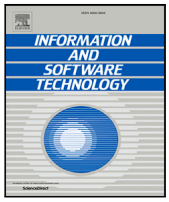
Information and Software Technology 155 (2023) 107085

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Exploring granular test coverage and its evolution with matrix visualizations

Kaj Dreef [a], Vijay Krishna Palepu [b,1], James A. Jones [a,*]

[a] University of California, Irvine, United States of America
[b] Microsoft, United States of America

ARTICLE INFO

Keywords:
Software testing
Visualization
Software comprehension
Software test comprehension

ABSTRACT

Context: Current software-development tools that are used in practice make understanding the test execution of software difficult, for both granular tasks (*e.g.,* answering questions such as, "which test cases execute this method?") and global tasks (*e.g.,* answering questions such as, "what is the proportion of unit tests to system tests?"). Current tools typically support local, file-based views of a project's test suite and its execution data, and rarely offer a global overview. Even more rarely, do they provide access to historical information of this nature. Such global overviews can provide a larger context for a method's execution by test cases; help identify other similar, or related methods; and even reveal similarity between individual tests.

Objective: This work approaches such challenges with a novel, interactive, matrix-based visual interface that provides a global overview of a software project's test suite, specifically in the context of the methods available in the project's codebase. Through a series of interactive functions to sort, filter, query, and explore a test-matrix visualization, we evaluate how developers can effectively answer questions about their project's test suite, and the code executed by such tests.

Method: We built a dynamic test-suite analysis and software-visualization tool that implements our designed interface to address the challenges of understanding the testing of software systems. With this implementation, we conducted a user study of 20 software developers to assess their ability to understand and report test execution information and measured accuracy and time. Additionally, we present a series of case studies to demonstrate a number of insights that our tool reveals.

Results: Our evaluations, performed on 26 real-world software systems, show that the interactive visualization assisted developers to answer questions about software tests and the code they execute. Further, the visualization consistently outperforms traditional development tools, both in accuracy and time taken to complete software-engineering tasks.

Conclusion: Global-overview test matrices offer novel perspectives on test-suite composition, which can guide software development and testing practices.

1. Introduction

Software developers routinely need to answer questions about their current testing practices in order to address current or future testing needs. Unfortunately, current software-development tools provide limited support to help answer such questions. In this work, we address the challenges of current tools with a novel, interactive, matrix-based visual interface that provides both (1) global overviews of a software project's test suite and (2) interactive functionality that allows developers to query granular details.

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

Questions about the "form" of a test suite often require a global, or overarching, understanding of the entire suite. These questions help identify the current/existing or future/potential organization of a test suite for presentation to a developer. The tests may be organized in a number of ways. For example, tests can be organized by results: *e.g.,* passing and failing. Or, tests can be organized into clusters, where each

* Corresponding author.
E-mail addresses: kdreef@uci.edu (K. Dreef), Vijay.Palepu@microsoft.com (V.K. Palepu), jajones@uci.edu (J.A. Jones).

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

https://doi.org/10.1016/j.infsof.2022.107085
Received 18 March 2022; Received in revised form 24 September 2022; Accepted 29 September 2022
Available online 20 October 2022

cluster represents a product behavior or functionality to be verified. And, perhaps a more common organization found in real-world test suites is to differentiate unit, integration, and system tests. In such cases, tests are organized into components that mirror the packages and methods that are directly verified by their respective tests. Organizing tests by different criteria may reveal different aspects of the suite as a whole, and help engineers navigate and understand their test suites. For instance, when organizing tests as unit/integration/system tests, an engineer can easily identify the test cases that test a method that the engineer is trying to refactor or modify. Similarly, when an engineering manager is trying to assess a product's testing effort, it can be useful to organize the tests by the tests' coverage (*e.g.,* increasing to decreasing), or cluster them by the product behaviors they are trying to verify.

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

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

Additionally, time is dimension that may be difficult for developers to know and comprehend, *e.g.,* how the tests evolved over the course of the project, and how the execution of those test cases changed over that time. Developers who understand test evolution may be better capable of identifying unusual new execution behaviors, or can better answer past design rationale decisions. Having both global overviews of the evolution of the form of test suites, as well as local views of the function of individual tests, may be useful to answering questions about the evolution of those tests, as well as to reveal interesting insights and patterns.

In this work, we visualize software test-execution data in an interactive matrix visualization that we call Morpheus (see Fig. 1) to support developer understanding, querying, discovery, and exploration of their test suites. Morpheus presents engineers with global overviews of their test suites in a visualization that captures granular data about a test suite's execution, to reveal patterns in the test suites and their executions. To seamlessly explore localized views of specific tests within global overviews, Morpheus enables user-driven interactions. Such interactions allow engineers to quickly filter test data in the visualization to a subset of tests and methods within a software project. Engineers would also be able to re-organize the data (tests and methods) to reveal patterns, both in local and global views, and to view historical test information.

The main contributions of this work are:

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

2. Motivating examples

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

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

As an informal feasibility study, we asked three developers with 9, 12, and 24 years of Java-development experience this question. All three developers agreed that this was a useful question and expressed surprise that such a question is so simple and that it was not immediately obvious how to answer it. Here are the ideas that they came up with, often after hours of reflection and brainstorming:

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

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

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

Regarding the first of these questions of "same or similar" methods, Begel and Zimmerman [3] conducted a survey that elicited 607 responses from software engineers and found that 81% of testers said

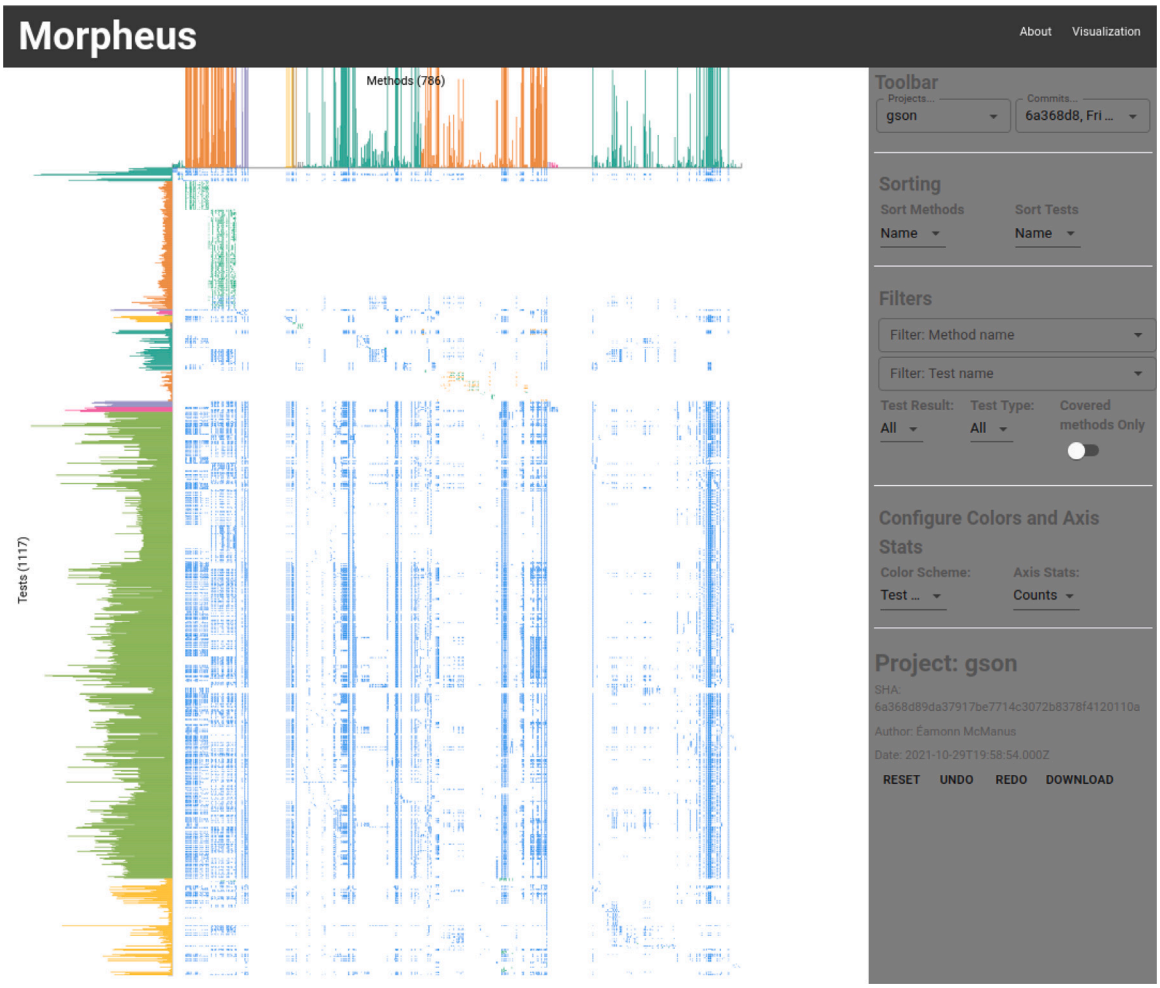


Fig. 1. Morpheus Web Application visualizing GSON's test results (commit 6a368 2021-10-29).

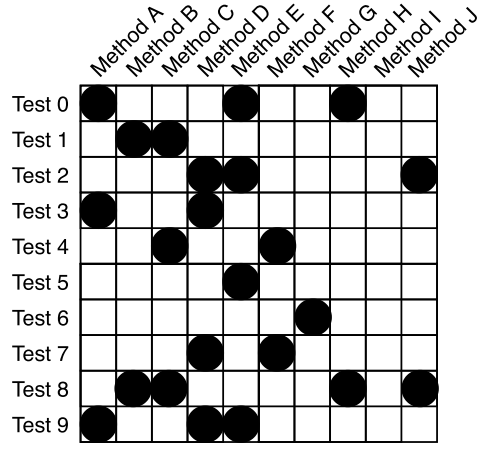
that answering questions such as "How should we handle test redundancy and/or duplicate tests?" was worthwhile. For the second of these questions, regarding the composition of our test suite in terms of unit, integration, and system tests, Begel and Zimmerman also found that their respondents asked "What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

To ground the research, we conducted a survey of software developers to help determine developer needs for comprehending software test suites. The survey was posted to a local chat channel and received 10 responses, from professional software engineers (6), graduate students (3), and a researcher (1), with a mean 9 years of software-development experience. A series of questions were posed to determine the perceived importance of a number of tasks related to understanding test execution, on a 7-point Likert scale. Several tasks that require a granular understanding of test execution (*i.e.,* "function") received the highest scores of importance, such as "Know the specific code executed by a failing test" (6.6), "Know if your newly written function is executed by a failing test case" (6.5), "Know the list of specific tests that execute code that looks suspiciously buggy/faulty" (6.1), and "Know the list of functions that are also executed when a buggy function is executed" (5.7). Questions that required understanding patterns of test execution (*i.e.,* "form") within a test suite (*e.g.,* identifying similar tests, identifying unit or system level tests) were rated as important but on a lower tier: 5 to 5.9. Participants also rated questions that require a historical analysis of test executions as important — 5.3 to 5.8 in the survey ratings. However, responses related to historical analyses also saw a larger standard deviation — some viewing the evolution of test execution as "Essential/Strongly Important" (7), whereas one participant viewed it as "Slightly Unimportant" (3). These results, as well as the prior studies by Torkar and Mankefors [6], Rafi et al. [2], and Begel and Zimmerman [3], demonstrate the promise of greater tool support for test-execution comprehension.

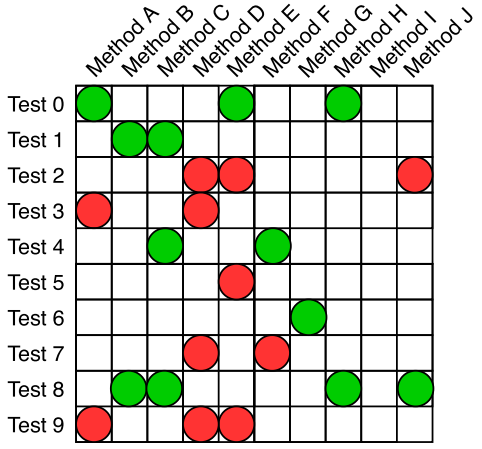
Existing developer tools cannot easily help to answer such questions. And, some research techniques (*e.g.,* dynamic slicing and change impact analysis) might only help to address a subset of them, and are not currently implemented in a way that developers can easily utilize them in practice. For each such question, we could imagine developing a specialized analysis technique to solve just that one problem (*e.g.,* using clustering on per-test-case execution data to identify groups of similar executions), but giving developers a way to view, query, discover, and explore their test execution behavior would allow for developers to form their own questions and answer them. Our goal with this work is to help to address these problems and to answer these questions, and more, with a query-able and interactive global-overview visualization of test-case execution.

3. Global overviews of granular test coverage with interactive matrix visualizations

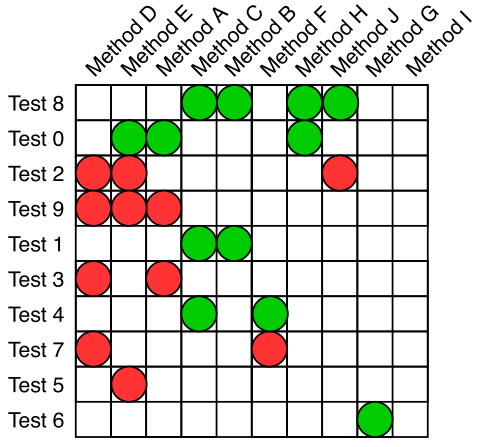
One way to simultaneously comprehend test code and product code is to trace and reveal relations between them; like done in a test-coverage matrix (or simply, "test matrix"). A test matrix places test



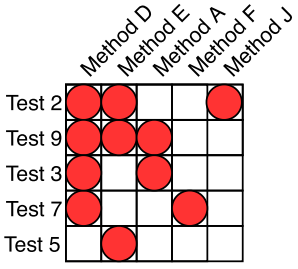
(a) Test matrix presenting tests and methods as rows and columns; the intersection shows a dot if it was covered.



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

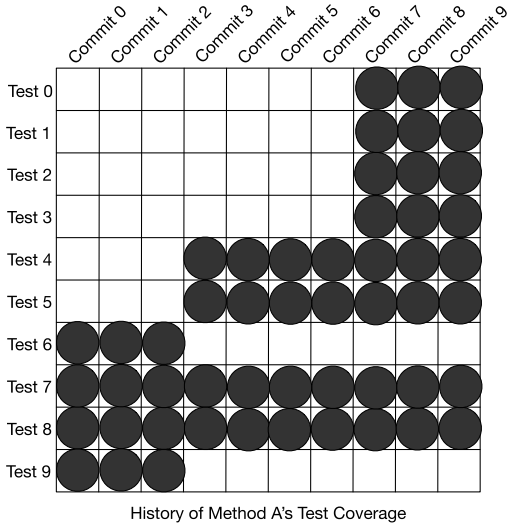


(c) Sorting tests and methods by coverage.

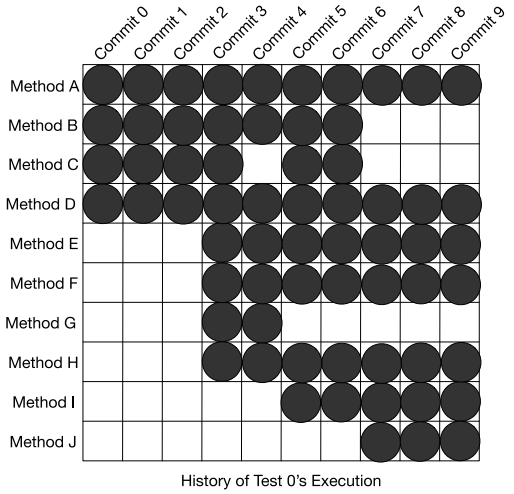


(d) Filtering to only failing tests and the methods executed by them.

Fig. 2. Test matrices organized and presented in various ways. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) Method A's Execution History: tests as rows, commits as columns; a colored dot shows that Method A was executed by a given test, in a given commit.



(b) Test 0's Execution History: methods as rows, commits as columns; a colored dot shows that Test 0 was executed a given method, in a given commit.

Fig. 3. Repurposing Test matrices to depict execution history of a specific test or method. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

cases on one dimension/axis and the program entities to be covered or executed on the other dimension/axis. Consider the simple test matrix shown in Fig. 2(a) that depicts the test cases on the vertical axis and the program methods to be executed on the horizontal axis, *i.e.,* each row represents a single test case, and each column represents a single method. The cells at the intersection of the rows and columns represent whether the test case (row) executes (*i.e.,* covers) the method (column). For example, the first row in Fig. 2(a) shows the coverage for Test 0. Test 0 covers Methods A, E, and H. Similarly, Method A was executed by Tests 0, 3, and 9.

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

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

3.1. Artifacts along rows and columns

The rows and columns of the test matrix can represent multiple artifacts, such as test cases, methods and code commits.

Test Coverage. For example in Fig. 2(a), rows represent test cases and columns represent program methods. However, this mapping is arbitrary and could equivalently be reversed (*i.e.,* methods on rows and test cases on columns). Moreover, these dimensions can be configured to represent other artifacts. For example, the axes may represent test cases on one axis and other granularities on the other axis: (a) Individual source-code lines; (b) Methods; (c) Source files; and (d) Modules or Packages. Finally, different versions of the artifacts, across different source code commits can also be represented. For instance, one could configure each dimension to show all methods and tests – both past and current – from across all versions of a project.

Method Execution History. Another use of one of the matrix's axes is to represent code commits — to depict evolving executions of artifacts such as tests or methods along that dimension. Consider Fig. 3(a) that shows the evolution of Method A's execution over Commits 0 through 9. In Fig. 3(a), each row represents the different tests that ever executed Method A; and columns represent different commits, *i.e.,* different versions of a software project. Fig. 3(a) shows how Method A was *executed* – not modified – with each passing commit or change. Fig. 3(a) shows how Method A was executed by a subset of four tests (Tests 6–9) during early commits (Commits 0,1,2), then later executed by Tests 4, 5, 7 and 8, starting in Commit 3. In the final few commits, Method A is executed by a larger number of tests — suggesting that either these newer tests were added to test Method A explicitly; or Method A is now transitively invoked by other methods in the project, which in turn are executed by the newer tests.

Test Execution History. Just as with a method's execution history, we can model a test's execution history using a test-coverage matrix. Fig. 3(b) shows the evolution of Test 0's execution from Commit 0 through Commit 9. The rows depict the methods executed by Test 0; and the columns depict different commits of the project. Fig. 3(b) shows that Test 0 executed Methods A through D for Commits 1, 2, and 3. However, in subsequent commits we find that Test 0 executes an increasing number of methods. Such a model of test execution history can easily reveal the growing coverage by a specific test.

3.2. Visually augmenting artifact attributes

Artifact Bars. Each dimension of the matrix represents a series of artifacts, be it methods, tests or commits. Those artifacts themselves have properties of their own, which are often numeric. For instance, every method is executed by a certain number of tests, or executed by a test across a certain number of commits. Similarly, a test executes a certain number of methods. Such numeric attributes (*e.g.,* number of executed methods/test, age, recency of change) can be represented with proportionally-sized bars for each artifact along a matrix's dimension — like bar-charts on either x- or y-axes. An example of such artifact bar charts for methods on the x-axis and tests for y-axis are shown in Fig. 1 for project GSON.

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

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

In our evaluation, we also use color overlays to highlight test pass-or-fail results and test type (*e.g.,* unit, integration, or system tests). The implementation also supports future extensions to represent other metrics, such as: performance data, code ownership, code churn, and distinct feature areas.

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

3.3. Juxtaposing artifacts via sorting

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

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

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

To illustrate how such sorting could work, consider the matrix in Fig. 2(c) that builds further upon Fig. 2(b). Both axes in Fig. 2(c) are sorted by coverage. Methods executed by more tests are sorted left to right, while tests executing more methods are sorted top to bottom. Such sorting could reveal methods that are never or barely tested (*e.g.,* only Test 6 executes Method G, and Method I is untested altogether), or reveal test cases like Test 8 that executes a broad swath of the program (suggesting that it is an integration or system test) and Tests 5 and 6 that execute only a single method (like unit tests).

3.4. Drill-downs via pan, zoom, and artifact filtering

Pan and Zoom. Real world projects have hundreds (and often thousands) of artifacts (tests, methods, and commits). Naturally, a two-dimensional matrix with hundreds of artifacts represented on the rows and columns can be overwhelming for a developer to view and navigate in its entirety. To alleviate such information overload, Morpheus affords the ability to pan and zoom the test matrix. The pan and zoom functionalities offers users a familiar interaction model to zoom-in, and pan around the more granular (or local) cells of the test matrix, or zoom-out and view the matrix's global form in its entirety.

Artifact Filtering. When trying to understand how a test suite is verifying product behavior, developers want to focus on specific tests, instead of the whole test suite in one view. To support such detailed exploration, we provide the ability to filter down to tests, production code (*e.g.,* methods), or commits that are of interest to developers. Unlike with pan and zoom, filtering allows developers to limit the scope of artifacts that they choose to inspect at any given time. We allow filtering for various aspects of the project and test suite, such as, method name, test name, level of testing (*e.g.,* unit, integration, system), commit, and test result (*i.e.,* pass or fail).

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

4. Morpheus visualization

To capture global overviews of granular test matrices of real-world programs, we created the Morpheus Visualization. We implemented Morpheus as a web application, and used it to evaluate the concepts presented in Section 3 — details of that evaluation are available in Section 5. In this section, we walk through how a developer would use Morpheus, and highlight the technical details of the underlying implementation.

4.1. A developer's walk-through

Select a Commit within a Software Project. After opening Morpheus, a developer could choose a specific commit of a software project, whose test-coverage data that she wishes to visualize. She would be able to make such project and commit selections in the left sidebar of Morpheus's interface. For instance, Annotation ① in Fig. 4(a) illustrates how a commit selection within the GSON project prompts Morpheus to visualize the corresponding test data as a highly granular matrix.

Use Toolbar to Sort and Filter. The left toolbar in Morpheus shows a range of options for sorting and filtering by methods and tests, as detailed earlier in Section 3. These options afford the developer to interact with the coverage data by molding the test matrix to reveal potential testing patterns for her project of interest. Fig. 4(b) shows different sorting options for tests and methods. For instance, Annotation ②a shows how the developer can choose to reorder the methods and tests by their associated code coverage. Such sorting would reveal the most executed methods and tests with the highest code coverage at the top-left corner of the matrix, while moving the lesser executed methods and tests to the bottom-right corner. Similarly, Annotation ②b in Fig. 4(b) shows how the developer can filter down to only failing or passing tests. Similarly, Annotation ②c shows how the developer can select the coverage data associated with a specific method, *e.g.,* `typeEquals`.

Apply Color Overlays and Artifact Bars. Annotations ③a and ③b in Fig. 4(c) show how the developer may select from an array of different options to both: (a) annotate the artifact axes with bar plots as discussed in Section 3, and (b) highlight artifact relations between tests and methods using color overlays, *e.g.,* using green and red to show failing and passing test coverage. The ability to augment the matrix using such visual cues add to the interactive affordances of Morpheus.

Explore History of a Specific Test or Method. Finally, the developer would also be able to right-click an artifact along either the x- or *y*-axis in Morpheus, to be able to view history of the said artifact. For instance, Annotation ④ in Fig. 4(d) illustrates how a developer may opt to explore the evolution of how the `Iterator<JsonElement>.iterator()` was executed by GSON's tests over successive commits.

4.2. Implementation

As shown in Fig. 5, our Morpheus implementation comprises three components: (1) a per-test coverage data analyzer as a containerized component, (2) a RESTful API serving access to the coverage data, and (3) a web-based visualization of the coverage data. The authors have open sourced this implementation to facilitate reproducibility and extension by other researchers and practitioners [7]. Although an IDE integration would likely be a preferable choice in practice for developers, we chose this approach to enable the evaluation, regardless of user platform and development-environment preferences.

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

RESTful Access to Coverage Data. The coverage data collected from the per-test analysis is stored in a database, and is exposed via a RESTful API as shown in Fig. 5. We collect code coverage at line-level granularity, thus creating granular traceability between tests and product code. For Morpheus, we focus on a method-level granularity, and translate line-level coverage to method-level coverage by tracking the line-mappings (beginning- and ending-source line numbers) for methods within a project.

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

This implementation affords users with the interaction capabilities of sorting, filtering, and color-based encoding of the coverage data, as detailed in Section 3, which provides the basis upon which we conduct our evaluations of our test-matrix visualization to support answering developer questions.

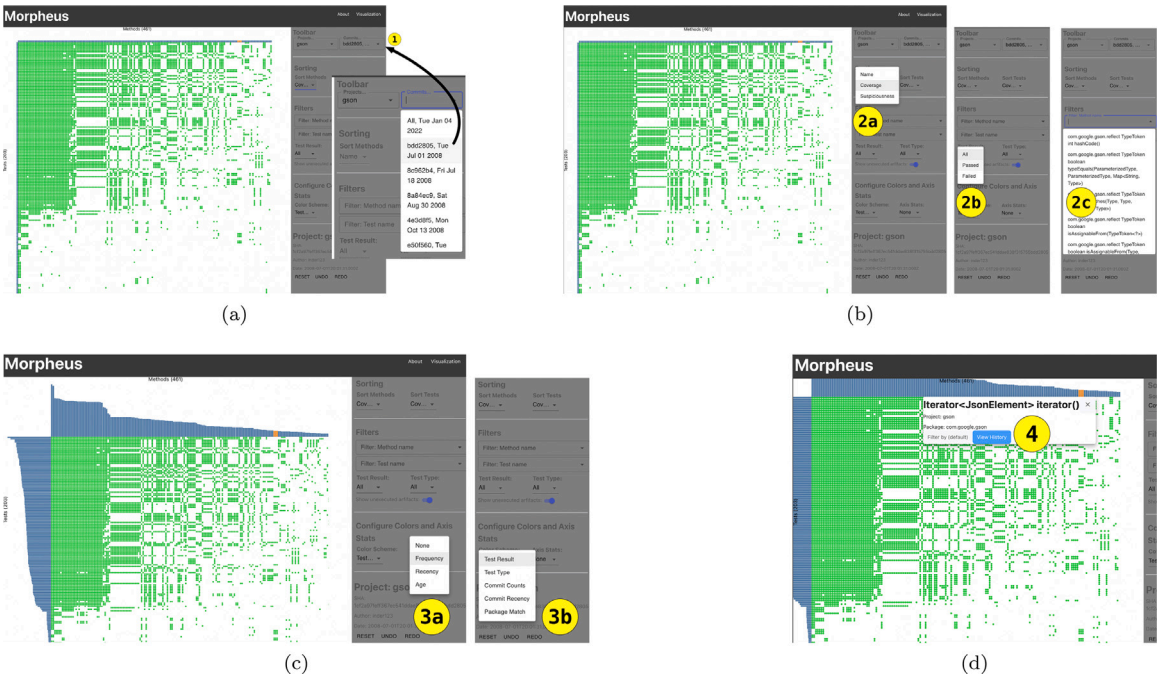


Fig. 4. A Developer's Walkthrough of Morpheus Visualization. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

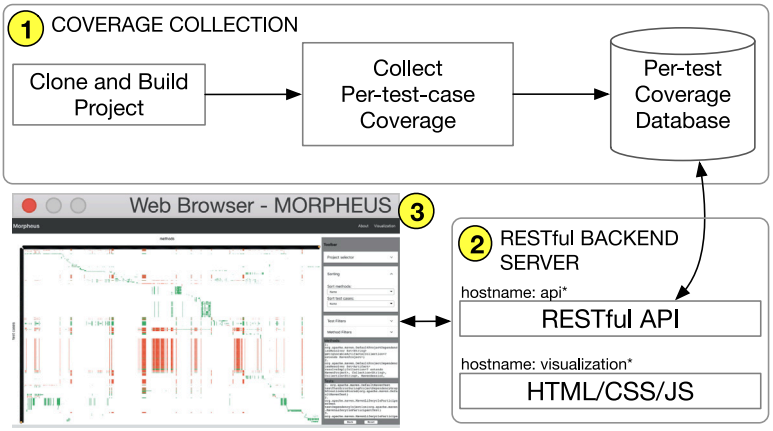


Fig. 5. Morpheus Implementation.

5. Evaluation

Using the implementation of Morpheus that we presented in Section 4, we evaluated its effectiveness in aiding developers when answering detailed questions about a project's a test suite. We specifically ask the following research questions:

RQ1 Does Morpheus help to reveal the function of individual test cases within a test suite?

RQ2 Does Morpheus help to reveal the global form of a test suite?

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

RQ4 Can Morpheus help reveal patterns in the evolution of a project's test suite over multiple code commits?

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

	User study			Case study		
	Task 1	Task 2	Task 3	Single-Project	Cross-Project	Project-History
RQ1	X	X	X			
RQ2			X	X	X	
RQ3					X	
RQ4						X

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

During this evaluation, we applied Morpheus on a set of 26 open-source Java projects. We chose a variety of popular projects in GitHub,

Table 1
List of mined software projects.

	Project	# of Commits Mined	Oldest Commit's Date	Newest Commit's Date	# of Test in Newest Commit	# of Methods in Newest Commit	LoC in Newest Commit	Files in Newest Commit
1	fastjson	82	1/11/13	5/17/20	4815	1736	175666	2888
2	commons-cli	3	3/9/17	10/23/21	382	254	6258	52
3	commons-configuration	18	7/24/16	3/9/20	2804	2718	68964	459
4	commons-csv	20	7/15/14	7/24/21	396	221	8788	42
5	commons-imaging	3	10/28/18	4/27/19	535	1743	38751	510
6	commons-io	20	4/10/12	7/9/21	1806	1484	40674	364
7	commons-jexl	6	12/18/15	6/17/21	792	2145	31753	177
8	commons-lang	14	12/22/13	2/26/21	6710	3291	88433	447
9	commons-net	11	4/15/16	2/13/21	280	1532	29278	274
10	commons-pool	33	10/30/13	8/14/21	303	761	15236	97
11	commons-text	16	1/25/17	7/21/20	1170	926	25535	187
12	commons-validator	10	11/18/15	8/3/20	509	627	16781	150
13	commons-vfs	4	12/24/19	3/6/21	244	2571	35761	525
14	dubbo	33	9/11/17	9/18/21	1042	12687	182681	2612
15	iotdb	7	5/7/20	9/7/21	1277	19948	227115	1781
16	maven	16	2/23/17	11/14/21	670	6665	84224	978
17	xmlgraphics-commons	1	1/21/21	1/21/21	187	2283	36010	400
18	zookeeper	1	3/17/21	3/17/21	1853	5742	113842	883
19	error-prone	11	5/28/20	11/4/21	4712	6858	202647	2103
20	google-java-format	8	4/12/16	11/19/21	1189	815	16936	76
21	gson	14	1/31/10	9/30/21	964	1345	25204	134
22	re2j	2	2/18/15	11/17/15	1524	423	13120	39
23	truth	22	8/7/12	5/25/21	1310	1958	34461	185
24	itext7	31	5/2/16	10/20/21	5456	11664	273568	2140
25	jsoup	43	1/31/10	9/30/21	964	1345	25204	134
26	jpacman-framework	75	4/23/16	1/4/19	45	191	5463	55

which were implemented in Java and built using Maven, which are requirements of our build-orchestration tool, Tacoco [8]. These projects, along with several metrics of those projects, are listed in Table 1. Each of the columns in Table 1 provides statistics about the Java artifacts specifically (*i.e.,* number of commits that we mined, oldest commit date, newest commit date, number of test cases in the latest commit, number of methods in latest commit, lines of code in latest commit, and number of files in the latest commit).[2] Further, for the purposes of the evaluation, we define "unit tests" as tests that execute methods within a single class; "integration tests" are tests that execute methods across multiple classes in a single package; and "system tests" are tests that execute multiple methods that reside in at least two different packages.

5.1. User study

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

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

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

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

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

Task 1: Locate all tests that cover a specific method. Developers often re-run tests after modifying code to check if the change caused any regression. However, current tools and IDEs offer limited support to trace how individual tests execute specific methods. Many IDEs have code-coverage tools, but it provides a file-centric view of the production code — showing which lines are covered, but not by which tests. So, we presented participants with this first task:

[2] Morpheus visualizations of all projects are provided in an online appendix, which is available at http://spideruci.org/morpheus-ist-appendix.pdf.

Task 1 List the set of tests that cover the following method:

Group A:	Group B: MissingOptionException.getMissingOptions()
HelpFormatter.getOptPrefix()	

Task 2: Find all methods that co-fail with a specific method. Methods that fail together, *i.e.,* they are executed by the same failing test cases, may offer clues about a failure's root cause. To mimic such a scenario, we asked participants to complete this second task:

Task 2.1 List the set of methods that are also executed by the one or more of the same failing test cases that execute:

Group A:	Group B:
Option.setArgName(String)	Option.setValueSeparator(char)

Task 2.2 List the set of failing tests that are testing method:

Group A:	Group B:
Option.setArgName(String)	Option.setValueSeparator(char)

Task 3: Distinguish different types of tests that execute a specific method. Developers often write multiple types of tests: unit, integration, and system tests. Understanding how the system is executed by various kinds of tests can give insight into a specific method's test plan and testability. As such, the participants' third task focused on understanding the composition of the test suite:

Task 3.1 List the set of unit tests for method:

Group A: HelpFormatter.findWrapPos(String,int,int)	Group B: HelpFormatter.renderWrappedText(StringBuffer,int,int,String)

Task 3.2 List the set of integration tests for method:

Group A: HelpFormatter.findWrapPos(String,int,int)	Group B: HelpFormatter.renderWrappedText(StringBuffer,int,int,String)

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

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

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

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

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

The row for Task 1 in Table 2 suggests perfect mean precision and recall scores of 1.0 for the Visualization rounds. Whereas, for the IDE rounds the mean precision and recall scores stand at 0.68 and 0.18, respectively. Notice, this trend continues for all tasks: the participants consistently provide much more accurate answers using Morpheus, in comparison to when using their own development environment tools.

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

Next, Fig. 6 presents the time taken (in seconds) by each participant to finish each task. These boxplots show the timing results per task and for both rounds. During the IDE round, the participants made use of a variety of tools to get to an answer. The results suggest that the participants were faster in reporting answers with Morpheus.

Table 2
Mean Precision, Recall, and F-score results.

Tasks	N	Precision		Recall		F-Score	
		IDE	Vis.	IDE	Vis.	IDE	Vis.
Task 1	20	0.68	1.00	0.18	1.00	0.23	1.00
Task 2.1	20	0.25	1.00	0.01	0.97	0.02	0.98
Task 2.2	20	0.11	0.90	0.05	0.90	0.05	0.90
Task 3.1	20	0.60	1.00	0.41	1.00	0.39	1.00
Task 3.2	20	0.19	0.94	0.10	0.94	0.10	0.94

Finally, we also asked participants their level of satisfaction with each of the tool treatments, on a scale of 1 (least satisfied) to 10 (most satisfied). The mean satisfaction score for the IDE treatment (*i.e.,* using any tool available) was only 4.4, whereas the mean satisfaction score for Morpheus was 8.8. In informal feedback from the participants following the study sessions, a common sentiment that was expressed was surprise that the tasks were so difficult to answer with their existing tools, and that Morpheus made these tasks truly easy to answer.

5.2. Commons-cli *case study*

To highlight to the reader the ability of Morpheus to reveal the overarching, global behavior of a test suite, we present a case study of a single software project (other projects are visualized in the next Section 5.3). When discussing the global form of the test suite, we specifically expect to be able to answer the following questions: (1) "what is (and is not) tested?"; and (2) "what types of tests are present in the test suite?" Using Commons-CLI's test suite as a case study, we show how Morpheus can aid in answering such questions about that project.

Fig. 7(a) shows how Morpheus presents the entire test suite for Commons-CLI to a developer. The initial view of the test matrix gives us an overview of all the methods (horizontal axis) and tests (vertical axis). The tests and methods are both sorted based on their names, and names of their enclosing packages and classes. Based on this view, we can make two observations. First, the horizontal strings of (green and red) test rows suggest that most tests seem to execute many methods. Second, the similarity between the horizontal (green and red) test rows shows that many tests execute common methods and are perhaps variations of one another.

5.2.1. *"Tested, or not tested"*

To determine what is (or not) tested, we sort the methods and tests by their coverage (*i.e.,* number of methods executed by a test; and number tests that a method is executed by). Fig. 7(b) shows the sorted view of Commons-CLI's test suite, with frequently executed (and tested) methods to the left, and tests executing the most number of methods at the top. By sorting the tests and methods, it becomes apparent that a selection of methods is very well-tested (on the left side), and the further you move to the right the sparser the coverage becomes, to the point of no coverage for a (small) group of methods. Developers can use this view to directly take steps on where the test suite can be improved.

5.2.2. *"Type of tests"*

Finally, Morpheus enables coloring to represent the type of test cases: *e.g.,* passing versus failing, or unit versus integration versus system. Figs. 7(a) and 7(b) color the tests according to their pass/fail status: green denoting passing and red denoting failing. Alternatively, Fig. 8(c) shows the colored result for each type of test: orange dots indicating integration tests and green dots indicating unit tests. From this view, we can clearly observe that Commons-CLI tests are primarily written as integration tests (326 tests), with limited amounts of unit tests (29 tests).

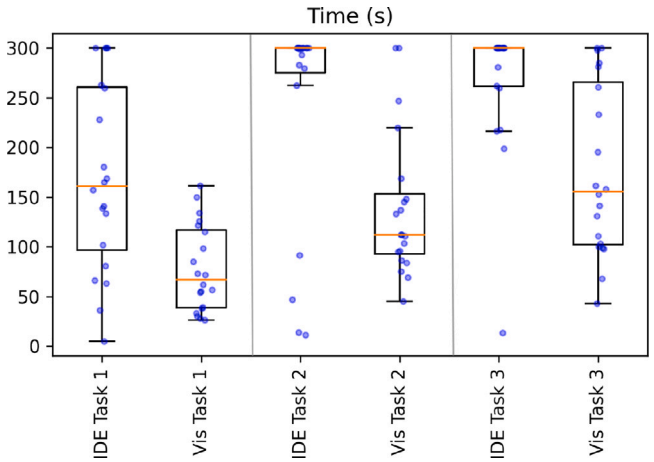
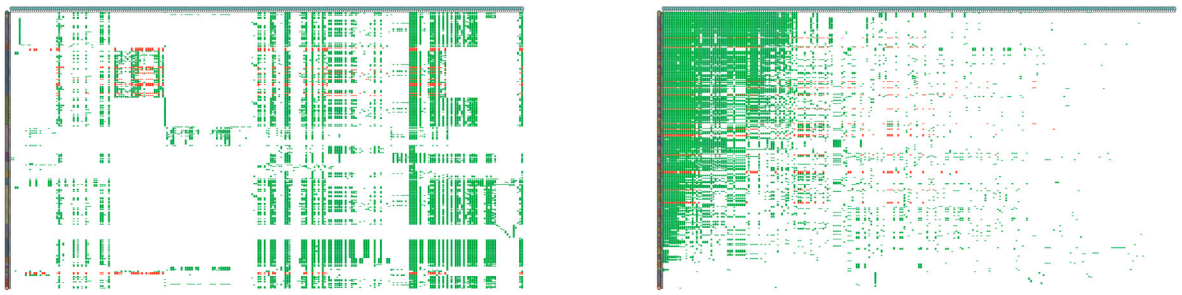


Fig. 6. Time (seconds) taken by participants to complete each round; separated per task. (Lower is better).



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

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

Fig. 7. Sorting Commons-CLI's Methods and Test cases. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.2.3. Case study inference

Morpheus provides multiple ways to better understand the test-suite composition, allowing engineers to sort through tested and untested methods. Combining sorting and coloring allows us to explore what is covered, what is tested together, and what types of tests are in the test suite.

5.3. Cross-project-comparison case study

So far, we focused on a single software project's test suite, and its form and function. We now compare and contrast visualizations from multiple projects, as a case study, to see if lessons can be learned about different test suites from differing software systems.

5.3.1. Tests-to-project fit

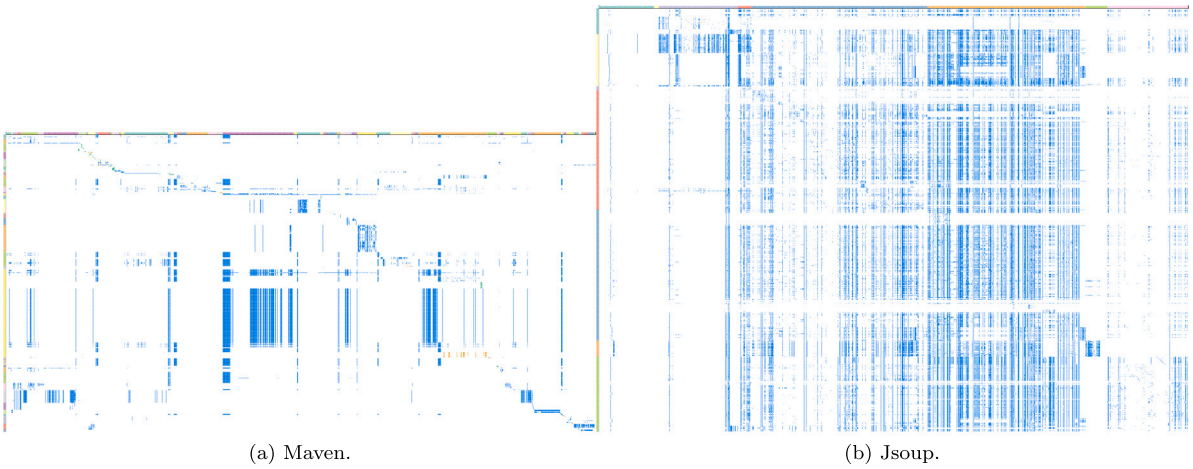
Inter-project comparisons may be useful to allow software engineers to assess if the degree and type of testing is appropriate for their type of program. For example, one may expect that the test suite for an API-based utility library to be largely comprised of unit tests — each method in the library performs some function that can be independently tested with limited dependencies among those methods. Similarly, for an interactive system, one may expect to find test suites that have many more system and integration tests, in addition to unit tests, because the system itself relies upon multiple interacting components to perform its functionality. Fig. 8 shows how Morpheus presents four projects – Maven, Jsoup, Commons-CLI, and Commons-IO – coloring according to test type: blue indicates system tests, orange indicates integration tests, and green indicates unit tests. All additional projects are also visualized in this way in the online appendix[1].

The visualizations in Fig. 8 make two aspects apparent, (1) long vertical lines, and (2) the sparseness of some matrices. The long vertical lines show us many tests cover the same or similar sets of methods. We see this happen mainly with Jsoup (Fig. 8(b)), and Commons-CLI (Fig. 8(c)), which can be attributed to the way tests are structured. Maven also contains some longer vertical lines, but it is not as apparent as Jsoup and Commons-CLI.

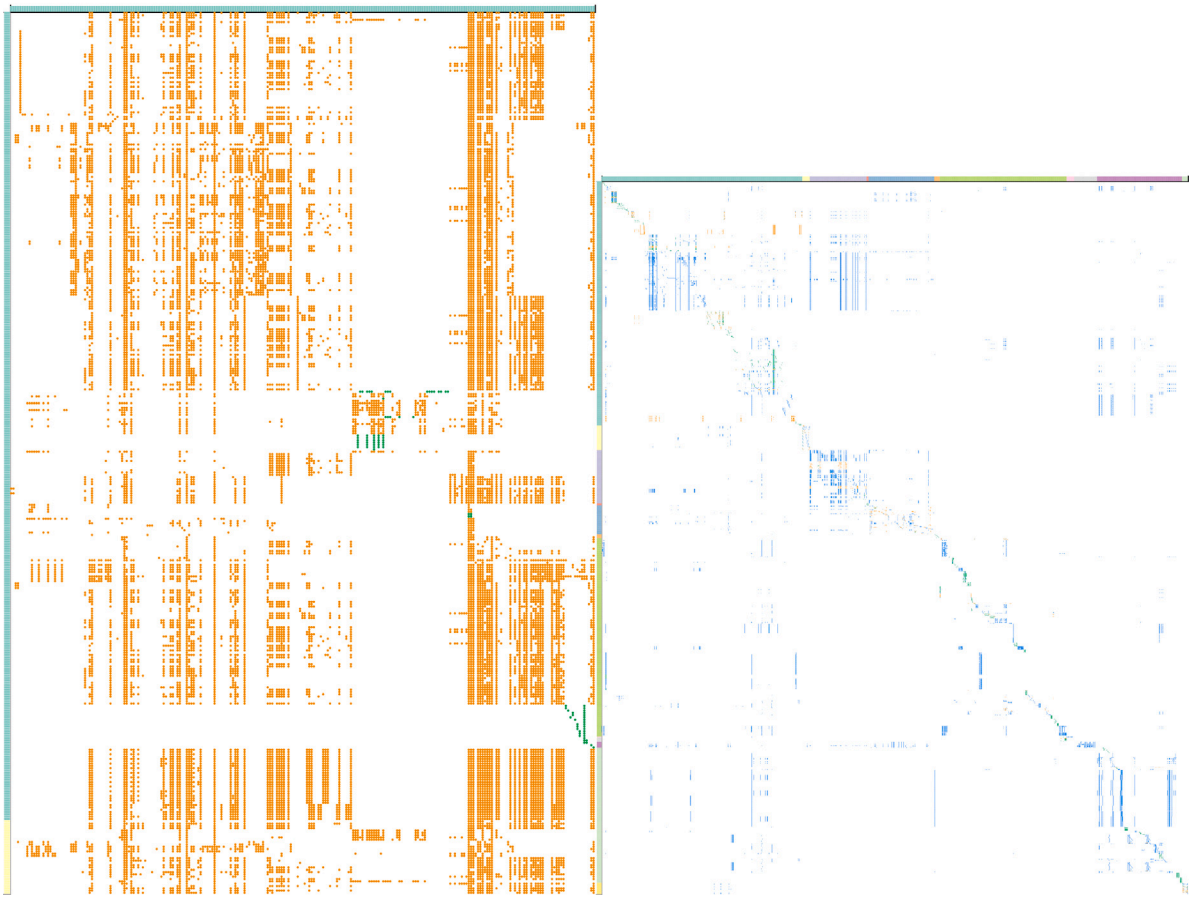
Jsoup structures their tests mainly around a small XML string that is being parsed by the library and finally, the results are verified. As a result, the majority of the tests are variations of each other, with each test covering different corner cases.

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

Now, consider the second aspect: difference in sparseness across the visualizations. Both Jsoup and Commons-CLI have denser visualizations, while Maven and Commons-IO are more sparse. This can be attributed, in part, to the composition of the test suites. Table 3 shows the distribution of tests within the four projects. Jsoup is comprised almost solely of system tests, whereas Commons-CLI is comprised almost solely of integration tests due to all classes living in the same package. As mentioned before, Maven also exhibits long vertical lines, but not as much; as evident in its sparser test distribution in comparison to Jsoup and Commons-CLI. Finally, one sees that Commons-IO focuses more on unit tests, as reflected in the sparseness of Morpheus; and this result matches our expectation for a utility library that contains loosely coupled methods. We also observe similar patterns across a collection of 22 additional subjects programs that we visualized using Morpheus, as shown in the online appendix[1].



(a) Maven. (b) Jsoup.



(c) Commons-CLI. (d) Commons-IO.

Fig. 8. Morpheus visualizing the test suites for four projects: Maven, Jsoup, Commons-CLI, and Commons-IO (blue indicates system tests, orange indicates integration tests, and green indicates unit tests). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

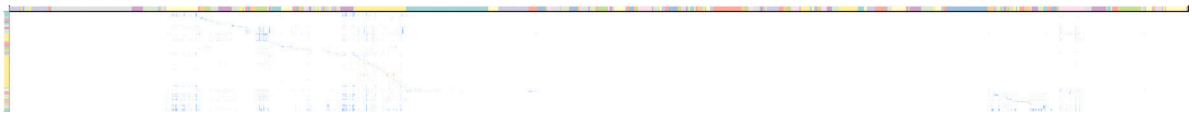
Table 3
Distribution of type of tests (in percentage).

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

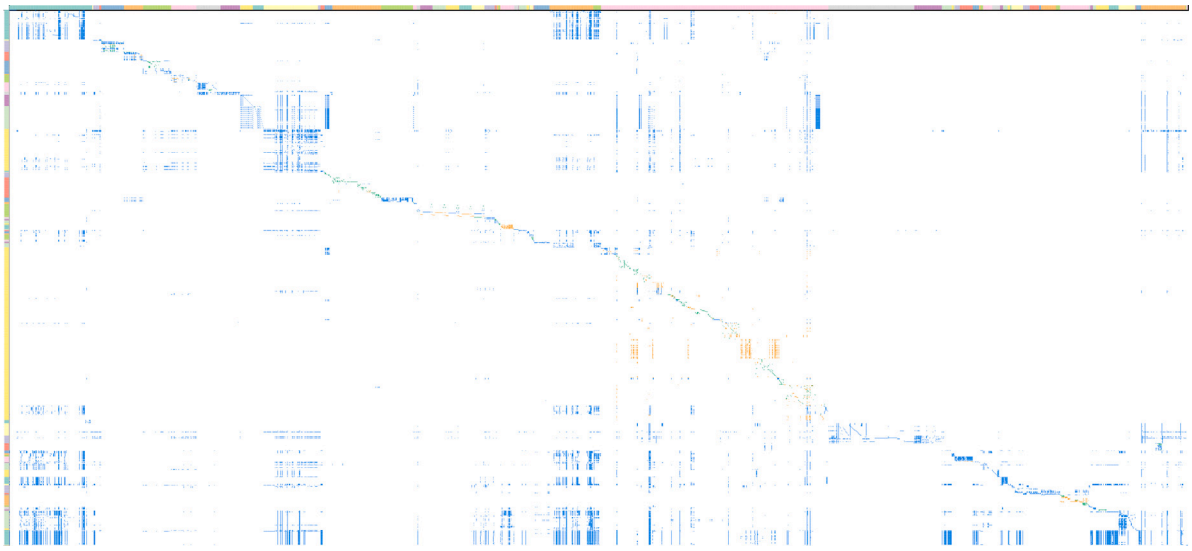
5.3.2. Sparseness reveals untested methods

When comparing the test matrices of various projects we found that several of them to be very sparse. Figs. 9(a) and 10(a) show instances of projects that exhibited sparse matrices. Even with features like zooming, the sparseness of the visualizations made it hard to infer anything meaningful about the projects' test coverage. We observe visually that the white-space when showing all methods for those projects overwhelmed the renders produced by Morpheus.

However, upon filtering away methods that were not executed by any test in those projects, we note changes to the shape and detail within the coverage matrices. Figs. 9(b) and 10(b) show the contrasting views where Morpheus has filtered away any uncovered methods. For both projects (Dubbo and iText7), we observe that these filtered views readily reveals patterns in the test coverage and are more amenable to

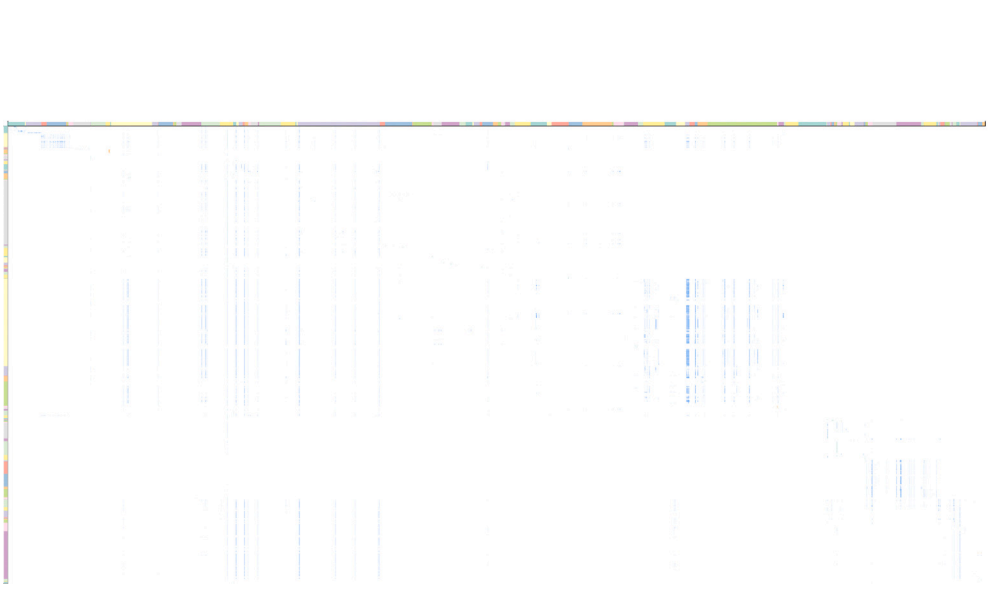


(a) Dubbo: all methods (12404 methods).

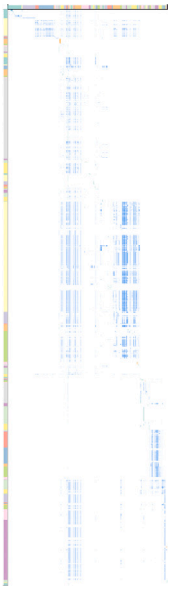


(b) Dubbo: only executed methods (2296 methods).

Fig. 9. Morpheus visualizing test coverage for Dubbo, showing (a) all methods and (b) only executed methods, by 1042 tests. The contrast of these two images reveals the high percentage of methods that are untested.



(a) iText7: all methods (11661 methods).



(b) iText7: only executed methods (1512 methods).

Fig. 10. Morpheus visualizing test coverage for iText7, showing (a) all methods and (b) only executed methods by 5456 tests. The contrast of these two images reveals the high percentage of methods that are untested.

exploration with pan-and-zoom, given the reduction of empty white-spaces. Further, in the case of iText7 we find that the visual shape of the text matrix itself changes: from showing more methods than tests, to a matrix that shows remarkably more methods for the tests that they cover.

Global overviews of these projects' test coverage visually hint at a sizable portion of untested methods within the projects, by the degree of the sparseness depicted by empty white-spaces. We envision that developers can use such sparse visuals to gain high-level summaries of their test coverage, which can then prompt investigations into further

opportunities to improve test coverage. We also note that Morpheus revealed the degree to which the projects are not tested, when presented in contrasting views, like in Figs. 9 and 10.

5.3.3. Case study inference

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

5.4. Project-history case study

We executed, analyzed, and visualized the evolution of test-case execution over many versions of our projects. Through this process, we discovered a number of phenomena that were revealed through the visualization. Namely, we discovered:

1. Growth in project- and test-code over time, for many of the projects.
2. Evolution of coverage by test cases, *e.g.,* for some specific test cases, the number of methods that they execute grows as the project evolves, and for other test cases, the number of methods that they execute stays fairly consistent.
3. Evidence of various refactorings of the code, which are revealed through changes in the test-execution coverage over the history of the project, *e.g.,* changing from traditional test cases to parameterized, refactoring method encapsulation and redirection, and refactoring the code structure.

5.4.1. Growth in project- and test-code over time

To illustrate the growth of a project (and its test suite and coverage) over time, consider Jsoup. Fig. 11 shows three different snapshots of Jsoup's coverage matrix, at different moments in its evolution, as generated by Morpheus. Fig. 11(a) shows how Jsoup was executing 291 methods using 107 tests in February 2010. By September 30th 2021 those numbers grow to 1345 methods and 964 testcases, as shown in Fig. 11(c).

To enable easy visual comparison, the three matrices in Fig. 11 are scaled proportionally to each other and the number of tests and methods they represent on their axes. And so, we observe that the size of the test suite and the resulting matrix has grow significantly. Between 2010 and 2016, the size of the test matrix has grown nearly 4-times (see Fig. 11(b)). This growth is attributed to both: growth in tests (an approx. 4-fold increase from 107 to 477 tests), and growth in methods (an approx. 3-fold increase from 291 to 1026 methods). And by 2021, the test matrix grows to more than twice its size from 2016 — largely due to a doubling of the test suite (from 477 tests to 964 tests).

Even without the specific number of tests and methods, we find that their relative growth is evident by the juxtaposition of the different test matrices rendered by Morpheus, like in Fig. 11. Indeed, we observed similar phenomena around growth in project- and test-sizes in other projects that we mined, *i.e.,* Morpheus-generated test matrices, over a project's history, are able to visually reveal such growth patterns.

5.4.2. Evolution of coverage by individual test cases

As developers, we expect to see an increase in test coverage when we write more tests. Indeed, the steady rise in the number of tests within projects (*e.g.,* Jsoup), reinforced our expectations as developers. However, we also found a rise in the number of methods executed by individual tests, over successive commits, when exploring the evolution of specific tests using Morpheus.

When examining Jsoup's tests we found several instances where the number of methods executed by tests would grow steadily over time. Fig. 12 shows four such instances where the number of methods executed by individual tests grew over time, with successive commits. Fig. 12 shows the evolution of four tests within Jsoup: `ParseTest.testNewsHomepage`, `ElementsTest.filter`, `HtmlParserTest.handlesUnclosedDefinitionLists`, and `SelectorTest.testByAttribute`. The y-axis of those four matrices depicts the different methods executed by the respective tests. The x-axis in each matrix shows the different commits in which we mined those test executions. The green dots/cells shows how the respective test executed a specific method during a specific code commit (as conceptualized in Fig. 3(b)).

In each matrix in Fig. 12, the black artifact bars along the y-axis denotes the number of methods executed by the respective test for a given commit. As such, the black bar plot along the y-axes of the 4 matrices show us that with each successive commit (ordered chronologically) the number of methods executed by each of the four tests grows over time. The prevalence of system-level tests in Jsoup is one possible explanation of such growth in the coverage levels of individual tests — where each test seems to execute nearly every method in the project. Such growth in coverage by individual tests shows that growth in coverage may not only happen with the addition of more tests, but also by existing tests covering more methods (or code) as the project evolves.

5.4.3. Evidence of various refactorings of the code

Through visualizing changes in the execution coverage of the test suite and project, we discovered several interesting phenomena, which upon further investigation revealed a variety of types of refactorings of the code and the test suites.

Changing from Traditional Test Cases to Parameterized. We found a remarkable change in test coverage in the Commons-IO project (See Fig. 13(a)) We examined the test-coverage history for the method `void needNewBuffer(int)` in the `ByteArrayOutputStream` class, and found that up until a point in history, that method was being executed by a smaller set of test cases, and then after that point, each of those test cases unfolded to a number of test cases. When we examined the referenced code, we found that these are examples of test cases that previously were written as traditional test cases, with singular test-case inputs, but after the changing commit became parameterized test cases, each providing a series of input/expected-output pairs.

Refactoring of Test Code Structure. When looking at the history Jsoup, we also found some remarkable changes in test coverage (See Fig. 13(b)). We examined the test-coverage history for the method `String normaliseWhitespace(String)` in the `TextNode` class. At the point in history marked with ⓐ in Fig. 13(b), we noticed a number of test cases that cease to execute the method, and an equal number of new test cases that were henceforth executing it. Upon investigation, we found that the test cases were restructured at this point in the history of the project: `ParserTests` became `HTMLParserTests`. This is an example of the coverage history revealing refactoring changes to the testing code.

Refactoring Method Encapsulation and Redirection. Looking at the same Jsoup method, we also see a remarkable drop in the number of test cases that execute it at the point in time marked with ⓑ in Fig. 13(b). When we examined the referenced code before and after this commit, we found a different phenomenon from the prior observation: here we find a refactoring change to the code, itself. Prior to ⓑ, method calls from many places in the codebase were directed to `TextNode.normaliseWhitespace(String)`, which is simply a wrapper method that passes all requests onto `StringUtil.normaliseWhitespace(String)`. After ⓑ, much of the Jsoup code was updated to call directly into the `StringUtil` method, rather than going through the `TextNode` wrapper — the test cases that were previously executing this method were now bypassing this wrapper method, and hence the remarkable drop in coverage. However, we note that the coverage did not drop to zero: still there are a few remaining calls into the wrapper method, which implies that such a visualization may reveal an incomplete refactoring and could direct further maintenance efforts.

(a) February 5 2010, 291 methods, 107 testcases.

(b) April 16 2016, 1026 methods, 477 testcases

(c) September 30th 2021, 1345 methods, 964 testcases

Fig. 11. Evolution of Jsoup's test suite execution, visualized by Morpheus for three different commits (scaled relatively). Tests on the y-axis, Methods on the x-axis.

5.4.4. Project-history case study inference

Morpheus is able to reveal test-execution evolution phenomena that give insights into the history of the project, and have implications that can inform future development and maintenance.

6. Discussion

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

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

For **RQ2**, the user-study participants needed to assess the composition of the test suite of unit, integration, and system tests for Task 3. Moreover, using Commons-CLI's test suite as a case study, we assess if Morpheus is able to highlight key aspects of the project's test suite. Morpheus's sort, filter, and coloring functionalities enabled us to breakdown Commons-CLI's tests into different types: 326 integration tests, and

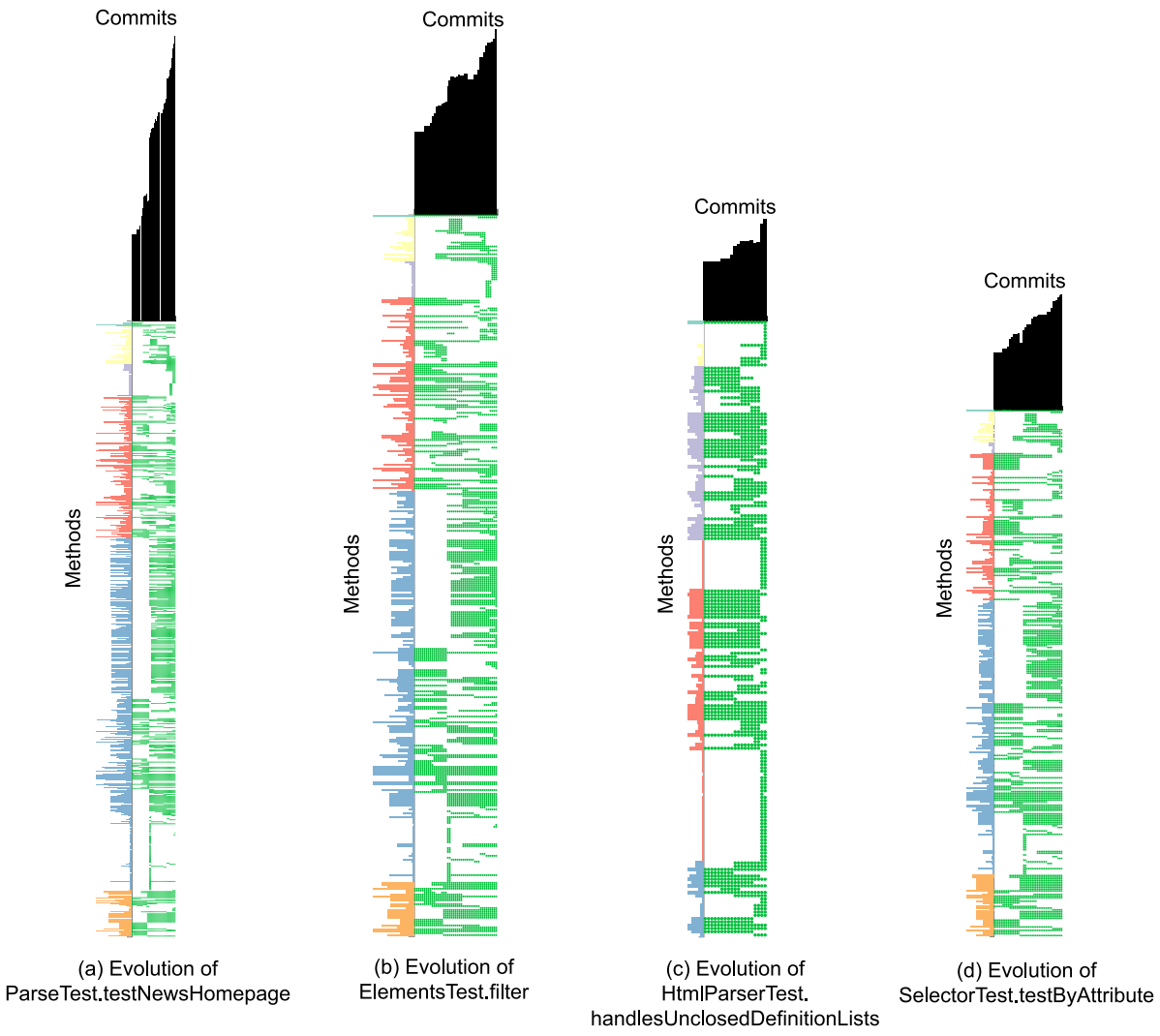


Fig. 12. Examples of Jsoup's tests that saw steady rise in the number of tests they executed over successive commits.

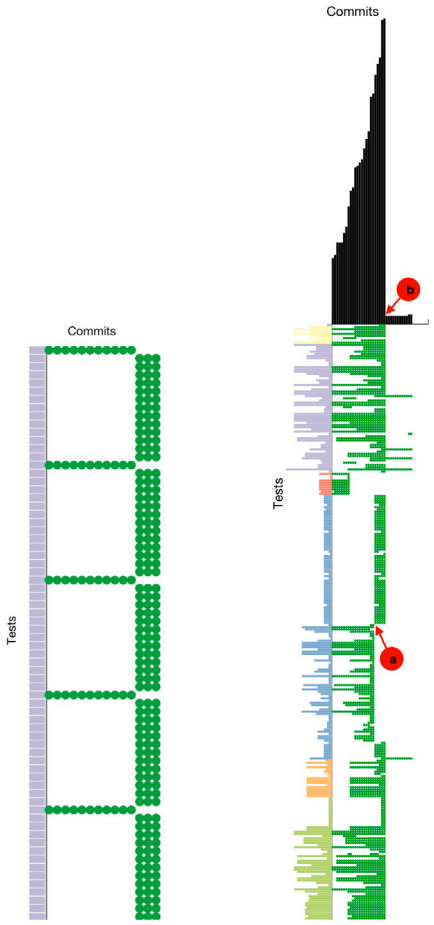
29 unit tests; suggesting a set of highly interdependent methods that invoke together across a large swath of the project's tests. Morpheus was also able visualize methods with sparse, or no test coverage; revealing opportunities to expand test coverage in Commons-CLI. Finally, Morpheus was successful at highlighting overall form differences among multiple projects in our cross-project-comparison case study, which also demonstrates its ability to reveal global overviews of test behavior. As such for **RQ2**, we are able to answer:

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

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

> Morpheus was able to highlight notable differences and similarities in the global structure (or form) for test suites across four independent real-world software systems. In doing so, we were also able to gain insights about the architecture of the software systems themselves.

For **RQ4**, we presented a project-history case study, in which we applied Morpheus to analyze the changing history of test coverage data. To do this, we built, executed, analyzed, and visualized over 400 versions of the software projects in our studies. The study reveals that by visualizing the changes in test coverage, Morpheus reveals remarkable changes in both the code and the test suite, and moreover in the execution of the code by the test suite. We found examples of (1) growth (in code, tests, and execution) over the evolution of the projects, (2) evolution of individual test-case execution coverage that reveals how an existing test case changes in its function as the project evolves, (3) evolution of individual methods' execution by test cases that reveals how an existing method may continue to be more thoroughly tested as the project evolves, and (4) evidence that reveals various forms of refactoring of the code and test suites. For **RQ4** we conclude in the affirmative:



(a) Cropped history view of Commons-IO. (b) History view of Jsoup.

Fig. 13. Refactorings that affect test execution. View (a) reveals traditional test cases that were converted to parameterized test cases, and View (b) reveals refactorings of code structure (at Label ⓐ), and method encapsulation (at Label ⓑ).

> Morpheus was able to reveal patterns in the evolution of a projects test suite over multiple code commits. In doing so, insights about the current state of the thoroughness of the testing can be ascertained in relation to how thorough it was in the past, and potential future maintenance activities may be suggested (*e.g.,* incomplete refactorings).

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

7. Threats to validity

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

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

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

8. Related works

Relationship between test and production code

Yu et al. [11] studied the different factors that impact a developer's ability to understand their test suite. They note that a mental model of the system under test, based on prior knowledge, helps improve an engineer's comprehension of the project's test suite. Morpheus leverages this insight to enable developers in tracing relationships between tests and production code. Prior works have explored using per-test-case coverage to aid developers for tasks such as fault localization [12]. Others focused on helping developers localize what has been tested and by what (*e.g.,* [13–20]). Van Rompaey and Demeyer [21] even propose approaches other than runtime analysis, to establish relations between production and test code, *e.g.,* test naming and design conventions,

static call graphs, lexical analysis and version log mining. Morpheus visualizes dynamically observed, per-test-case code coverage data to reveal traceability between test- and production-code as well. However, Morpheus does so in the global context of test cases and production code that house individual code-to-test relationships, enabling answers to questions such as, "what other tests are failing when executing a given method?"

Synchronous co-evolution of tests and code has received prior study and investigation (*e.g.,* [22–24]). Zaidman et al. [22] and Ens et al. [23] show that there is often a synchronous co-evolution of tests and code. Wang et al. [24] study co-evolution of tests and code, in attempt to help to identify outdated tests. 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. In this current work we also studies the relation between tests and code. However, instead of addressing questions of co-evolution, we analyze the execution of the code by the test cases.

The study of the evolution of code coverage has been studied by Hilton et al. [25]. They study the impact of code-changing patches on code coverage to assess the impact of such patches. In contrast, in this current work, we develop an interactive tool and visualization to allow developers to gain insights and explore their test coverage.

Dynamic behavior comprehension

Multiple prior works have studied comprehension of software behavior. Such works typically reveal relationships between different parts of a project's source code, typically using visualizations (*e.g.,* [26–29]). Similarly, comprehension of software execution traces, aided with visualization also has received prior study (*e.g.,* [30,31]). The main purpose of such works is to understand a single execution trace for a specific software program. Executions from test-runs can aid in understanding production code. Prior works looked to extract product use-case diagrams based on the behavior of a single test [32,33].

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

Matrix-based visualizations

Prior works in information-visualization research have employed matrix-based visualizations for a wide variety of applications. Fernandez et al. [34] created Clustergrammer, a tool to visualize high-dimensional biological data as a matrix visualization. Similarly, matrices have been used to visualize social networks [35–37]. Prior work has shown the advantage of matrix-based visualizations [38–40] to explore graph data at many levels. Ghoniem et al. [41] shows that the readability of matrix-based visualization outperforms node-link diagrams when graphs become bigger than twenty vertices. Our work too presents high-density information in a matrix-based visualizations. However, the information that we visualize is software test coverage data as applied specifically to the field of software engineering.

Researchers have studied the strengths and limitations of matrix-based visualizations compared to other visualizations. Okeo and Jianu [42] observed that "matrices favor dense networks but not sparse ones (empty matrices are as large as dense ones)". For our purposes, such perceived sparseness of the matrix is a benefit — it serves to highlight an absence of test coverage. Ghoniem et al. [43] conducted a study to compare users of matrix visualizations versus node-link visualizations and found that matrices were generally favored. They found that "only path finding is consistently in favor of node-link diagrams throughout the evaluation". For our purposes, we are representing an N (test cases) to M (methods) relationship, as opposed to a N-to-N graph (with N nodes on one axis and the same N nodes on the other axis). As such, traversing paths in the graph is not a need for our approach. Tilstra et al. [44] analyzed the comparative benefits of matrix versus graph visualizations for the specialized purpose of product design. They noted that "[Matrices] can present an overwhelming number of pairs for examination". This observation is indeed relevant to our task, which we address with the interactive features of Morpheus, such as filtering, sorting, coloring, zooming, and panning.

9. Conclusions

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

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

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

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

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.infsof.2022.107085. The supplement includes visualizations for all 26 projects, a video demonstration of the tool, and a replication package for our user study.

References

[1] S. Vasanthapriyan, J. Tian, D. Zhao, S. Xiong, J. Xiang, An ontology-based knowledge sharing portal for software testing, in: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C, 2017, pp. 472–479, http://dx.doi.org/10.1109/QRS-C.2017.82.

[2] D.M. Rafi, K.R.K. Moses, K. Petersen, M.V. Mäntylä, Benefits and limitations of automated software testing: Systematic literature review and practitioner survey, in: 2012 7th International Workshop on Automation of Software Test, AST, 2012, pp. 36–42, http://dx.doi.org/10.1109/IWAST.2012.6228988.

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

[4] E. Daka, G. Fraser, A survey on unit testing practices and problems, in: 2014 IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 201–211, http://dx.doi.org/10.1109/ISSRE.2014.11.

[5] L. Zhao, S. Elbaum, Quality assurance under the open source development model, J. Syst. Softw. 66 (1) (2003) 65–75, http://dx.doi.org/10.1016/S0164-1212(02)00064-X, URL https://www.sciencedirect.com/science/article/pii/S016412120200064X.

[6] R. Torkar, S. Mankefors-Christiernin, A survey on testing and reuse, ISBN: 0-7695-2047-2, 2003, pp. 164–173, http://dx.doi.org/10.1109/SWSTE.2003.1245437.

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

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

[9] Jacoco, URL https://www.eclemma.org/jacoco/.

[10] M. Bostock, V. Ogievetsky, J. Heer, D^3 data-driven documents, IEEE Trans. Visual. Comput. Graphics 17 (12) (2011) 2301–2309.

[11] C.S. Yu, C. Treude, M. Aniche, Comprehending test code: An empirical study, in: 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2019, pp. 501–512, http://dx.doi.org/10.1109/ICSME.2019.00084.

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

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

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

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

[16] M. Hammad, A.F. Otoom, M. Hammad, N. Al-Jawabreh, R. Abu Seini, Multiview visualization of software testing results, Int. J. Comput. Digital Syst. 9 (1) (2020).

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

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

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

[20] A. Rodrigues, M. Lencastre, A.d.A. Gilberto Filho, Multi-VisioTrace: traceability visualization tool, in: 2016 10th International Conference on the Quality of Information and Communications Technology, QUATIC, IEEE, 2016, pp. 61–66.

[21] B. Van Rompaey, S. Demeyer, Establishing traceability links between unit test cases and units under test, in: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, 2009, pp. 209–218.

[22] A. Zaidman, B. Van Rompaey, A. van Deursen, S. Demeyer, Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining, Empir. Softw. Eng. 16 (3) (2011) 325–364.

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

[24] S. Wang, M. Wen, Y. Liu, Y. Wang, R. Wu, Understanding and facilitating the co-evolution of production and test code, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2021, pp. 272–283.

[25] M. Hilton, J. Bell, D. Marinov, A large-scale study of test coverage evolution, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 53–63.

[26] A. Kuhn, D. Erni, P. Loretan, O. Nierstrasz, Software cartography: thematic software visualization with consistent layout, J. Softw. Mainten. Evol. Res. Practice 22 (3) (2010) 191–210, http://dx.doi.org/10.1002/smr.414.

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

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

[29] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, M. Duchrow, Cluster analysis of java dependency graphs, in: Proceedings of the 4th ACM Symposium on Software Visualization, SoftVis '08, ACM, New York, NY, USA, 2008, pp. 91–94, http://dx.doi.org/10.1145/1409720.1409735, URL http://doi.acm.org/10.1145/1409720.1409735.

[30] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J.J. van Wijk, A. 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, IEEE Computer Society, Washington, DC, USA, 2007, pp. 49–58, http://dx.doi.org/10.1109/ICPC.2007.39.

[31] Y. Feng, K. Dreef, J.A. Jones, A. van Deursen, Hierarchical abstraction of execution traces for program comprehension, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 86–96, http://dx.doi.org/10.1145/3196321.3196343.

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

[33] S.M. Nasehi, F. Maurer, Unit tests as API usage examples, in: 2010 IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–10.

[34] N.F. Fernandez, G.W. Gundersen, A. Rahman, M.L. Grimes, K. Rikova, P. Hornbeck, A. Ma'ayan, Clustergrammer, a web-based heatmap visualization and analysis tool for high-dimensional biological data, Sci. Data 4 (2017) 170151.

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

[36] J.S. Yi, N. Elmqvist, S. Lee, TimeMatrix: Analyzing temporal social networks using interactive matrix-based visualizations, Intl. J. Hum. Comput. Interact. 26 (11–12) (2010) 1031–1051.

[37] N. Henry, J.-D. Fekete, M.J. McGuffin, Nodetrix: a hybrid visualization of social networks, IEEE Trans. Vis. Comput. Graphics 13 (6) (2007) 1302–1309.

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

[39] J. Abello, F. Van Ham, Matrix zoom: A visual interface to semi-external graphs, in: IEEE Symposium on Information Visualization, IEEE, 2004, pp. 183–190.

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

[41] M. Ghoniem, J.-D. Fekete, P. Castagliola, On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis, Inform. Visual. 4 (2) (2005) 114–135.

[42] M. Okoe, R. Jianu, S. Kobourov, Node-link or adjacency matrices: Old question, new insights, IEEE Trans. Vis. Comput. Graphics 25 (10) (2018) 2940–2952.

[43] M. Ghoniem, J.-D. Fekete, P. Castagliola, A comparison of the readability of graphs using node-link and matrix-based representations, in: IEEE Symposium on Information Visualization, Ieee, 2004, pp. 17–24.

[44] A.H. Tilstra, M.I. Campbell, K.L. Wood, C.C. Seepersad, Comparing matrix-based and graph-based representations for product design, in: DSM 2010: Proceedings of the 12th International DSM Conference, Cambridge, UK, 22.-23.07. 2010, 2010.