. I aim

to present engineers with global overviews of their software test suites in a visualization that is high in terms of information density. High information-density visualizations, especially for global overviews, are effective in revealing patterns within both the overarching form of test suites and the methods/components under test. To seamlessly shift between global overviews and more localized functionalities of specific tests, this work enables user-driven interactions within the visualization. Such interactions enable engineers to quickly narrow the scope of the data presented within the visualization to individual or a subset of tests and methods within a software project. Engineers would also be able to re-organize the data (tests and methods) in a way that helps reveal patterns, irrespective of the scope of the data presented — local or global.

I implemented such a visualization in a tool that called Morpheus. Using Morpheus, I evaluated the effectiveness of the visualization in helping engineers answer questions about the tests within a software project. Further, I conducted multiple case studies that are presented in this thesis, where each case study highlights novel aspects of Morpheus that help in answering complex questions about a software project and its tests.

The main contributions of this work are:

1. A novel application of the matrix-styled visual representation towards presenting high information-density insights into test suites within a software project;

2. A series of interaction capabilities, atop the test suite visualization, that enable seamless transition between global and local views of a software test suite to help answer questions about the form and function of software tests;

3. Morpheus [9]: An open-source, publicly deployed, and accessible web-application[1] that implements the proposed visualization and interaction capabilities; with the ability to visualize test suites for large, real world projects.

[1] `http://morpheus.kajdreef.com/`

4. An evaluation of the proposed visualization and interaction approaches when answering engineering-focused questions about test suites in real world software projects.

Chapter 2

Software Test Comprehension: Motivations and Challenges

For projects with established automated testing practices, and sizable test suites, the challenge of software comprehension often is not limited to understanding production code[1]. For such projects, test comprehension, *i.e.,* understanding test code that verifies and validates the product code, can also be a challenge. However, first consider why software tests are important in software engineering.

2.1 Motivations for Testing

Software tests help in performing three functions: (a) executing production code in a controlled environment; (b) verifying product behavior and correctness; and (c) comprehending expected product behavior. Let us consider how these functions aid in the overall process of software engineering.

[1]I consider any code that is used by end-users as part of a software product (or service) to be "production" code to distinguish it from its associated "test" code.

Executing production code The first step in verifying code is to execute it. Issues relating to the internal consistency in code are often revealed (*e.g.,* as null-pointer exceptions, or stack and buffer overflows) by simply executing code, without any explicit correctness checks. In other words, just by executing product code — even without checking for correctness — a test case can highlight runtime failures in production code. And by doing so in a controlled environment, tests enable the reproduction of an executable code path, often in a way that is desirably deterministic.

Verifying product behavior Tests also establish a test oracle [18, 40] that can compare the actual behavior of product code, against expected behavior and assess if the expected and actual behaviors are the same. Such test oracles are the mechanism by which correctness is assessed.

Comprehending expected behavior Engineers can perhaps best understand product behavior by reading and inspecting the production code. However, by independently establishing a test oracle, tests provide an alternate avenue for engineers to understand both actual and expected product behavior. Alongside production code, engineers can also read and inspect test code to better comprehend product behavior.

Tests aid in critical tasks such as software comprehension and verification; thus, making tests central to understanding product code and behavior. By extension, not understanding the form and function of tests can pose impediments to understanding production code and behavior. However, understanding tests is also riddled with its own uniques challenges, which I explore next.

2.2 Challenges in Test Comprehension

Software tests are software programs themselves. As such, there is no reason to suggest that unlike any other software program, test code and logic will be programmed correctly. The same problems concerning correctness and comprehension of production code are concerns when authoring test code and logic. For instance, it is possible to have faulty and error-prone test code. It may also be entirely possible to not completely understand how a test functions — what it tests or how it establishes a test oracle. For example, the same testing logic may be cloned, inadvertently, across different parts of the test suite, resulting in poorly architected test code.

However, in addition to such maintenance challenges, comprehension of test code often runs into its own set of unique challenges.

2.2.1 Challenge: Large test suites with many tests

When done correctly, each unitary component of software code (*e.g.,* a function or method) is accompanied by multiple tests that verify various aspects of the component's correctness and behavior. As such, a mature test suite for real-world projects often have large test suites with hundreds, and often thousands, of test cases. Such large arrays of test cases can be necessary in adequately testing a software system. And yet, navigating thousands of tests for a software product presents a problem of scale in comprehension.

Consider the following questions. If a developer were to write a new test for a function, how does she assess that a similar test does not already exist? Or, how does an engineer identify all existing tests that verify an overarching behavior or feature in a software product? Conversely, how does an engineer understand gaps in what a product's test suite is testing?

Such questions can be answered when the number of tests are small and manageable. However, as tests grow into the thousands, answering even basic questions about a product's test suite can inch towards a daunting prospect.

2.2.2 Challenge: Maintaining test suites: an unlikely priority

Software tests are typically not shipped as part of packaged software, or deployed as a software service, to be consumed by end-users. And, it is not uncommon for a software project to commit only finite resources towards testing, thereby, limiting the time spent on understanding and maintaining the project's tests.

Insufficient resources lead to insufficient understanding of tests and their form and function. That, in turn, leads to brittle test suites that are hard to maintain and change — e.g., adding new tests, deleting old tests or updating existing tests.

A loss in understanding how the tests are structured or function can result in a host of challenges that make the test suite a liability and not an asset. For instance, engineers may not entirely understand a project's testing infrastructure, thus making it harder to add new tests, and not adding tests for new features can lead to untested code, which may lead to undetected bugs that are deployed to customers.

Engineers may not understand the functioning of an existing test, making it a challenge to update such tests. Existing tests might get outdated if they are not updated with an otherwise evolving codebase.

Poorly understood test logic may yield faults in the test logic itself; thereby giving rise to flaky tests that pass and fail non-deterministically. Flaky tests, with even modest levels of false positive rates, can undermine confidence in test results.

There may be insufficient understanding about how tests are distributed across different feature areas or levels of abstraction (*e.g.,* unit, integration, system). Consequently, uneven degrees of testing can cause over-testing in some areas, while foregoing testing entirely in others.

2.2.3 Challenge: Tools with local views offer inadequate comprehension

The structure and workings of tests are often tied to the code they verify. To truly understand how a test case functions, and its place in a larger test suite, it is worth examining the different components that the test suite ultimately executes, and verifies, and the degrees to which it does so. Examining such extents of a test's impact necessitates a global overview of a test suite and the code base that it helps to verify.

Current tools for developing software are overwhelmingly based on a developer's Integrated Development Environment (IDE). While IDEs improve developer productivity, they are based on a file-centric view of a software project; typically in terms of how the actual code is stored in the files and directories of a given operating system. Such a file-wise focus on tests and code makes it difficult for software developers to examine the test cases in the larger landscape of both the test suites that the tests are a part of and the code base that the test suite is meant to verify.

These challenges and motivations suggest that software comprehension can occur effectively when understanding both test and production code, as a collective unit, in their entirety. To that end, and to address the distinct challenge of scale with large test suites, I opt for a visualization-based approach.

Chapter 3

Morpheus Visualization: Comprehending Software Tests

One way to simultaneously comprehend test and product code is to trace and surface relations between them; like done in a test coverage matrix. A test coverage matrix (or more simply "test matrix"[1]) is a matrix in which test cases are on one dimension/axis and the program entities to be covered are on the other dimension/axis. Consider the simple test matrix shown in 3.1. In this example, I depict the test cases on the vertical axis and the program methods to be covered on the horizontal axis, that is, each row represents a single test case, and each column represents a single method. The cells in the intersection of the rows and columns represent whether the test case (row) covers (*i.e.,* executes) the method (column). For example, consider the first row that shows the coverage for Test 0. Test 0 covers Methods A, E, and H. Similarly, Method A was covered by (*i.e.,* executed by) Tests 0, 3, and 9.

Simple test matrices such as this can concisely represent the coverage of a program by its test suite. However, a standard test matrix would run into the information overload challenges

[1]In this work I use the terms "test coverage matrix" and "test matrix" interchangeably.

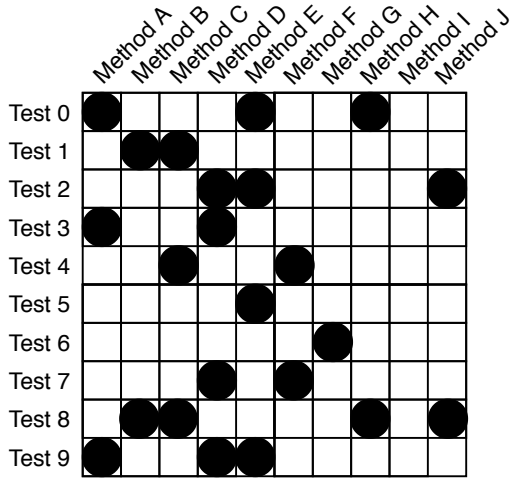


Figure 3.1: Example visualization of presenting tests and methods as rows and columns respectively; the intersection shows a colored dot if it was covered or not.

highlighted by Section 2.2.1, in the presence of too many test cases in a test suite.

To address challenges of scale and scope, I re-imagine the test coverage 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. By itself a test matrix is a static, unchanging data structure that can, indeed, reveal nuanced details of a project's testing efforts. However, test matrices for real-world test suites with thousands of test cases can be visually intractable. I refer to this interactive, visual rendition of the test matrix as the Morpheus Visualization.

In fact, test matrices have traditionally been limited to their use as educational aids. They serve to illustrate software testing in "toy", or contrived programs, when explaining concepts around software testing, such as spectra-based fault localization (*e.g.,* [30, 19]), test-suite minimization (*e.g.,* [34]), and regression-test selection (*e.g.,* [32, 33]). Beyond educational illustrations for "toy" programs, they scale poorly for meaningful, real-world test suites. Engineers would likely be overwhelmed when consuming data from fixed, globally scoped matrices that surface all test cases and their relationships to product code. The Morpheus Visualization expands on the academically-familiar, and static, test matrix with a series of

interactive capabilities that allow developers to alter how and which parts of the underlying test-to-code traceability data are presented.

Maletic *et al.* [25] proposed a "task-oriented framework for software visualizations" to help elucidate the function and purpose of software visualizations. I adopt this framework here to describe ours by defining the *Task*, *Representation*, *Target Data*, and *Audience* for MORPHEUS. MORPHEUS will enable software test comprehension (*task*), and by extension aid code comprehension generally, for stakeholders in a software project (*audience*). It will do so with a matrix-based, interactive visualization (*representation*) of the fine-grained relationships between test code and production code (*target data*). I will cover the final element of Maletic's task-oriented framework, *i.e., Medium*, in Chapter 4 when describing the web-application that implements MORPHEUS.

In the rest of this chapter I will breakdown the *target data* and *representation* elements of this visualization, and elaborate on the interactive capabilities of the visualization, as follows.

1. Artifacts along Rows and Columns
2. Overlays via Color and Hover
3. Juxtaposition of Data via Sorting
4. Drill-downs via Filtering

3.1 Artifacts along Rows and Columns

The rows and columns of the test matrix can represent multiple artifacts. For example, in Figure 3.1 rows represent test cases and columns represent program methods. However, this mapping is simply arbitrary and could equivalently be reversed (*i.e.,* methods on rows and test cases on columns).

Moreover, I envision that each of these dimensions can be parameterized to represent other artifacts. For example, one could have test cases on one axis and other granularities on the other axis:

1. Individual source-code lines;
2. Methods;
3. Source files; and
4. Modules or Packages;

To take this idea even further, I envision allowing each dimension to be used to represent a history of past versions. For example, if rows were configured to show test cases, and columns were configured to be used to show multiple versions of a particular method, then a user could see how the test-suite coverage evolved over time with regard to that method. As another example, if columns were configured to show methods, and rows were configured to show the history of a particular test case, then a user could see how the coverage of that test case evolved over time with regards to the methods it is covering.

3.2 Overlays using Color

Conveying information through the visualization can be done in various ways, so far I have explored ways to communicate information through positioning tests and methods along the axes.The visualization can be split into two components here: (1) the axis, presenting the methods and tests, and (2) the matrix where the connections between methods and tests are presented. For each of them color is approached potentially independently.

When coloring an individual axis, the Morpheus Visualization encodes within each node information relative to other nodes, *e.g.,* the package it belongs to. By presenting the package

the method or test belongs in color, it enables developers to observe patterns or potentially gaps within the test suite. While color is currently limited to the package a test or method belongs to, it is possible to encode other information in color. For example, encode the suspiciousness of a method in color as done by Tarantula [21], or in case of test cases the last time a test was changed.

Coloring of a node within the matrix encodes a relationship between a test and a method. The presence of a dot here, signifies that the method was covered by a test, while the color represents the passing or failing of a test case. Figure 3.2 shows the same test cases and methods of Figure 3.1, except that the nodes are colored according to whether its test case passed or failed.

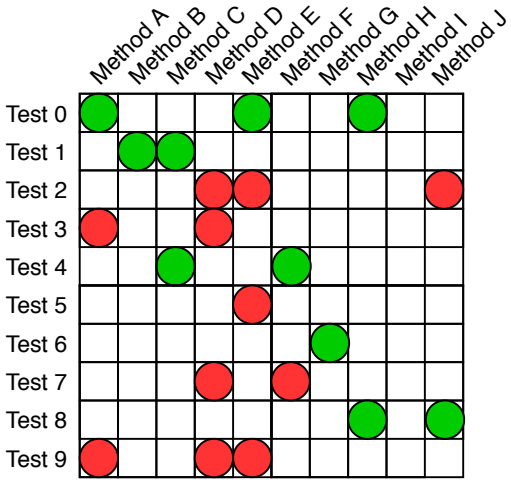


Figure 3.2: Example visualization of presenting tests and methods as rows and columns respectively; the intersection shows a colored dot if it was covered or not.

Finally, hovering over nodes are used to obtain more information than presented on the screen, but specific to the node being hovered over. At the moment this is limited to each axis to present the test or method signature (*i.e.,* package name, class name, method decleration).

3.3 Juxtaposition of Data via Sorting

Artifacts within a software project — tests, source lines, methods, files, packages — are typically not isolated from each other. There are latent dependencies snaking across them that tie all of them together — thereby making them a single software product.

As such, it would make sense if the artifacts represented along the rows and columns in a matrix can be shuffled around in a manner that juxtaposes artifacts that are related to each other. For instance, if different tests, which suppose are shown as rows, are testing the same method or package, it might make sense to place those test rows in the matrix next to each other. Scattering those test rows across 100's or even 1000's of other tests in the matrix does little to help identify patterns in how such tests are different or similar to each other.

As such, the visualization presents a variety of different functions to sort the rows and columns such that artifacts represented on those rows and columns can be bundled together with those they depend on, or are related to each other.

Specifically, I focus on the following sorting utilities 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 path 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 components (methods, lines, packages) by metrics such as coverage and suspiciousness, respectively

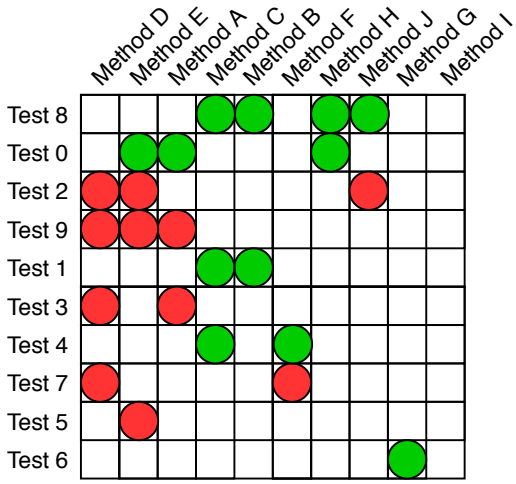


Figure 3.3: Sorted the test matrix (Figure 3.2) both axes by coverage.

To illustrate how such sorting could work, an example is presented in Figure 3.3 that builds further upon Figure 3.2. Each of the axis is sorted by coverage, methods often covered are presented on the left, while tests covering many methods are placed on the top. Such a sorting could be beneficial to reveal methods never or barely tested (such as Method G with only one test case that executes it, or Method I that is untested altogether), as well as to reveal test cases that execute a broad swath of the program (like an integration or system test), like Test 8, versus those that execute only a single method (like a unit test), like Tests 5 and 6.

Beyond these preset sorting capabilities, I envision to facilitate future developer extensions to provide custom sorting functions, and to plug them into this visualization to better comprehend their software project.

3.4 Drill-downs via Filtering

When trying to understand how a test suite helps in verifying product behavior, developers want to focus on specific tests at a time, instead of the whole test suite in one view. To

support such focused inspection, I provide the ability to filter down to tests or production artifacts that are of specific interest at any given moment.

As part of this work, I present and evaluate the following filters on a test matrix that directly aid in improved comprehension of a project's testing:

Filter by Name This allows developers to simply filter down to one or many test, method or module by their name.

Filter to a Cluster This leverages the sorting and clustering capabilities that I enumerated in the previous section, allowing developers to not just cluster their tests and production artifacts, but also inspect such clusters one at a time;

Filter Production Artifacts by test levels 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 Changes to Production Artifacts This helps focus on changes happening in a project and the ensuing evolution in how those changes are best managed by their tests.

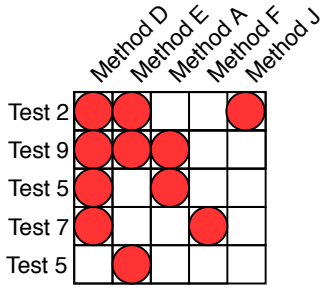


Figure 3.4: Filtered the sorted test matrix (Figure 3.3) to only failing test cases and methods covered by them.

In Figure 3.4, an example is presented of how filtering by failing test cases works when applied on the test matrix presented before (Figure 3.3). The order of methods and tests

along the axis are maintained, but I filter passing test cases out and simultaneously remove methods that are not covered anymore. Such filtering may be useful for assisting during debugging to identify the methods that may contain the bugs that are causing the test case failures.

It is worth noting that when filtering, I apply a given filter either along the rows or columns, but not both. For instance, when I filter down to a group of tests shown as rows, I simply do not filter out the columns that the filtered tests have no relationship to. Doing so allows us to maintain context for the filtered tests within the larger scope of the project.

Chapter 4

Implementation

The implementation of MORPHEUS is comprised of three components: (1) the visualization, (2) server with RESTful API to serve the data to the visualization, and (3) the Dockerized analysis environment allowing reproduction of each run. The main component users will interact with is the front-end as presented in Section 3. In this section I focus on the implementation of the analysis framework including the database scheme (Section 4.1) and the architecture of how I serve the content to the users (Section 4.2).

4.1 Data Collection

Creating traceability between tests and methods requires two sources of data, (1) per-test-case code coverage, and (2) line ranges of the entities I want to create traceability between, *i.e.,* method-line ranges. In Figure 4.1 I describe the process for collecting the per-test-case code coverage and the method-line ranges. The first step is to clone the project followed by creating a build environment for that specific commit based on data in the repository, *e.g.,* JDK version provided in configuration files. After compiling the production and test

code, the parsing of the methods and obtaining coverage information of the system can be performed.

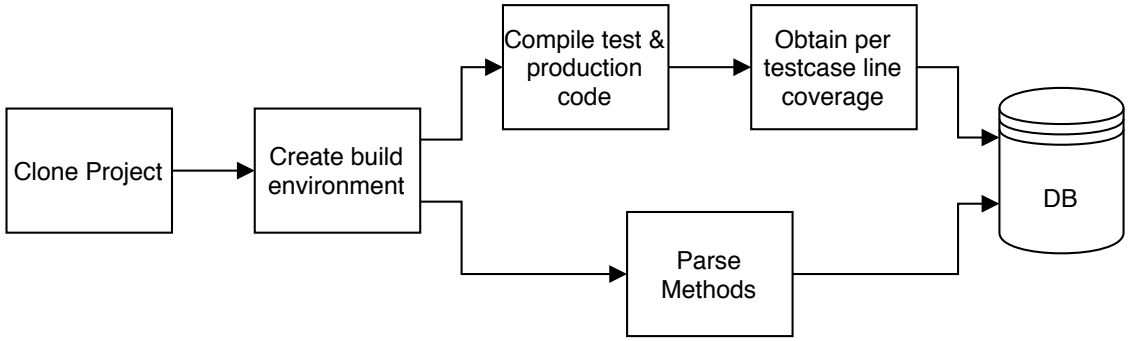


Figure 4.1: Data collection steps

4.1.1 Collecting Per-Test-Case Coverage

TACOCO [22], a software-analysis framework that allows instrumentation of individual test cases, provides per-test-case code coverage. In this scenario, each test case was instrumented using JACOCO[1], *i.e.,* a code coverage tool. TACOCO discovers the compiled tests within a project, and determines what test runner is necessary to run them by extending the JUnit platform.

When TACOCO has determined which test runner to use, it is ready to start running the tests. Based on events generated by the JUnit Platform, it becomes possible for TACOCO to determine if (1) a tests is about to start, (2) a test concluded successfully, or (3) a test concluded with a failure. Using this information JACOCO can be instructed to start or stop tracking the coverage. As a result, TACOCO can obtain per-test-case coverage, which is unique for each project and commit pair.

[1] JACOCO: `https://www.eclemma.org/jacoco/`

4.1.2 Obtain Method Line Ranges

The code coverage is collected at line-level granularity, allowing us to create traceability between tests and entities at different levels of granularity. For MORPHEUS, I focus on method-level granularity, meaning I need to obtain: (1) all method signatures (*i.e.,* package name, classname, and method declaration) within the system, and (2) collect the line range for that specific commit. Due to methods changing over time, the line ranges of a specific method are not constant. Therefore, the methods' signatures are not bound to one specific commit, while the line range for a specific method (or versions of the method) *is* specific to a single commit.

By having line ranges tied to a specific commit, but not method signatures, the system gives the option to track coverage of a single method through history, and consequently, I can track a specific method's tests coverage through time.

4.1.3 Containerize Analysis

While TACOCO is able to discover the tests in the system and run them, it is not capable of compiling the system under study itself. As a result, the burden of compiling a system and making it runnable is put on us. To increase the chance of successfully building the project, I analyze the configuration files in the project, *e.g.,* the *pom* file for maven projects, to determine the specified jdk. This information is used to create a docker container that closely matches the intended build environment. The advantage is twofold, (1) creating an environment similar to the intended build environment increases the chance of successfully building the system and thus obtaining the coverage data, and (2) it makes the study more reproducible because the build environment is automatically generated.

4.2 Architecture

In Figure 4.2, the system's architecture is presented. I make use of a client server architecture, where the client would be a browser where the visualization is rendered. The backend consists of a proxy-server (*i.e.,* nginx) that routes the traffic based on hostname to either the visualization or the API to access the per-test-case coverage.

The API is a RESTful API to access the content stored in the database. Due to the nature of a RESTful API all the responses to a GET request to the same URI should be the same, making it possible to cache the content for reuse, thereby improving the performance of the backend.

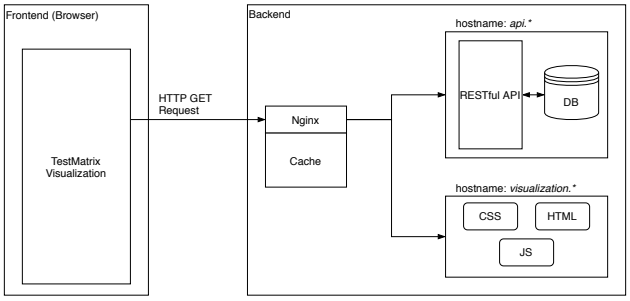


Figure 4.2: Architecture of system.

4.3 Morpheus Visualization: Implementation

In Figure 4.3, the implementation is presented and consists of two parts: (1) the test matrix visualization, and (2) the toolbar. The visualization shows the connections between methods and tests, while the toolbar provides a set of ways to filter and sort the data.

Morpheus Visualization, as explained in Section 3, was implemented as a web-based HTML5 application built using React and D3.js [3]. React was used to build the interface and to determine when, and if, the visualization needs to be updated. The test matrix

visualization is completely implemented in D3.JS. Since the visualization is built using web-based standards, it works with any modern web-browser supporting HTML5.

Figure 4.3: MORPHEUS VISUALIZATION User Interface.

4.3.1 Morpheus Visualization: Interactions

When using the visualization, developers will interact it in various ways. Interacting with MORPHEUS VISUALIZATION can be done in three ways: (1) filtering or sorting using the toolbar; (2) filtering by clicking on a test or method on one of the axis; and (3) hovering over a method or test to obtain the method name.

A user can use the toolbar to filter, sort, or search for specific methods or tests. Giving them fine-grained control over what they want to see. Sorting can be applied separately on both axis, *e.g.,* the tests are sorted by coverage, while methods are suspiciousness.

Filtering, can be done based on multiple properties of a test or method. Tests can be filtered by the following properties:

- Test type, *i.e.,* unit, integration, or system test;

- Test result, *i.e.,* passed or failed;
- Coverage of a test case, *i.e.,* number of methods covered by test;

Moreover, methods can be filtered based on the amount of tests are covering that method.

An extension of filtering Morpheus Visualization provides is 'searching' a specific test or method. Search filters the tests down to all tests covering at least one of the methods covered by the selected test case. Similarly, when searching for a method it presents all methods that are co-tested by at least one of the tests covering the selected method.

While the toolbar is useful for creating specific situation, it is not as well equipped for exploring the visualization. To overcome this challenge, one can filter by clicking on a method or test to obtain a more close up view. For example, when a user user clicks on a method they only see the test cases that cover that method, while giving context presenting methods that are also covered by those tests.

To alleviate the information overload by presenting all method labels along the axis, it was decided to hide them and only present them upon hovering over the node. Allowing developers to gain more information on-demand and keeping the overview, while limiting the information overload.

Finally, when developers explore the test suite Morpheus maintains a history of previous states of the filters. Consequently, enabling developers to move back to a previous view, and encourage developers to explore, because previous states are maintained.

4.3.2 Morpheus Visualization: Color

Morpheus Visualization has two main components that allow communication of information through color: (1) the axis presenting the tests and methods; and (2) the nodes on

the intersections of tests and the methods they cover. In my implementation, the color of the nodes on the axes present the package they belong to, allowing users to observe methods that are part of the same package. It is possible to extend this functionality by coloring based on different properties. For example, one could color the methods by suspiciousness [19], or the test cases based on test result, *i.e.,* passed or failed.

The intersections between tests and methods is provided a color based on the test result. Moreover, it is again possible to modify this to properties from both the test or methods that are intersecting. For example, one could color the node based on suspiciousness of the method being covered.

Finally, to show the user by which test case or method they are filtering by, Morpheus Visualization highlights the chosen nodes by presenting a thin boundary around it. As a result, make it possible for developers to maintain overview of what method they are filtering by.

Chapter 5

Case Studies

To illustrate the effectiveness of Morpheus, I applied Morpheus on a set of four open-source Java projects (see Table 5.1). For this case, I focus on three usage scenarios:

- Exploration (Section 5.1)
- Test Failure Comprehension (Section 5.2)
- Inter-Project Test Suite Patterns (Section 5.3)

Table 5.1: Projects used in the case study.

Project	Description
Commons-CLI	A command line options parser library
Commons-IO	Utility library for IO functionality
Jsoup	XML parser library
Maven	Build system

In this paper, I use the following definitions for types of tests:

- **Unit**: Tests that cover methods within a single class.
- **Integration** Tests that cover methods across multiple classes in a single package.

- **System**: Tests that cover multiple methods which reside in at least two different packages.

5.1 Test Suite Composition

Motivation. A developer wants to start contributing to a new project. Before starting to contribute, the developer wants to get a general understanding of the test suite's composition to better understand how well tested a specific method is, e.g., what kinds of test cases are testing a specific method?

Approach. Comprehending the test suite composition is a difficult task. On top of that, there isn't always a clear division between the types of tests in the project. Consequently, it is unclear for developers what is covered by which tests and what is covered together. This may lead to not understanding their test suite properly, resulting in tests not becoming an asset to a developer.

In Figure 5.1 I present the test suite of Commons-CLI. The initial view of the test matrix gives us an overview of all the methods (x-axis) and tests (y-axis). The tests and methods are both sorted by default, based on first the package, second the class, and third the method or test name. Just based on this view, I come to two observations: (1) most tests seem to execute many methods, and (2) most tests seem to be variations of one another.

A developer would be interested in the composition of the test suite, specifically: (1) what is (not) tested, and (2) what types of tests are present in the test suite. The former can be determined by sorting the methods and tests by their coverage, *i.e.,* by how many tests is a method is covered, see Figure 5.2. It directly 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

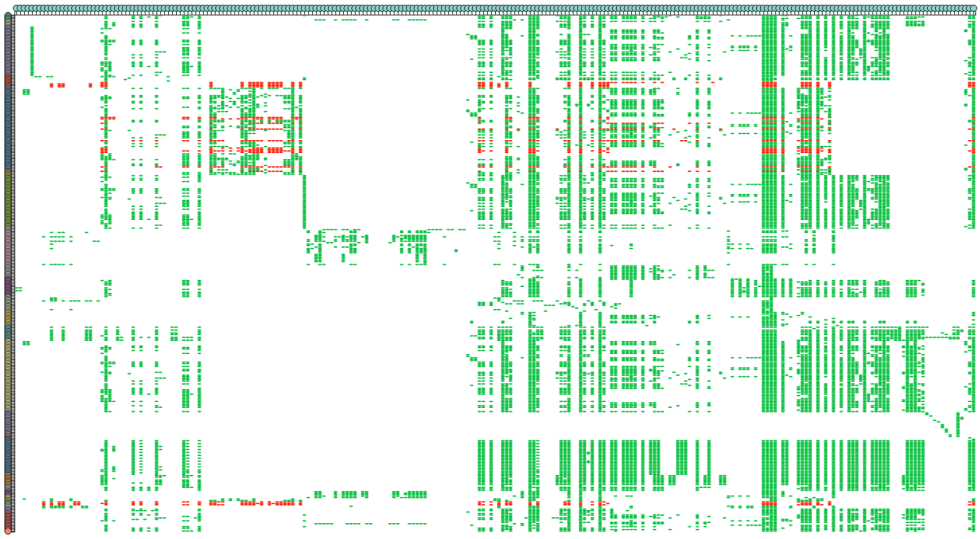
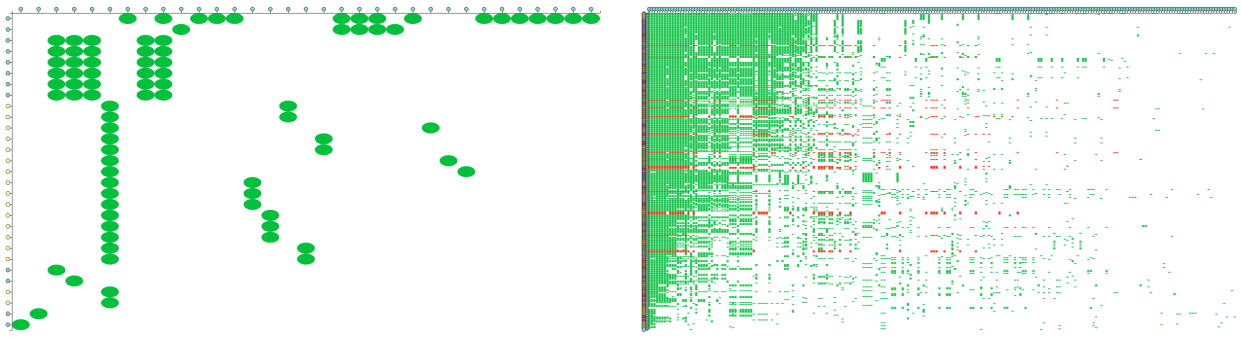


Figure 5.1: COMMONS-CLI. All tests and methods sorted by name.

can use this view to directly take steps on where the test suite can be improved.

Figure 5.2: COMMONS-CLI. All tests and methods sorted by coverage.

Next, MORPHEUS provides ways to filter based on the type of test cases, *i.e.,* unit, integration, or system test. In Figure 5.3, I show the filtered result for each type of test. COMMONS-CLI tests are primarily written as integration tests with limited amounts of unit tests (29 unit tests vs. 326 integration tests). There are no system tests within this system due to all classes residing in the same package; thus the figure for system tests was omitted.



(a) Unit tests. (b) Integration tests.

Figure 5.3: COMMONS-CLI testsuite composition filtered by type of tests.

MORPHEUS provides multiple ways to obtain a better understanding of the test suite composition, allowing you to sort to see what methods are covered, and what is not, and filtering to show the different types of tests available. Combining the two allows us to explore what is covered, what is tested together, and what types of tests are in the test suite.

5.2 Test Failure Comprehension

Motivation. A developer notices a set of test cases are failing and that the same method is covered each time. As a starting point, the developer debugs that method to determine if the problem lies there, however, that method doesn't seem to be the problem. To determine what set of methods to look at next, the developer now wants to know what set of methods are also executed by the same failing test cases.

Approach. Finding a starting point to debug a set of failing test cases or even a single test case can be challenging. MORPHEUS helps the developers by presenting using red dots to represent the intersections of test cases that fail and the methods it. While this gives an overview of what is covered in a failing test case, it still presents a voluminous amount of information. To help developers gain better insight into which tests are failing together and

what methods are involved, MORPHEUS provides multiple ways to filter and sort the data.

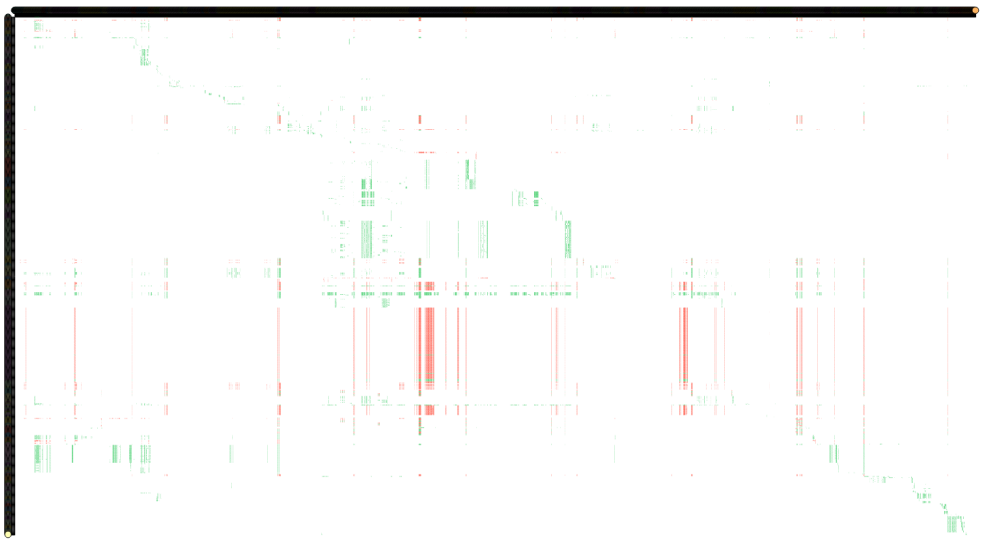


Figure 5.4: MAVEN. All tests and methods sorted by name.

One way of distilling the information down is by filtering the methods and tests. In Figure 5.4 I show the overview of all the tests and methods being covered in MAVEN. Due to the amount of methods and tests in the system, it becomes difficult to extract any information out of it. To alleviate this problem, I filter to only the relevant tests and methods, *i.e.,* (1) show only failing test cases, and (2) the methods covered by those failing test cases. To accomplish this, two filters are applied. First, only failing test cases are presented, considerably limiting the amount of tests to be observe, but this alone is not enough. All the methods are still presented, both which are and are not covered by the presented tests. Second, to filter further, only methods covered by the failing tests are presented. The set of methods are filtered to only methods that are at least tested once by the tests currently presented, see Figure 5.5. Filtering gives us a subset of relevant tests and methods to look at, and furthermore, it shows us that most tests cover a similar set of tests with some deviations.

Based on the current information, the developers have a set of methods, and they know for each method which tests cover it. However, this may still be a large set of methods, so to prioritize the methods, a suspiciousness score [20] is computed for each method. This

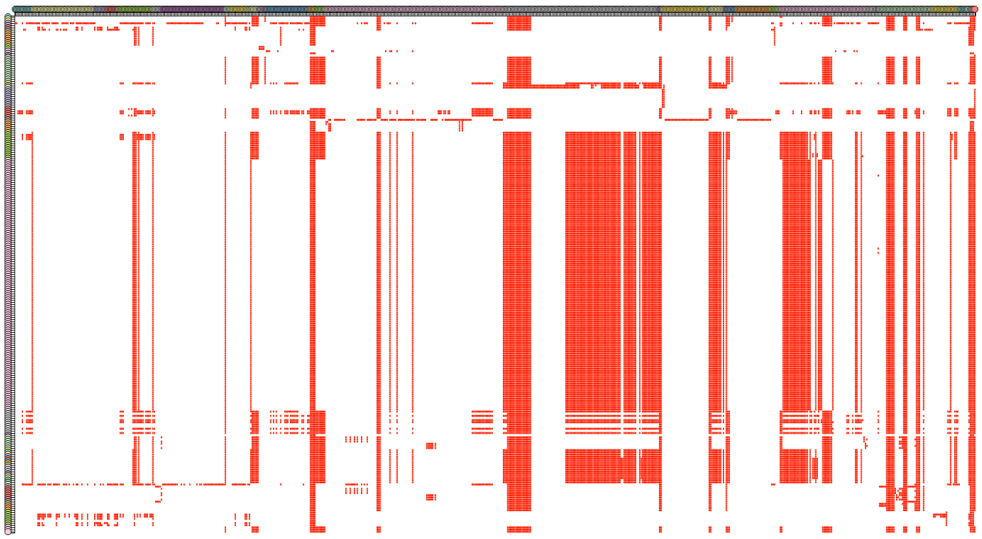


Figure 5.5: MAVEN. Failing test cases and the methods covered by it.

organizes the methods in a way that can help developers choose which methods to start looking at, see Figure 5.6.

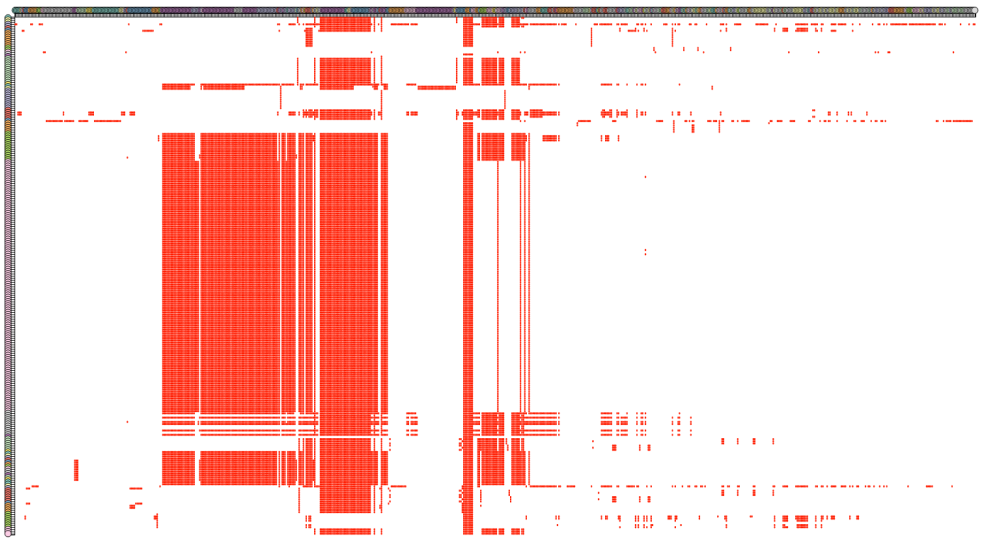


Figure 5.6: MAVEN. Sorted methods by suspiciousness and tests by name.

Once a developer has chosen the method to look at, it may also be of interest to determine which test has covered the fewest entities to assist debugging the method. As shown in the previous section, MORPHEUS allows sorting the test axis based on the amount of methods it covers. As seen in Figure 5.7, the top of the screen presents the tests with the most covered

entities, and at the bottom, I present tests with the smallest amount of covered entities.

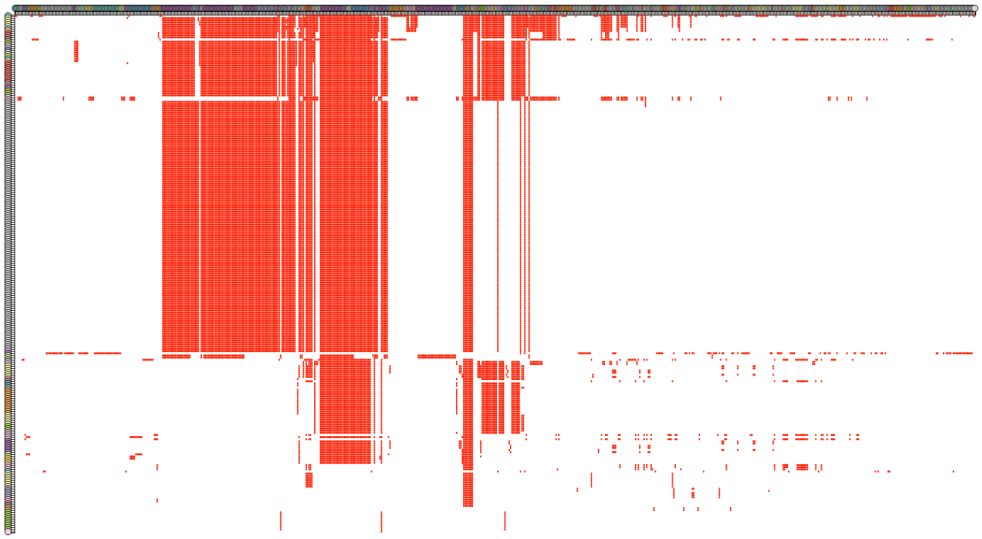
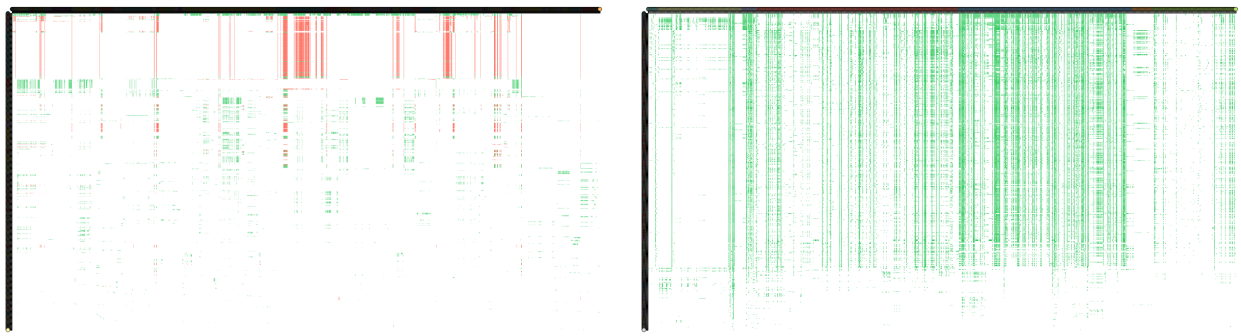


Figure 5.7: Maven: Sorted methods by suspiciousness and tests by coverage.

Finally, MORPHEUS can assist in determining: (1) which methods are covered by failing tests, (2) methods are most likely to be the cause of the failure, and (3) which test case covers the least amount of entities to assist in debugging the failure.

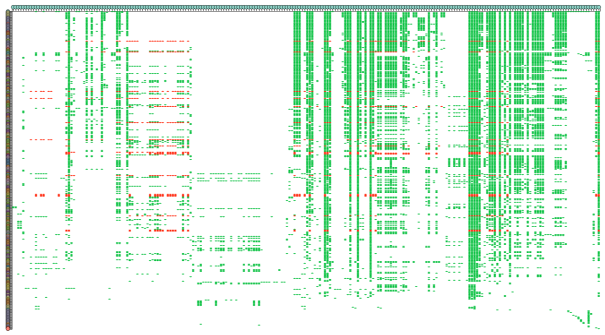
5.3 Inter-Project Test Suite Patterns

So far when looking at projects, I have focused on a single test suite's form and function. In this case study, I compare and contrast visualizations from multiple projects 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 the degree to which their projects are tested, and if the type of testing is appropriate for their type of program. For example, one may expect that a utility library (*i.e.,* API) may be largely comprised of unit tests — each method in the library performs some function that can be independently tested with very few 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

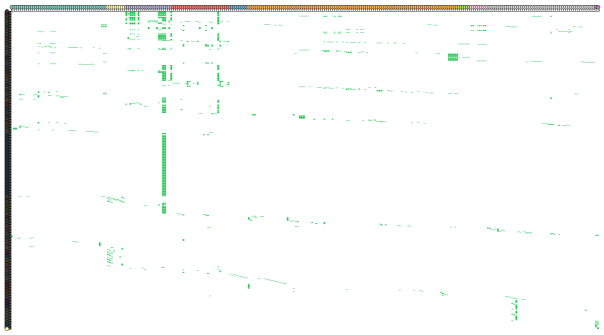


(a) Maven.

(b) Jsoup.



(c) Commons-CLI.



(d) Commons-IO.

Figure 5.8: A set of three projects filtered by all methods covered by at least one test and sorted by name for methods and coverage for tests.

to unit tests, because the system itself relies upon multiple interacting components to perform its functionality. In Figure 5.8, I present Morpheus for four projects, *i.e.,* Maven, Jsoup, Commons-CLI, and Commons-IO.

When looking at the visualizations, there are two aspects that become 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. I see this happen mainly with Jsoup (Figure 5.8b), and Commons-CLI (Figure 5.8c), 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

Table 5.2: Distribution of type of tests (in percentage).

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

of each other covering different corner cases each time.

To test some parts the Commons-CLI library the 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.

The second aspect of the visualization to note is the difference in sparseness between the visualizations. Both Jsoup and Commons-CLI are more dense visualizations, while Maven and Commons-IO are more sparse. This can be attributed, in part, to the composition of the test suites. Table 5.2 shows us the distribution tests within the four projects. Jsoup consists 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 I mentioned before, Maven also has those long vertical lines, but not as much, which one can see back in its test distribution which is more spread out 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 my expectation for a utility library that contains little inter-method dependencies.

In conclusion, Morpheus is able to show us some patterns that one can see across projects, *e.g.,* composition of tests, and how developers test their system. The sparseness can give an indication of the present of unit, integration, and/or system tests, while the vertical lines can point to commonly used methods.

Chapter 6

User Study

I performed a user study with 11 participants to evaluate the effectiveness of Morpheus for three software-maintenance tasks. Each of the participants have a minimum of 1 year of Java software-development experience. Table 6.1 presents an overview of the average amount of experience (in years) across all participants. Similarly, I report the distribution of occupations of all participants in Table 6.2. Nine participants are Software Engineering (SE) graduate students, one is a SE undergraduate student, and one is a software test engineer.

Table 6.1: Average experience (in years) of the participants.

Description	Experience
Years of Java experience	2.5 years
Years of software development experience	3 years
Years of Object Oriented Programming Experience	2.6 years
Software Testing Experience	1.8 years

Table 6.2: Participants occupation

Occupation	Participants
Undergraduate student	1
Graduate students	9
Software Test Engineer	1

6.1 Overview and Research Questions

The study consisted of two rounds. In both rounds, a participant performed three tasks, once with their own IDE or toolset of choice, and once with Morpheus. When using their own IDE, the participants were allowed to make use of any tool in their developer toolkit, and they were asked to report on all the tools they used. In Table 6.3, I am reporting the range of tools used by the participants of the user study. Table 6.3 shows that most participants used Intellij, and one participant even used terminal (command-line) tools such as Vim and grep. Each participant also received assistance in setting up the project so they could at least run the test suite and obtain test coverage information using Jacoco[1].

Table 6.3: Tools that participants reported using for performing IDE user study.

Tool	Often used
IntelliJ	7
VS Code	1
Eclipse	1
vim	1
grep	1
Navigate Test and Source Code[2]	1

Finally, the users got a brief hands-on training before starting with the visualization. During the training they were presented with: (1) how the tool works, (2) the features it contains, and (3) how to use the tool, all on a different and smaller program than the one used during the experiment.

In both rounds, the students were given the same program and same type of tasks, but asked to identify different entities (test cases and methods). For example, if a participant would be asked to identify all integrations tests for method A during the IDE round, then for the visualization round, the participant would be asked to identify all integrations tests for method B. The group of participants was split into two groups — one group would perform

[1] Jacoco: `https://www.eclemma.org/jacoco/`

[2] "Navigate Test and Source Code" is an advanced feature in the IntelliJ that allows a developer to automatically locate tests for a class, based on the test and class names.

the tasks using method A for the IDE and method B for the visualization; the other group was asked to do the opposite to avoid any bias due to one method potentially making the task more difficult.

The performance of each participant is measured in two ways: (1) the time it takes the participant to complete each task (within a 5-minute time limit per task), and (2) the correctness of the given answer to each question.

The goal is to evaluate the effectiveness of different facets of the visualization. In particular, I focus on the following research questions:

RQ1 Can the test-matrix view provide insights into the composition of the test suite?

RQ2 Can a test matrix view provide traceability between test and production code?

RQ3 Can the visualization help identify sets of methods that fail together (*i.e.,* executed by the same failing test cases)?

6.1.1 Task 1: Distinguish different types of tests that cover a specific method

Scenario. A developer wants to start contributing to a new project. Before starting to contribute, the developer wants a general understanding of the test suite's composition to better understand how well tested a specific method is, *e.g.,* what kinds of test cases are testing a specific method?

The participants' final tasks focus on understanding how the test suite is built up. Developers are encouraged to write multiple types of tests, e.g., unit, integration, and system tests. Having a good understanding of which parts of the system are covered by what kind of tests can give insight into where there are gaps in the test suite.

The concrete questions that were asked of participants for this task were:

Task 1.1	Group A	List the set of unit tests (*i.e.,* only tests that cover methods within a single class) are there for method "org.apache.commons.cli.HelpFormatter int findWrapPos(String, int, int)"
	Group B	List the set of unit tests (*i.e.,* only tests that cover methods within a single class) are there for method "org.apache.commons.cli.HelpFormatter StringBuffer renderWrappedText(StringBuffer, int, int, String)"
Task 1.2	Group A	List the set of integration tests (*i.e.,* only tests that cover methods across multiple classes in a single package) are there for method "org.apache.commons.cli.HelpFormatter int findWrapPos(String, int, int)"
	Group B	List the set of integration tests (*i.e.,* only tests that cover methods across multiple classes in a single package) are there for method "org.apache.commons.cli.HelpFormatter StringBuffer renderWrappedText(StringBuffer, int, int, String)"

6.1.2 Task 2: Locate all tests that cover a specific method

Scenario. A developer wants to refactor an existing method. Before starting, the developer is interested in determining which tests are covering this method to assure that the refactoring will not cause any regression.

During development, developers are often interested in changing their code and then re-running their tests to determine if the change caused any regression. Currently however, there is limited support within an IDE to (visually) inspect which test case covers which set of methods and what kind of tests, *i.e.,* unit, integration, or system tests. Many IDEs have code-coverage tools, but it provides a view from the production code perspective, showing

which lines are covered, but not by which tests.

The concrete questions that were asked of participants for this task were:

Task 2	Group A	List the set of tests that cover the following method: "org.apache.commons.cli.HelpFormatter String getOptPrefix()"
	Group B	List the set of tests that cover the following method: "org.apache.commons.cli.MissingOptionException List getMissingOptions()"

6.1.3 Task 3: Locate all methods that are co-failing within a specific method

Scenario. A developer noticed a set of test cases are failing and noticed a method was covered each time. As a starting point, the developer debugged that method to determine if the problem lies there. However, that method does not seem to be the problem. The developer now wants to know what set of methods are also executed by the same failing test cases to determine what set of methods to look at next.

Just like in Task 1, developers rely on the test suite to alert them when there is a regression. However, after a set of test cases fail, developers become interested in which methods failed and which failed together (*i.e.,* executed by the same failing test cases), as co-failing methods may be an indication of the source of failure.

The concrete questions that were asked of participants for this task were:

Task 3.1	Group A	List the set of methods that are also executed by the one or more of the same failing test cases that execute "org.apache.commons.cli.Option void setArgName(String)"
	Group B	List the set of methods that are also executed by the one or more of the same failing test cases that execute "org.apache.commons.cli.Option void setValueSeparator(char)"
Task 3.2	Group A	List the set of failing tests that are testing "org.apache.commons.cli.Option void setArgName(String)"
	Group B	List the set of failing tests that are testing "org.apache.commons.cli.Option void setValueSeparator(char)" and its co-failing methods

6.2 Results

In Table 6.4, I present the mean results for precision, recall, and the F-score for each task performed by the users. Mean values for precision, recall, and f-score values for each task are tabulated in their own row. I report mean scores for both the IDE and visualization tasks. Figure 6.1, 6.2, and 6.3 show the boxplots for the precision, recall, and f-score, respectively, for each task. Additionally, I compute Precision, Recall, and F-score as follows:

$$\text{Precision} = \frac{\#\text{Correct Answers}}{(\#\text{Correct Answers}) + (\#\text{Incorrect Answers})} \tag{6.1}$$

$$\text{Recall} = \frac{\#\text{Correct Answers}}{(\#\text{Correct Answers}) + (\#\text{Correct Answers Not Given})} \tag{6.2}$$

$$\text{F-score} = 2 \times \frac{(\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})} \tag{6.3}$$

For Equations 6.1 and 6.2, "Answers" are provided by the study participants in the form of a set of methods or test cases. For example, for Task 2, 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.

The row for Task 1.1 in Table 6.4 suggests perfect precision and recall scores of 1.0 for the Visualization tasks. Whereas, for the tasks performed with the IDE the (mean) precision score stands at 0.375, and (mean) recall score of 0.247. Indeed, notice this trend continues throughout all tasks. The participants consistently performed better using the visualization in comparison using their own development environment.

Table 6.4: Mean precision, recall, and f-score results for each task.

Tasks	Precision		Recall		F-Score	
	IDE	Visualization	IDE	Visualization	IDE	Visualization
Task 1.1	0.3750	1.0000	0.2470	1.0000	0.2699	1.0000
Task 1.2	0.1250	0.8750	0.0057	0.8750	0.0109	0.8750
Task 2	0.6364	1.0000	0.0627	0.9924	0.1116	0.9960
Task 3.1	0.2424	1.0000	0.0059	0.9847	0.0115	0.9916
Task 3.2	0.1061	0.8182	0.0115	0.8182	0.0068	0.8182

It is worth noting that for the IDE tasks, the mean precision scores are typically higher than the recall scores. The low mean precision scores suggest that in the IDE tasks, when trying to report the correct set of methods or test cases for each task, the users consistently reported the incorrect set of methods and test cases. Moreover, the even lower mean recall scores indicate that they left unreported many more methods and test cases, than the few methods and tests reported correctly.

Table 6.5: Pairwise t-test p-value results.

Task	p-value$_{Precision}$	p-value$_{Recall}$	p-value$_{F-Score}$
Task 1.1	0.0112	0.0020	0.0020
Task 1.2	0.0025	0.0002	0.0020
Task 2	0.0379	0.0000	0.0000
Task 3.1	0.0001	0.0000	0.0000
Task 3.2	0.0017	0.0001	0.0001

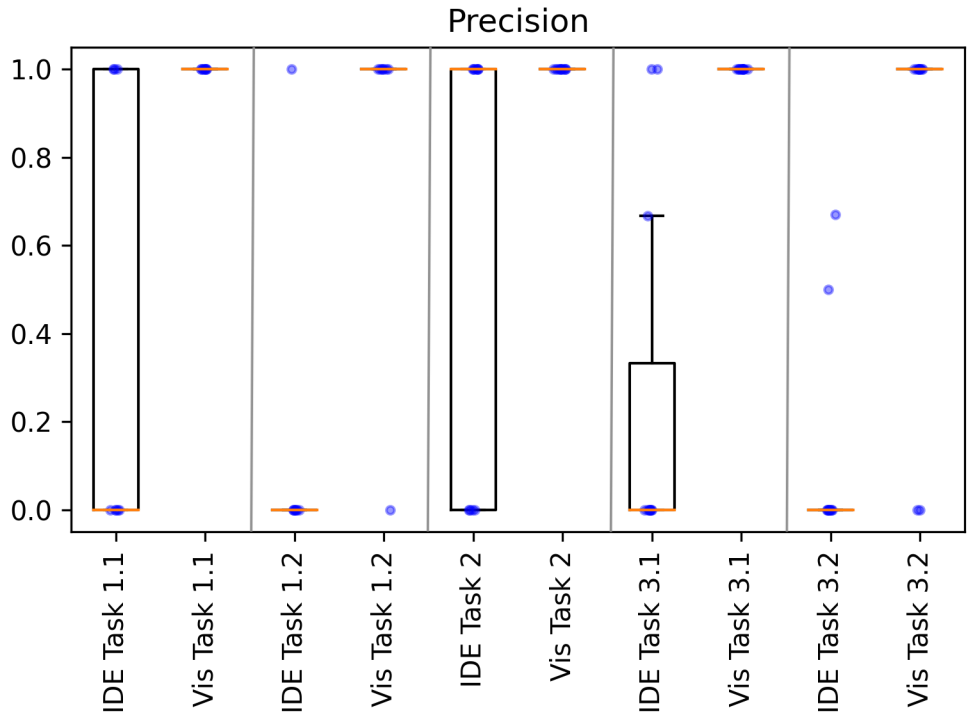


Figure 6.1: Boxplots for the precision of each task and tool.

I further analyzed the results on statistical significance using a pairwise t-test. Table 6.5 presents those results and show that based on the F-score, all results are significant in difference, however, for precision and recall the results are not always significant. It should be noted that the Anderson-Darling test fails to show normality of the data, and as such the t-test can not be reliably used in this context. I, nevertheless, include the t-test results here as a simple indication of the degree of difference between these two sets of results.

Table 6.6: Average time (in seconds) taken by participants per task.

Tasks	Time (in seconds)	
	IDE	Visualization
Tasks 1.1 & 1.2	244	197
Task 2	153	78
Tasks 3.1 & 3.2	199	115

In Table 6.6, I present the average time (in seconds) a participant took to finish each task. In Figure 6.4, a boxplot of the timing results per task and for each tool is given. During

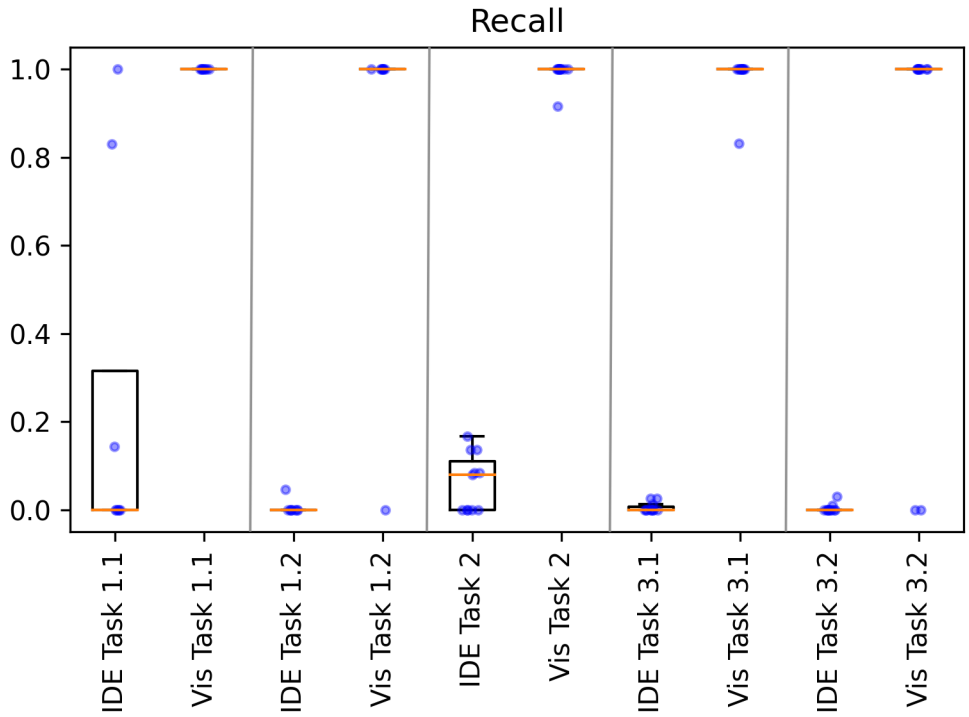


Figure 6.2: Boxplots for the recall of each task and tool.

the IDE round, the participants made use of a variety of tools to get to an answer. All participants extensively made use of the search functionality within the IDE, while some used *grep* on the whole repository to locate relevant files. One participant made use of an IntelliJ IDE feature allowing you to navigate between a test and the subject under test, this connection is based on the test method name and class under test.

Finally, at the end of each round, the participants were asked to provide a score between 1–10 of their satisfaction with the tools they used. In Figure 6.5 a boxplot of the satisfaction results is given. On average, developers gave their own development tools a rating of 4.6, while the Morpheus tool was given a rating of 8.8.

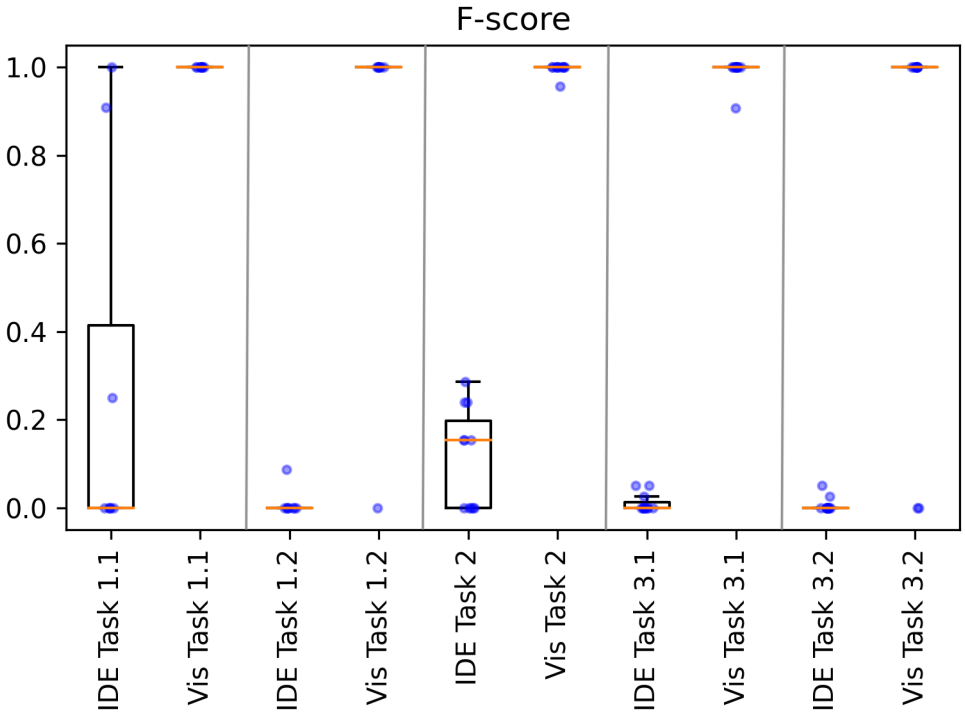


Figure 6.3: Boxplots for the f-score of each task and tool.

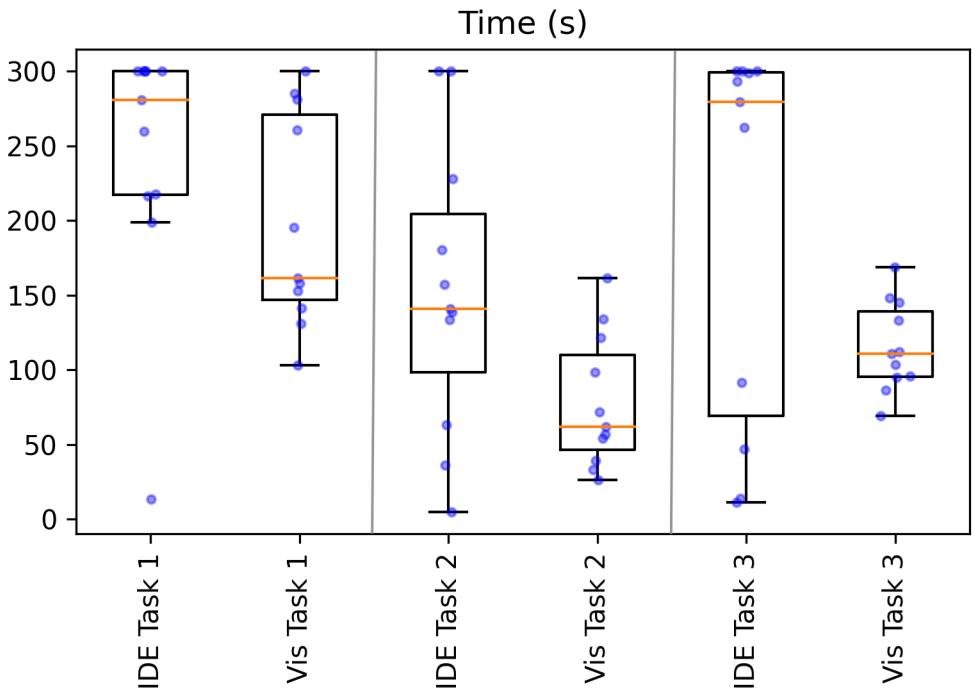


Figure 6.4: Boxplots for the time (seconds) a participant took to complete each task separated per tool.

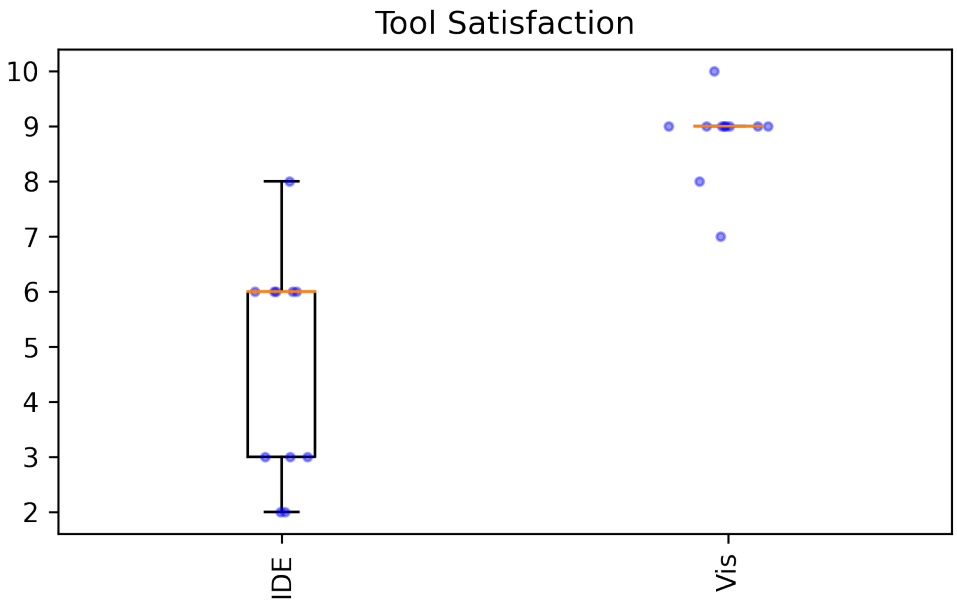


Figure 6.5: Boxplots for the tool's satisfaction scores; IDE is the left satisfaction score and Morpheus is the right satisfaction score.

Chapter 7

Discussion

In this section I discuss the results with the goal to answer the research questions proposed in Chapter 6 and thus addressing the challenges proposed in Chapter 2.

7.1 Challenge: Large test suites with many tests

As presented in Section 2.2.1, before a test suite can become an asset, it has to be understood, meaning (1) a developer needs to be able to know what a test is testing, and conversely (2) by what tests a method is covered? To address this challenge, I introduce **RQ2** to see if Morpheus is able to provide traceability between test and production code.

In the *Test Failure Comprehension* case study (Section 5.2), I show that the visualization is able to communicate in two ways: (1) it is able to show which test cases are covering a specific method and vice versa; and (2) I am able to show, given a method, which other methods it is co-executed with.

The results from the user study also confirm that the test-matrix view is able to present the

connections between tests and methods well. In Table 6.4, one can see that Task 2 (*i.e.,* locate all tests covering a specific method) is performed with a higher precision, recall, and f-score on average. The significance of the results are confirmed in Table 6.5, where it shows that for both tests the mean difference is statistically significant.

As such, MORPHEUS is able to provide traceability between test and production code. Allowing developers to get on top of a high-level overview of the test suite also an in-depth view of the connections between tests and methods.

7.2 Challenge: Maintaining test suites

As presented in Section 2.2.2, maintaining a large-scale system requires being able to trace what is covered and also by what kind of tests. Thus **RQ1** was introduced to see if MORPHEUS is capable of providing insights into the composition of the test suite. In the *Test Suite Composition* case study (Section 5.1), it was presented how MORPHEUS can be used to see what is and is not tested, and as an extension of that what parts of the system are tested together. It also shows, that is is possible to see what kinds of tests are present within the system.

Similarly, one can see in the *Inter-Project Test Suite Patterns* case study (Section 5.3), how the MORPHEUS can give insight into the types of tests being used, but also help identify commonly used methods within the test suite.

The case study's findings are confirmed by the user-study findings. In Table 6.4 one can see that tasks focused on locating types of tests, *i.e.,* Task 1.1 (locating unit tests) and Task 1.2 (locating integration tests), are performed with a higher precision, recall, and f-score on average. The significance of the results are confirmed in Table 6.5 where it shows that for both tests the mean difference is statistically significant. As a result, one can see that

developers are better equipped to locate different types of tests for a given method using Morpheus than using their own development environment.

Most of the time participants were struggling with locating tests that were testing a method transitively (for example, Test Case A directly calls Method X, which then calls Method Y). Approaches that were used to counter these problems were building a mental static call graph or stepping through tests with a debugger. However, these are all manual, and as a result, inherently error-prone processes. Furthermore, due to the dynamic nature of some of the systems, only focusing on the static code will not allow you to get the full picture. Morpheus improves on this, by capturing the dynamic information and doing so all automatically, allowing developers to focus on the task at hand.

In conclusion, Morpheus allows developers to more effectively answer questions regarding the composition of the test suite. They are able to determine with a high precision and recall how many unit, integration, and system tests are available. Also, the user study shows that the visualization can show what each of those tests cover, and what is covered often and what not.

7.3 Challenge: Tools with local views offer inadequate comprehension

As presented in Section 2.2.3, IDEs provide a local view, which limits understandability of the test suite, especially if they test multiple components. Thus **RQ3** was introduced to see if Morpheus is capable helping to identify sets of methods that fail together (*i.e.,* executed by the same failing test cases). To accomplish this, a developer would need to step outside of just a local view and look at the bigger picture.

In the case study presented in Section 5.2, I show how in a few steps one can morph the matrix view using filtering and sorting to only show the failing test cases and the methods that they are covering. As can be seen in Figure 5.5, one can quickly locate all methods failing together in combination with the test cases covering them. On top of that, I provide ways to sort the data to sort the methods by *suspiciousness* giving developers a starting point to take action.

During the user study the participants were able to locate all test cases that were failing, and the methods that were executed. Table 6.4 shows that on average Task 3.1 (*i.e.,* determining methods are covered by failing methods and were covered together with a given method) and Task 3.2 (*i.e.,* the failing tests covering a given method) are performed with a higher precision, recall, and f-score on average.

As a result, the visualization helped developers identify sets of methods failing together, and also provided the traceability so developers could determine which tests test cases were covering them. Morpheus is capable of providing a more global understanding that can help developers get insights into the working of their system.

7.4 IDE versus Visualization

As the results show, the developers were more satisfied with the visualization in comparison to their own development environment. The spread of grades with IDE is considerably more broad than with the visualization. When looking at the results, the participants that were able to find a partial answer to the questions were more likely to give their development environment a passing grade, while developers that were unable to come up with a solution typically gave a low score. Therefore, it makes us think that developers that gave a higher score to their environment were maybe overconfident in the correctness of their answer and

as a result in the ability of their IDE to do its job (given the actual accuracy scores for their IDE answers). In the end, MORPHEUS had a higher satisfaction score and at the same performed better than the participants chosen environment.

Chapter 8

Related Works

In this chapter, I review the prior research in areas related to the work presented in this thesis and discuss the similarities and differences of such prior work with Morpheus. I partition the related works into the following categories: dynamic behavior comprehension, test & production code relationships, and matrix-based visualizations.

8.1 Dynamic Behavior Comprehension

Yu *et al.* [42] studies the different factors that impact a developer's ability to understand their test suite. They noted that "prior knowledge of the software project reduces the time that developers spend reading test code," meaning that a mental model of the tested system helps improve comprehending the test suite. Morpheus builds upon this idea by enabling developers to trace the relationship between test and production code in both ways.

Cornelissen *et al.* [6] created ExtraVis with the goal to ease the comprehension of large execution trace files. The tool consists of two views: one focuses on the order that the events happen, and one focuses on the connections between different components in the system.

Feng *et al.* [12] took a different approach to find phases in the execution and visualized the behavior at a higher level of granularity. The main purpose of both tools is to help understand a single execution trace. In comparison, MORPHEUS focuses on creating a mapping between production and test code, while also giving an overview of the composition of the test suite.

Tests can be used as means to understand the production code. Prior work has proposed tools that enables developers to extract use-case diagrams based on the dynamic behavior of a single test as API usage examples [5, 26]. This allows developers to obtain a mapping from a single test case to what it is testing, however, it is unable to provide us a single high-level overview of the test suite and a direct mapping from a production method to which tests cases are being tested by it.

Another approach to understand the dynamic behavior of a system is focusing on source-code dependencies. As such, Kuhn *et al.* [24] visualized a software system on a file level based on their "vocabulary". Deng *et al.* [7] created CONSTELLATION visualization, a statement-level visualization that encodes static dependencies between program instructions. Palepu *et al.* [28, 29] take a similar approach for their CEREBRO visualization, however, it focuses on the dynamic source-code instructions and clusters them together based on the dynamically obtained information. Bietrich *et al.* [8] focuses on visualizing the modular structure of programs with their tool BARRIO. Chen *et al.* [4] created a visualization assisting in creating traceability between source code and documentation. In comparison, MORPHEUS focuses on the dynamic dependencies between tests and what they cover, while also allowing developers to interactively explore the visualization.

8.2 Test & Production Code Relationship

The co-evolution of tests and code has received prior study and investigation, often aided with visualizations. Works by Zaidman *et al.* [43] and Ens *et al.* [11] show that there is often a synchronous co-evolution of tests and code. In studying co-changes made across code and tests, these works are focused on questions regarding the software processes employed in real-world software projects. This work also studies the relation between tests and code. However, instead of addressing process questions, the visualization-driven approach in this work looks to answer questions concerning the composition and working of test cases at a given moment in the lifetime of a software project. Indeed, the MORPHEUS VISUALIZATION is designed to be configurable to study co-evolution of tests and code; however, that is only one of several other questions that MORPHEUS is seeking to answer.

Per-test-case code coverage allows developers to inspect the bidirectional traceability between test and production code. Indeed, prior works have explored using per-test-case coverage to aid developers in more efficiently performing tasks such as fault localization [23]. Similarly, others focused on helping developers localize what has been tested and by what [36, 23, 27, 15, 37, 38, 2, 31].

MORPHEUS visualizes per-test-case code coverage data to reveal traceability between test- and production-code as well. However, unlike prior approaches MORPHEUS offers engineers a global context of test cases and production code that house individual code-to-test relationships, say between a test and a method. When situating such individual test-to-method relations in larger contexts, developers are able to ask and answer broader questions, *e.g.,* "what are some other tests that are failing when executing a given method?"

Van Rompaey and Demeyer [39] propose different approaches, other than runtime analysis, to establish traceability links between production and test code, *e.g.,* test naming and design conventions, static call graphs, lexical analysis and version log mining. Their work

also specifically focused on creating a one-to-one relation between a unit test and a single unit of code. Unlike such approaches that indirectly infer relationships between tests and code, MORPHEUS recognizes traceability using concretely observed runtime execution data. Further, MORPHEUS also looks to identify individual test-to-code relations. However, it does so in the context of other links that a test might have with different areas of code.

A closely related area of work that employs a matrix-based visualization to highlight test-based, code-coverage data is the TARANTULA visualization by Jones and colleagues [19, 20]. TARANTULA combines code coverage with test-result information to visually present suspiciousness scores for each line of code in a software project using a matrix-based visualization. TARANTULA aims to aid developers in diagnosing faulty areas in their code with a suspiciousness-laden code visualization. In contrast to fault localization, MORPHEUS aids in software comprehension; specifically it aims to help developers comprehend their test suites and product code, together. Indeed, developers may still opt to color cells within MORPHEUS's test matrix with suspiciousness scores. However, developers also have the flexibility to choose other metrics that better highlight the link between code and its test cases, *e.g.,* the frequency with which tests execute a given method, runtime memory consumption, power consumption, or time spent in a method during a test case execution.

8.3 Matrix-Based Visualizations

Prior works in information-visualization research have employed matrix-based visualizations for a wide variety of applications, in areas beyond software engineering. Fernandez *et al.* [13] created CLUSTERGRAMMER., a tool to visualize high-dimensional biological data as a matrix visualization. Similarly, matrices have been used to visualize social networks [16, 41, 17]. While some such works make use of node-link style graph visualizations as well [17, 41], matrices still make up as the main components in such works, owing to their ease of readability.

To visualize high density data sets, prior work has shown the advantage of matrix-based visualizations [10, 1] to explore graph data at many levels. Ghoniem *et al.* [14] shows that the readability of matrix-based visualization outperforms node-link diagrams when graphs become bigger than twenty vertices. Rufiange and Melançon proposed AniMatrix. to visualize the evolution of software [35]. Using matrix visualizations over node-link diagrams can be beneficial especially for high-density matrices for readability. For Morpheus, I employ the matrix approach due to the density and the improved readability it can provide, as well as the intuitive interactability (through means explored in Chapter 3, such as sorting, filtering, and coloring), and I specifically employ this in the context of software testing and test comprehension.

Chapter 9

Conclusions

Bidirectional traceability between test cases and the methods that they execute is currently not easy for developers to gain insight to. Therefore, even simple questions like "by which tests is this method covered" are difficult and time consuming to answer. As a result, developers are facing three challenges: (1) developers may not understand their project's tests, (2) tests are not assets when they are poorly understood, and (3) current tooling is too local for adequate comprehension.

My solution tackles these challenges by presenting per-test-case coverage data into a test-matrix visualization named Morpheus. The initial matrix will give a global overview of all the test cases and methods within the system, presenting which test cases cover which methods. Additionally, a developer can filter and sort the tests and methods according to their own interests. Therefore, a developer is able to get insight into what a test is covering, thus increasing their understanding about the project's tests.

I evaluated Morpheus by performing both three case studies, each on a different real-world system, and a user study. Each case study focuses on a different usage scenario, (1) comprehending test-suite composition, (2) locating a set of methods that are tested by failing

test cases, and (3) examining similarities and differences among multiple software systems and their test suites. Furthermore, the user study demonstrates that when given MORPHEUS participants were better equipped to answer questions regarding traceability between tests and methods, in terms of (1) accuracy, (2) time, and (3) satisfaction with the tool set.

To summarize, my contributions are as following:

1. A novel application of the matrix-styled visual representation towards presenting high information-density insights into test suites within a software project;

2. A series of interaction capabilities, atop the test-suite visualization, that enables seamless transition between global and local views of a software test suite to help answer questions about the form and function of software tests;

3. MORPHEUS: An open-source, publicly deployed and accessible web-application that implements my proposed visualization and interaction capabilities, with the ability to visualize test suites for large, real-world projects; and

4. An evaluation of the proposed visualization and interaction approaches when answering engineering-focused questions about test suites in real-world software projects.

Chapter 10

Future Work

In its current state, MORPHEUS enables developers to sort, filter, and interact with the visualization to explore a test suite. However, there are a four main directions the visualization could be extended or improved to enable deeper exploration, those are:

- Configurable coloring system, allowing developers to visualize other properties next to package name, and test result;
- Configurable dimensions for the axes, *e.g.,* tests and commits to see historical test results;
- Sorting based on test or method similarity, *i.e.,* based on what a test covers, or by which tests a method is covered;
- Allow visualization based on different levels of granularity, *e.g.,* line, methods, classes, and packages

Color is currently used in two ways: (1) to show which methods belong to the same package along the axis, and (2) to show which tests cases pass or fail. However, coloring the nodes,

both along the axis or on the intersection, in different ways could enable the user to gain a deeper insight into their test suite. For example, Morpheus can use suspiciousness scores to sort methods, however, the method's suspiciousness could also be used to color nodes to move the developer's attention to the most suspicious methods. Coloring can help guide the developer's attention to different properties of the tests or methods.

In its current state, Morpheus focuses on a single snapshot of the test suite, meaning it assists in understanding the test suite by visualizing: (1) what it covers, and (2) the result of each test in a single moment in time. However, using the data in the repository, there is a vast amount of knowledge available regarding the test suite's previous states. By enabling developers to obtain historical test coverage of methods, it becomes possible to explore: (1) how the coverage changed over time, and (2) the history of test results. One way of visualizing this could be to replace the methods in the matrix with commits, and present all the tests on the other axis. If one wants to get an overview of all tests that cover a specific method, an interaction could be developed that filters the tests to only the tests that cover a specific method. Developers could use this to gain insights in historical test results, *e.g.,* can this test exhibit "flaky" behavior (*i.e.,* non-deterministic behavior) based on historical test results?

Sorting is an integral part to uncover patterns within the test suite. The tools used so far have been sorting by name, and coverage for both the methods and tests, while for methods another sorting option based on suspiciousness was added. Future works, could focus on automatically cluster methods or tests based on similarity. Hence, such future work could provide insights into tests covering similar behaviors, or methods that are often tested by the same tests.

Morpheus, as presented in this thesis, focuses on presenting data at a method-level granularity, however, there is no reason this could not be extended to allow different levels of granularity, *e.g.,* line, class, or package. Line-level granularity will, for even small projects,

run into scalability issues. An option could be to automatically determine, based on available screen space, what the best granularity is. If there is not enough space on the screen, MORPHEUS could go into a higher level granularity. When filtering one could automatically or manually allow to see more details. For example, if there are a 1000 methods spread over 100 classes and one would only have 500 pixels on the screen available for the visualization, MORPHEUS should adapt its view to present a higher level granularity and allow developers to dive in deeper to parts of interest.

Finally, although the studies thus far showed quite clear benefits and improvement when using MORPHEUS over traditional tools, further empirical study is warranted. More participants can be recruited with more varied tasks, perhaps incorporating some of the additions presented above.

Bibliography

[1] James Abello and Frank Van Ham. Matrix zoom: A visual interface to semi-external graphs. In *IEEE symposium on information visualization*, pages 183–190. IEEE, 2004.

[2] Nadera Aljawabrah and Abdallah Qusef. Tctracvis: test-to-code traceability links visualization tool. In *Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems*, pages 1–4, 2019.

[3] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[4] Xiaofan Chen, John Hosking, and John Grundy. Visualizing traceability links between source code and documentation. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126. IEEE, 2012.

[5] 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*, pages 213–222, Los Alamitos, CA, USA, mar 2007. IEEE Computer Society.

[6] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 49–58, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Fang Deng, Nicholas DiGiuseppe, and James A. Jones. Constellation visualization: Augmenting program dependence with dynamic information. In *Proceedings of International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–8, 2011.

[8] Jens Dietrich, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow. Cluster analysis of Java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*, SoftVis '08, pages 91–94, New York, NY, USA, 2008. ACM.

[9] Kaj Dreef. Morpheus: Software test comprehension and analysis. Github, https://github.com/spideruci/morpheus, 2020. commit-id:831a8d2dffd30b83ca4321212a3cff55b419e37c.

[10] Niklas Elmqvist, Thanh-Nghi Do, Howard Goodell, Nathalie Henry, and Jean-Daniel Fekete. Zame: Interactive large-scale graph visualization. In *2008 IEEE Pacific Visualization Symposium*, pages 215–222. IEEE, 2008.

[11] Barrett Ens, Daniel Rea, Roiy Shpaner, Hadi Hemmati, James E Young, and Pourang Irani. Chronotwigger: A visual analytics tool for understanding source and test co-evolution. In *2014 Second IEEE Working Conference on Software Visualization*, pages 117–126. IEEE, 2014.

[12] Yang Feng, Kaj Dreef, James A. Jones, and Arie van Deursen. Hierarchical abstraction of execution traces for program comprehension. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 86–96, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Nicolas F Fernandez, Gregory W Gundersen, Adeeb Rahman, Mark L Grimes, Klarisa Rikova, Peter Hornbeck, and Avi Ma'ayan. Clustergrammer, a web-based heatmap visualization and analysis tool for high-dimensional biological data. *Scientific data*, 4:170151, 2017.

[14] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[15] Maen Hammad, Ahmed Fawzi Otoom, Mustafa Hammad, Nadera Al-Jawabreh, and Rawan Abu Seini. Multiview visualization of software testing results. *International Journal of Computing and Digital Systems*, 9(1), 2020.

[16] Nathalie Henry and Jean-Daniel Fekete. Matlink: Enhanced matrix visualization for analyzing social networks. In *IFIP Conference on Human-Computer Interaction*, pages 288–302. Springer, 2007.

[17] Nathalie Henry, Jean-Daniel Fekete, and Michael J McGuffin. Nodetrix: a hybrid visualization of social networks. *IEEE transactions on visualization and computer graphics*, 13(6):1302–1309, 2007.

[18] William E Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, (4):293–298, 1978.

[19] 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*, pages 273–282, November 2005.

[20] James A. Jones, M. J. Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, 2002.

[21] James A. Jones, Alessandro Orso, and Mary Jean Harrold. Gammatella: Visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, 2004.

[22] Junghun Kim, Vijay Krishna Palepu, Kaj Dreef, and James A. Jones. Tacoco: Integrated software analysis framework. Github, https://github.com/spideruci/tacoco, 2015.

[23] Negar Koochakzadeh and Vahid 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*, TAIC PART'10, page 129–136, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. Software cartography: thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):191–210, 2010.

[25] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A task oriented view of software visualization. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '02, pages 32–, Washington, DC, USA, 2002. IEEE Computer Society.

[26] Seyed Mehdi Nasehi and Frank Maurer. Unit tests as api usage examples. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[27] Ahmed Fawzi Otoom, Maen Hammad, Nadera Al-Jawabreh, and Rawan Abu Seini. Visualizing testing results for software projects. In *Proc. of the 17th International Arab Conference on Information Technology (ACIT'16), Morocco*, 2016.

[28] Vijay Krishna Palepu and James A. Jones. Visualizing constituent behaviors within executions. In *1st IEEE Working Conference on Software Visualization, New Ideas and Emerging Results Track (VISSOFT-NIER)*, pages 1–4, September 2013.

[29] Vijay Krishna Palepu and James A. Jones. Revealing runtime features and constituent behaviors within software. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 86–95. IEEE, 2015.

[30] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 654–664. IEEE, 2017.

[31] Adriana Rodrigues, Maria Lencastre, and A de A Gilberto Filho. Multi-visiotrace: traceability visualization tool. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 61–66. IEEE, 2016.

[32] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529 –551, aug 1996.

[33] G. Rothermel and M.J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering, IEEE Transactions on*, 24(6):401 –419, jun 1998.

[34] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 34–43. IEEE, 1998.

[35] Sébastien Rufiange and Guy Melançon. Animatrix: A matrix-based visualization of software evolution. In *2014 Second IEEE Working Conference on Software Visualization*, pages 137–146. IEEE, 2014.

[36] Amjed Tahir and Stephen 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*, pages 21–30. IEEE, 2015.

[37] Thomas Tamisier, Peter Karski, and Fernand Feltz. Visualization of unit and selective regression software tests. In *International Conference on Cooperative Design, Visualization and Engineering*, pages 227–230. Springer, 2013.

[38] Bart Van Rompaey and Serge Demeyer. Exploring the composition of unit test suites. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, pages 11–20. IEEE, 2008.

[39] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 209–218. IEEE, 2009.

[40] Elaine J Weyuker and WEYUKER EJ. The oracle assumption of program testing. 1980.

[41] Ji Soo Yi, Niklas Elmqvist, and Seungyoon Lee. Timematrix: Analyzing temporal social networks using interactive matrix-based visualizations. *Intl. Journal of Human–Computer Interaction*, 26(11-12):1031–1051, 2010.

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

[43] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.