TestAxis

Save Time Fixing Broken CI Builds Without Leaving Your IDE

Casper Boone

TU Delft

TESTAXIS: Save Time Fixing Broken CI Builds Without Leaving Your IDE

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Casper Boone
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
ewi.tudelft.nl

TESTAXIS: Save Time Fixing Broken CI Builds Without Leaving Your IDE

Author: Casper Boone
Student id: 4482107
Email: mail@casperboone.nl

Abstract

The most common reason for Continuous Integration (CI) build failures is failing tests. When a build fails, a developer often has to scroll through hundreds to thousands of log lines to find which test is failing and why. Finding the issue is a tedious process that relies on a developer's experience and increases the cost of software testing. Providing CI build test results with additional context in the developer's local development environment could help solve failing tests more quickly. We propose TESTAXIS, a test result inspection tool that brings CI test results to the Integrated Development Environment (IDE) offering an experience similar to running a test locally. Moreover, it surfaces additional information that is too expensive to collect in local development, for example, a unique view of the code under test that was changed leading up to the build failure. We implement TESTAXIS as a plugin for IntelliJ and conduct a user study to evaluate its usefulness and performance benefits. The participants solve programming assignments evaluating the three main features: the test results overview, the test code editor, and the changed code under test display. We show that TESTAXIS helps developers fix failing tests 13.4% to 30.4% faster. The participants found the features of TESTAXIS useful and would incorporate it in their development workflow to save time. With TESTAXIS we set an important step towards removing the need to manually inspect build logs and bringing CI build results to the IDE, ultimately saving developers time.

Thesis Committee:

Chair:	Prof. dr. A.E. Zaidman, TU Delft
Committee Member:	Dr. A. Katsifodimos, TU Delft
Committee Member:	C.E. Brandt, TU Delft

Preface

Nine months ago I started this turbulent journey called a thesis project. Apparently, that is a thing people do to at the end of their Master's, so I wanted to give it a try as well. Since you are looking at a thesis that (at least) I consider to be finished, I think that worked out. While I was working on my small project to improve a developer's quality of life, many scientists showed the world the importance and power of science by making extremely impressive efforts towards fighting the global COVID-19 pandemic. Although these times required physical isolation, I was never isolated from the many people that helped me make this thesis possible.

First, I would like to thank my supervisors, Andy Zaidman and Carolin Brandt. Andy supported my initial research idea for this thesis from the start and helped me refine it to a concrete research proposal (while thankfully preventing me from wanting to do *too much* on multiple occasions). I am very grateful for Andy's openness to new ideas, feedback on my work, and help in finding concrete solutions to certain problems. During the project, I had the pleasure to discuss my work with Carolin every week. Carolin always provided very useful and detailed feedback on my work. She motivated me to go the extra mile while also making sure that I would finish my thesis according to planning.

Second, I would like to thank the participants of the experiment. All 16 participants freed up 1,5 hours of their busy lives to provide me with invaluable feedback on the project. I appreciate their willingness to participate in the experiment and their openness in sharing their opinions.

Last, I obviously want to thank my family and friends for their support through this journey. From the weekly thesis coffee break discussions with Joël, to their help in finding participants for the (pilot) experiment. I could not have finished this work without the great care and support of my boyfriend Bouke who always followed my thesis endeavors closely, if only due to the working-from-home situation.

Casper Boone
Delft, the Netherlands
June 11, 2021

Contents

List of Figures

Chapter 1

Introduction

Continuous Integration (CI) is a common software engineering practice where functional and quality attributes of a software application are verified after a change to the source code of the application. CI is a wide-spread practice in both industry and open-source projects [27, 62]. It was originally introduced by Booch et al. in 1994 [11] and it became part of the Microsoft and the Extreme Programming practices a few years later [18, 5]. The goal is to detect potential issues with the integration of a new change in the main codebase as soon as possible by providing early feedback before a change makes it to production. This avoids defects but also increases developer productivity [39], accelerates release frequency [24, 27], and, as a side effect, improves communication of changes [20].

These potential issues are detected during a CI build that is usually triggered automatically after the CI provider detects a change in the Version Control System (VCS). A typical CI build comprises building the application to ensure the code compiles, executing the tests to check whether the application still works as expected, and running static analysis tools to safeguard the quality of the codebase and detect potential issues. If any of these steps fail, the whole CI build is considered to be failing, which is sometimes referred to as a "broken build". In this thesis, we focus on builds that break due to failing tests, since failing tests are the most common reason for build failures [7, 65, 48].

When a build fails, the developer has to find and investigate the cause of the build failure. The build results are usually presented in the form of a build log consisting of hundreds to thousands of lines of output by, for example, build tools, compilers, test frameworks, or static analysis tools. The typical steps when encountering a build failure are inspecting the build log, developing a hypothesis about why the build is failing, confirming this hypothesis in a local development environment, and finally implementing a fix [67]. CI build logs typically contain a lot of irrelevant information, and developers indicate to feel overwhelmed by the amount of detail [2]. This makes finding the issue causing the failure a tedious and challenging process that relies on a developer's experience and intuition [67, 28] which increases the cost of software testing [63]. Developers could be supported in the build-fixing process by offering better debugging assistance that obviates the need for manual build log inspection [28].

In case a build breaks due to a failing test, CI providers offer very limited inspection and debugging functionality compared to a local development environment [27]. Commonly

used techniques in a local development environment, such as a debugger or quick navigation from tests to the code under test, are not available on CI platforms. Using such techniques requires a context switch to an integrated development environment (IDE) after developing an intuition on the CI platform. Providing access to more information about failing tests in a developer's local development environment could help them solve failing tests more quickly.

1.1 Problem Statement

We identify the following four problems that may arise in a developer's typical CI or test-fixing workflow:

1. **Build Failure Cause Identification** A In a build log of hundreds to thousands of lines of code, it is difficult to identify why a build is failing [2, 28]. To make it easier to find the cause of a failing build, better assistance is needed to guide developers through the build log [28]. In this thesis, we mainly focus on failing builds due to test failures, the major reason why builds fail [7, 65, 48]. Because developers are overwhelmed by the amount of detail in a build log and want to "find the most faults while investigating the fewest log lines possible" [2], it is important to make it easier for developers to know *which* tests failed. Some CI providers show the failed tests after a build failure [58, 45, 23], however, most CI providers do not provide enough interpretation of the build log to make recognizing the failed tests easier and there is no general-purpose solution.

2. **Lack of Assistance When A Test Fails** B A name and possibly a stack trace of a failing test, the information that is typically shown in a build log, is often not enough information to fix a failing test. From the name of a test, it is sometimes unclear what its intent or actions are. Beller et al. found that developers switched windows within five seconds after encountering a build failure [6], indicating the need for additional context about the reason for the failure to fix a failing test. Furthermore, CI providers do not provide developers any practical assistance while fixing tests, such as quick navigation from a stack trace entry to the source code, making failing tests during CI builds considerably harder to fix than when they fail in a local development environment [28]. By not providing the information in the local development environment, developers must switch context between the CI platform and their local development environment, which may be an interference in their development process. A deeper integration between the local development environment and the CI provider is needed to assist the debugging process [6].

3. **Code Under Test Identification** C While inspecting a test failure, it is useful to be able to quickly identify the production code that is targeted by the test, the *code under test*. In fact, reading production code is the first thing the majority of developers do when they encounter a test failure [6]. In strict unit tests, the connection between a test and the code under test is often obvious to the developer. However, most tests in software projects are more involved [64] and for such higher-level tests, it is not always

easy to link a test to the code it is targeting. Discovering the links programmatically is a nontrivial matter since there is rarely an explicit connection between the test method and the code under test [52]. A commonly used technique to establish links is using code coverage [47, 29]. This is, however, a resource-intensive process that slows down local development because this information is only available after a complete test run. Running tests with code coverage collection on CI platforms is less costly because all tests are always run, but most coverage tools do not support tracing coverage back to individual tests and most CI platforms do not offer a way to inspect these per-test coverage results. There is also no further integration between the local development environment and CI platform making use of potentially established links. However, the large amount of code covered by higher-level tests can be overwhelming, resulting in too many links between a test and the code it covers to investigate [13, 47]. Developers need to be assisted in identifying the relevant links to fix the test failure.

4. **Awareness of Build Failures** D One of the goals of continuous integration is to speed up the release cycle by having more confidence in the correct working of the change. However, to make sure there are no unnecessary stalls in this process, it is important to become aware of a build failure quickly so that the issue can be fixed as soon as possible. Especially in a workflow where co-workers need to further process the change or are dependent on it, the time that a build for a code change is failing should be minimized. Kerzazi et al. found that, on average, it takes three hours for developers to become aware of a build failure [32]. Also, only 28% of the 28 developers the authors interviewed pays attention to build notifications, for example via email. A number of developers also indicate to mainly become aware of build failures via colleagues. All mentioned notification approaches require developers to leave their development environment to become aware of build failures. Thus, there is a need for better build notifications in the local development environment.

1.2 Research Questions

We hypothesize that the problems sketched in Section 1.1 can be solved by showing CI build results and additional context to test failures in an IDE. In this thesis, we investigate how providing these different types of context influences the developer's failure-fixing behavior. We measure the impact this has on the time a developer needs to fix a failing test that failed during a CI build and hypothesize that the additional context helps them to fix these tests more quickly. The additional context includes the following aspects: presenting an interactive stack trace, showing the test code, visualizing the changed code under test, and raising warnings on the health and history of a test. To determine whether these aspects save time and are perceived as useful, we pose the research questions below. The research questions address one or more of the problems of Section 1.1. We depict their relationship in Figure 1.1.

RQ1

What is the influence of presenting a test failure in the IDE over a CI build log on the time a developer needs to fix a failing test?

We want to determine the influence of showing *which* tests failed in the local development environment over having to scroll through build logs, to confirm whether this aspect of CI build test result inspection is a successful attempt at solving the *Build Failure Cause Identification* A problem.

RQ2

What is the influence of showing the test code on the time a developer needs to fix a failing test?

We want to determine the influence of showing the test code of a test that failed in a CI build in the local development environment, to confirm whether this aspect of CI build test result inspection is a successful attempt at solving the *Lack of Assistance When A Test Fails* B problem.

RQ3

What is the influence of showing the code under test, where the changed code is highlighted, on the time a developer needs to fix a failing test?

We want to determine the influence of showing the changed code under test of a test that failed in a CI build in the local development environment, to confirm whether this aspect of CI build test result inspection is a successful attempt at solving the *Lack of Assistance When A Test Fails* B and the *Code Under Test Identification* C problem.

RQ4

To what extent do developers prefer to be actively notified of CI build failures in the IDE over their current approach?

In an attempt to solve the *Awareness of Build Failures* D problem, we investigate the hypothesis that CI build notifications in the IDE makes developers more aware of a build failure.

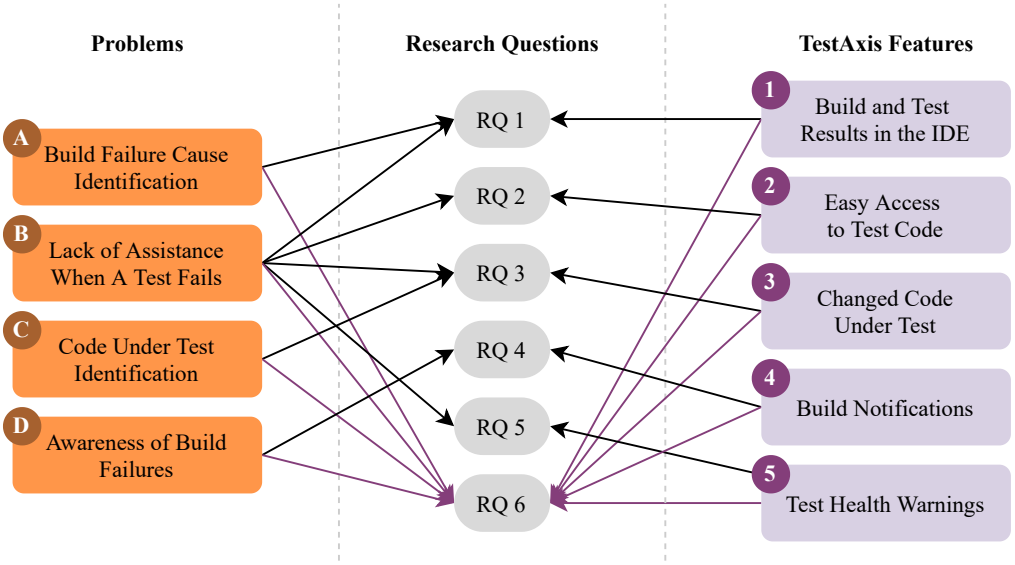


Figure 1.1: The relationship between the problems (Section 1.1), the research questions (Section 1.2), and the features of TESTAXIS (Section 1.3). For RQ6 about the usefulness, we consider *all* aspects of TESTAXIS, indicated by the purple arrows.

RQ5

To what extent do developers find it useful to be warned about the health and history of a failing test?

Based on the (historical) test execution data, developers can be warned about long-running, often-failing, or flaky tests. This may help to solve the *Lack of Assistance When A Test Fails* B problem if developers consider the warnings useful.

RQ6

To what extent do developers consider a CI build test result inspection IDE plugin useful?

A part of the solution to all four problems could be to present CI build information in an IDE plugin. When a build fails due to a failing test, additional context related to the failing test can be provided. In RQ1-RQ3 we ask whether additional context aspects can help developers solve issues more quickly, here we ask whether developers perceive these aspects to be useful.

1.3 Approach

To provide the basis for our research, we start by laying out the current state-of-the-art of CI build test result inspection using three different investigations. First, we conduct an exploratory literature study to identify the relevant research in this area. This research helps to understand the (lack of) currently available solutions for the problems in Section 1.1. It also gives insights into previous attempts at solving these problems and thus directs us in what to investigate next. Second, we systematically study the available CI build result inspection tools in the market places of two widely used IDEs. Lastly, we ask developers about their current workflow when they encounter a failing build in our user study. This gives us insights into whether CI build result inspection tools are commonly used.

To address RQ1-RQ6, we propose and design TESTAXIS: a CI build test result inspection tool. During CI builds, TESTAXIS captures test execution and coverage information. This information is used to present the build results in the TESTAXIS IDE plugin. The IDE plugin shows all failing tests and obviates the need to look at the build log (RQ1, 1). For each failing test, the plugin shows the name of the test, the failure message, and the corresponding stack trace. Moreover, the test code is shown to make it easier to understand the intent of the failing test (RQ2, 2).

TESTAXIS also features an overview of the relevant code under test (RQ3, 3). By collecting coverage per test execution, TESTAXIS is capable of determining which parts of the code are executed by which test. Combining this information with the code changes leading up to the build failure allows TESTAXIS to pinpoint potential locations of issues in the codebase.

By combining the CI build test results and code under test information, TESTAXIS warns the developer, about often-failing, slow, or flaky tests (RQ5, 5). This information can be useful in determining the cause of the failing test.

Furthermore, to improve build failure awareness, TESTAXIS sends out notifications after a build has finished (RQ4, 4). The developer receives this notification in the IDE, where they can immediately act on the failure.

We create a proof-of-concept implementation of TESTAXIS that we use to evaluate the effectiveness and perceived usefulness of several aspects of CI build test result inspection in our experiments. First, we implement a "backend" service that collects and analyzes the build result data provided during CI build runs. Second, we implement an IDE plugin for the IntelliJ IDEA Platform. The plugin provides access to all the features described above to evaluate their effect on a developer's failure-fixing experience. A demonstration of the plugin is available at `https://youtu.be/4sfnKsvqwKw`.

In order to provide an answer to our research questions, we perform experiments with the implementation of TESTAXIS in our user study. The user study consists of two parts: a one-group pretest-posttest study and a within-subjects study. Both parts are pre-experimental, meaning that the outcome of the experiment can only show that *there is* an effect after using TESTAXIS but not that this effect is necessarily *caused by* TESTAXIS. The experiment comprises an opening questionnaire, eight programming assignments, and a closing questionnaire. During the opening questionnaire, we ask participants questions about, for example, their past experience and their habits. After the opening questions, we ask participants to

fix eight randomly assigned failing tests that are part of four categories. Per category, they solve one assignment without and one assignment with the help of TESTAXIS. For the assignments without TESTAXIS, we gave the participants access to a GitHub pull request together with a GitHub Actions build log. The four categories imply the usage of different features of TESTAXIS: simple cases where the issue can be spotted in the stack trace alone (RQ1), cases where the issue is in the test code (RQ2), simple cases where the issue is in the code under test (RQ3), and advanced cases where the issue is in the code under test (RQ3). See Figure 1.1 for the overview of the relationships between the research questions and the features described above. The closing questionnaire asks participants about their perception of the assignments, their experience with TESTAXIS, the aspects they found most useful (RQ6), and their opinions on the build notifications (RQ4) and test health warnings (RQ5).

1.4 Contributions

The contributions of this thesis are fourfold:

1. TESTAXIS: A modular software system that collects and analyzes CI build test results, and that visualizes these results with additional context (such as the test code or the code under test) in the IDE. We make the following open-source software artifacts available:

   a) The TESTAXIS backend that collects and analyzes test reports, and exposes them through an API[1].

   b) The TESTAXIS IDE plugin for IntelliJ Platform IDEs that presents build and tests results to the developer[2].

   c) A Gradle plugin that collects and generates coverage reports per executed test[3].

2. An evaluation of the effect of providing CI build test results with additional context (such as the test code or the code under test) in the IDE on the failure-fixing time performance.

3. An evaluation of the perceived usefulness of a CI build test result inspection tool for the IDE.

4. A publicly available dataset containing the data collected during the experiments [12]. Among others, this includes the timing results, anonymized questionnaire results, notes including observed time spent on actions, and analysis scripts.

[1] Available at `https://github.com/testaxis/testaxis-backend`
[2] Available at `https://github.com/testaxis/testaxis-intellij-plugin`
[3] Available at `https://github.com/testaxis/coverage-per-test-gradle-plugin`

1.5 Thesis Outline

In Chapter 2, we sketch an overview of the conceptual design of TESTAXIS and describe its feature set. Then, in Chapter 3, we present the implementation of the TESTAXIS backend and IDE plugin. We describe the design of the user study in Chapter 4 and the results of the questionnaires and programming assignments from the study in Chapter 5. In Chapter 5, we also analyze and discuss the results to answer RQ1-RQ6. We sketch the current state-of-the art of CI build result inspection and related tools in Chapter 6. Finally, we summarize our findings and discuss opportunities for future work in Chapter 7.

Chapter 2

Conceptual Design and Features

While the features of TESTAXIS were shortly introduced in the introduction, in this chapter, we lay out the conceptual design by describing the features in detail. The conceptual design can be applied to different types of implementations of which we give one example in Chapter 3. To describe how these features may be used in practice, we also describe the use cases of TESTAXIS in the form of functional requirements as part of the requirements analysis. This analysis also contains a discussion of the non-functional requirements of the tool.

2.1 TESTAXIS Features

The functionality of TESTAXIS can be categorized into five main features. These features are: the display of CI build results, the (meta) information about the failure of a test case and its source code, the presentation of the changed code under test, the communication of build failures through notifications, and the display of warnings about the quality of the test suite.

2.1.1 Presentation of CI Build Test Results

The core feature of TESTAXIS where the other features build upon is presenting the CI build test results in a more accessible format than raw build logs. Through communication between the CI build provider and TESTAXIS, the raw results can be parsed and interpreted by TESTAXIS. This makes it possible to indicate the reason for the build failure and to present an overview of the specific tests that failed, obviating the need to inspect the build log manually. Where such build logs often provide a linear listing of the test failures that occurred during the build, in order of the test runs, TESTAXIS groups failures by test class. Figure 2.1 shows an example of test case executions grouped by their class. This provides additional structure to the results, making it easier to identify in which part(s) of the system the failures occur.

TESTAXIS includes the results of all tests, also the ones that passed. The tool can therefore explicitly confirm that a certain test was executed and that it passes, for example when it previously did not pass. It also enables the possibility to identify slow tests that impact the build time, since TESTAXIS displays the run time for each executed test. Finally,

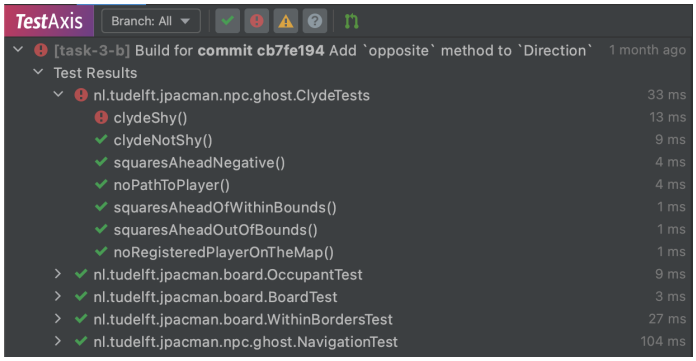


Figure 2.1: An example of the presentation of CI build test results in the IDE implementation of TESTAXIS.

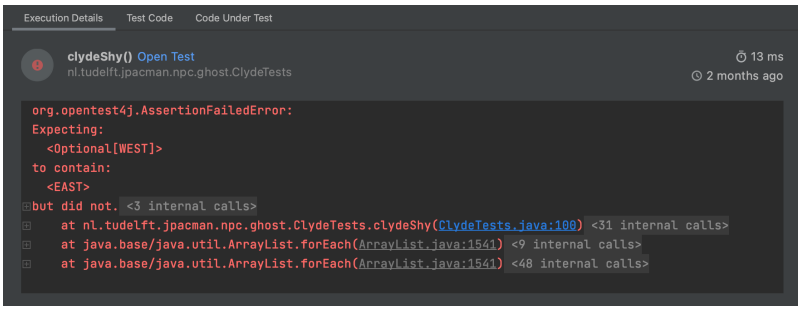


Figure 2.2: An example of the presentation of test failure details in the IDE implementation of TESTAXIS.

it enables further inspection of successful tests using the features described below. This can be useful in a debugging context where you want to compare a failed test run against a successful run of the same test.

For CI build failures due to something other than tests, TESTAXIS indicates that the build has failed due to a reason outside the scope of the tool.

2.1.2 Test Failure Details and Test Code

When a developer inspects a failing test, TESTAXIS shows the basic details of the test, such as the test name, whether the test passed, the run time, and the execution date. While just knowing which test failed may sometimes already provide enough hints to the developer to identify the issue, often this information is too limited. Therefore, TESTAXIS also shows the failure message of the test, as part of the stack trace, see Figure 2.2. The presentation of the stack trace includes links to the mentioned files or classes and, by default, hides any internal language or framework calls. This allows for quick navigation from the failure message to the code which is not available in a CI build log.

In addition to the failure details and interactive stack trace, TESTAXIS also presents the source code of the test under inspection. By providing an easy way to read the test code,

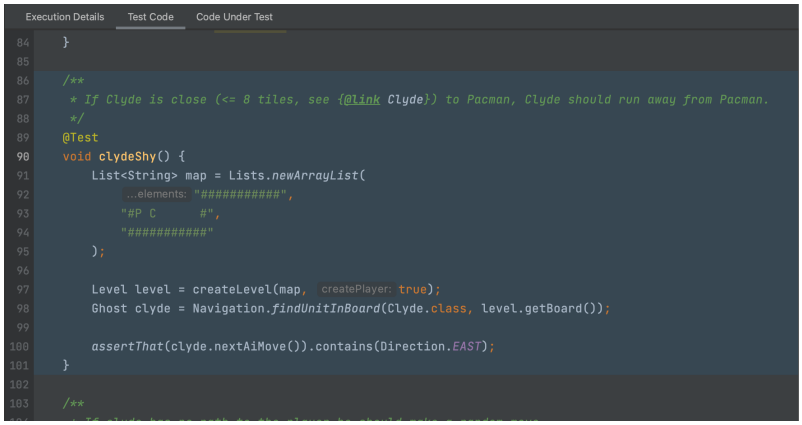


Figure 2.3: An example of the presentation of test source code in the IDE implementation of TESTAXIS.

the need for manual search and navigation is obviated. On one hand, reading the test code may give developers additional context to understand the intent of the test, making it easier to understand why the test is failing. On the other hand, showing the test code may help developers spot obvious mistakes more quickly. TESTAXIS presents the code of the test case in the context of the full test file which potentially helps the developer to identify the test context as well. The code of the test case under inspection is highlighted to provide a visual hint to the developer. Figure 2.3 shows an example of what this looks like in the IDE implementation of TESTAXIS.

2.1.3 (Changed) Code Under Test

The goal of the code under test feature is to highlight the parts of the production code that are likely to contain the issue causing the test to fail. To find out which parts of the production code the test is targeting, TESTAXIS makes use of code coverage. By collecting code coverage *per test* during the CI build run, TESTAXIS knows which lines of the production code are covered by a particular test. Since most CI builds run the whole test suite, collecting coverage as a side-effect is a "cheap" operation in terms of the impact on build performance. Especially compared to the cost of collecting coverage information locally, where it is not always feasible to continuously run all tests.

While this coverage information could be specific enough to investigate failures of strict unit tests with a small set of covered files, most tests are more involved and interact with multiple parts of the codebase [64]. Such tests may therefore cover a large number of lines of the production code, making it difficult to identify the issue. The amount of potentially relevant lines to investigate can, however, be reduced by selecting the locations that were recently changed. Assuming that the test fails due to an intrinsic issue in the code itself [51], it is likely that the issue is located in a part of the production code that is covered by the test but also changed in the commits leading up to the build failure. This cannot be concluded for an extrinsic issue, such as a configuration problem. Since CI builds are

```
/**
 * Creates a new pellet.
 *
 * @return The new pellet.
 */
public Pellet createPellet(int points) {
    return new Pellet(
        PELLET_VALUE,
        sprites.getPelletSprite()
    );
}
```

A) Covered

```
/**
 * Creates a new pellet.
 *
 * @return The new pellet.
 */
public Pellet createPellet(int points) {
    return new Pellet(
        PELLET_VALUE,
        sprites.getPelletSprite()
    );
}
```

B) Changed

```
/**
 * Creates a new pellet.
 *
 * @return The new pellet.
 */
public Pellet createPellet(int points) {
    return new Pellet(
        PELLET_VALUE,
        sprites.getPelletSprite()
    );
}
```

C) Covered and Changed

Figure 2.4: An example of how combining coverage and change information leads to more focused potential issues that require attention.

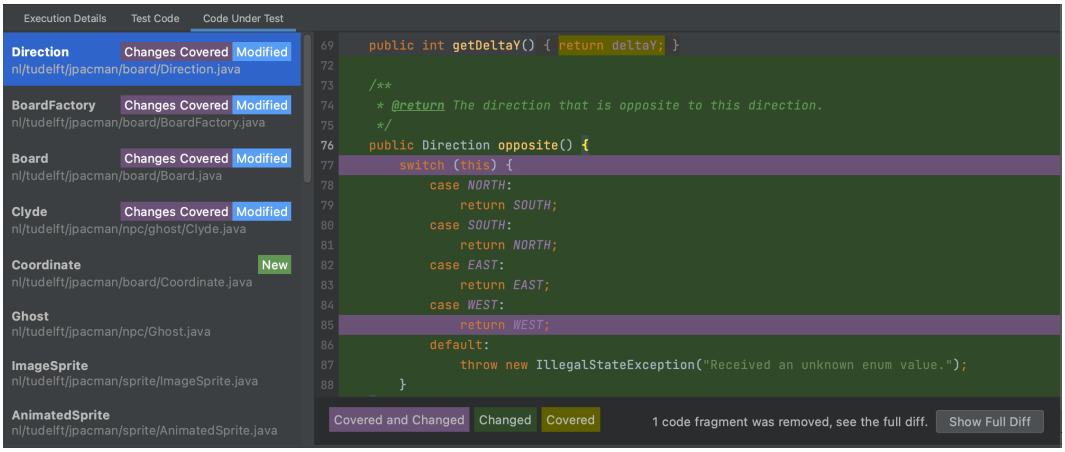


Figure 2.5: An example of the (changed) code under test feature in the IDE implementation of TESTAXIS.

commonly triggered after pushing new commits, TESTAXIS can make use of the full change information that is available through the VCS. Applying this insight to the production code results in a list of potential locations of the issue that can be suggested to the developer to investigate. Figure 2.4 shows how intersecting coverage and change information reduces the locations of interest that a developer must investigate. See Figure 2.5 for an example of the implementation of this feature.

2.1.4 Build Notifications

The communication between the CI build provider and TESTAXIS when a build has finished, provides an opportunity to immediately notify the developer of failing builds. In fact, using the analyses of the build results that TESTAXIS performs, it already includes the reason for the build failure in the notification. TESTAXIS can deliver these notifications in an IDE without relying on external tools or services, such as email or messaging apps. By providing the notifications in the local development environment, the need for a context switch to a different application is obviated. From the notification messages, the developer is

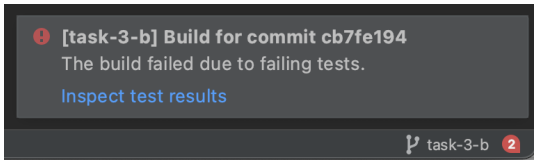


Figure 2.6: An example of a build notification in TESTAXIS.

immediately guided to additional support (the other features of TESTAXIS) that may help solve the build failure. Figure 2.6 shows an example of a build notification in TESTAXIS.

2.1.5 Test Health Warnings

To help developers maintain the quality of their test suite and alert them of potential issues, TESTAXIS raises so-called "health warnings". Once a build has finished, TESTAXIS performs multiple analyses on the results and may compare the build to previous builds. TESTAXIS detects three types of potential issues:

1. **Slow Tests** To make the developer aware of the impact of a particular test on the performance of the test suite, a warning is shown on tests performing slower than other tests in the codebase. A test is considered slow if the run time exceeds two times the average run time of the tests.

2. **Often-Failing Tests** One of the key ideas behind TESTAXIS is that it has a deeper understanding of what is going on during the testing phase of CI builds than the CI providers do. This provides the opportunity to shift the axis from a history of build runs to a history of test executions. Using this history, we can determine when a specific test failed in the past and thus how often that test case fails. As a proof-of-concept, we consider a test to be failing often when it failed 10% of the times in the last 50 builds. A future study could determine what threshold is sensible to developers in practice, or the threshold could be made configurable. If the threshold is crossed, the developer is presented with a warning that suggests investigating why that test fails often and checking whether the test is potentially flaky or too tightly coupled to the production code.

3. **Flaky Tests** Flaky tests are tests that give unreliable and unpredictable results: they sometimes pass and other times fail without any changes to the code [34]. Because the results are non-deterministic, the outcome cannot be trusted. To detect potential flaky tests, TESTAXIS combines the build results and the data from the code under test feature. If a test fails for which there are no changes in the code it covers, the test is likely showing flakiness. When TESTAXIS detects this, it warns the developer about a potential flaky test. Eloussi found that this approach is successful in recognizing flaky tests in more than 70% of the cases investigated in [22]. The warning includes a disclaimer saying that the assumed flakiness may also be explained by an extrinsic issue such as a configuration change because the applied technique will only reliably detect intrinsic issues [51]. In those cases, however, the past behavior of the test is

relevant to figure out whether the test has shown flaky behavior before. The analysis of often-failing tests described above may provide an answer to that.

2.2 Requirements Analysis

To determine the needs of an implementation of TESTAXIS, we analyze the requirements for such a tool. The analysis consists of two types of requirements: functional and non-functional requirements. The functional requirements also serve as an example of the use cases of the TESTAXIS feature set described in the previous section.

2.2.1 Functional Requirements

We list the following functional requirements and describe their use cases in a developer's workflow:

1. **Receiving Notifications**

   1.1. As a developer, I want to receive notifications in my local development environment when my build fails

   1.2. As a developer, I want to click a build notification to inspect the failing build

   When a developer creates a pull request or pushes changes, often a CI runner service starts to build the code and execute the tests. When one of the tests is failing, this results in a CI build failure. To become aware of a build failure, developers usually check the build status on the CI platform or VCS platform manually, or they receive a notification through email or a messaging app. As a developer, removing the need to do manual checks or to switch to external services can increase build failure awareness. This can be accomplished by displaying build notifications in an already used local development environment (requirement 1.1). When a developer reads a notification, they should be guided to the other features of TESTAXIS to support inspecting the build failure (requirement 1.2).

2. **Inspecting Build Results**

   2.1. As a developer, I want to know which tests failed during a CI build

   2.2. As a developer, I want to review the history of failing builds to compare results to the past

   2.3. As a developer, I want to filter the history failing builds on branch, status, and build type

   When the developer inspects a failing build, by clicking the notification, for instance, they can be helped by showing *which tests* caused the build to fail (requirement 2.1). Usually, finding out that failing tests are the cause of a failing build and finding out *which tests* are failing requires opening the website of a CI platform (for example Travis CI or GitHub Actions) and scrolling through the logs. However, as a developer,

this is a tedious process that can be avoided by presenting the build results in a more accessible manner.

Moreover, when a developer wants to compare the results of a build or a specific test from the past, they also can be helped by displaying the history of failing builds and tests. To avoid this, the history should be filterable to assist the search for the right build. It should support filtering on the build status: passed, failed due to tests, failed due to a different reason, or unknown. Furthermore, when changes are pushed to a branch belonging to a pull request, some VCS providers start two builds: a "push" build that builds the current branch and a "pull request" branch that first merges the branch with the main branch and then performs the build. TESTAXIS should therefore also support filtering on build type. Finally, it should allow filtering on the branch.

3. **Inspecting Failing Tests**

   - **Details**

     3.1. As a developer, I want to know the name of a failing test (meta information)

     3.2. As a developer, I want to see the failure message of a failing test

     3.3. As a developer, I want to interact with the stack trace of a failing test

     After the developer has used the functionality resulting from requirements 2.1-2.3, they probably want to see more information about a failing test. The name of the test and/or the stack trace of the failure containing the failure message may already provide enough information to the developer to know why the test is failing. This could for instance be the case when the developer is very familiar with the project and can quickly think of the reason why the test is not passing. To allow for easy navigation through the entries of the stack trace, similar to other tools in their local development environment, the code files referenced in the stack trace should be clickable.

   - **Test Code**

     3.4. As a developer, I want to quickly be able to inspect the test code of a failing

     3.5. As a developer, I want to quickly navigate to the test code in my code editor

     3.6. As a developer, I want to edit the test code from within the tool to make quick fixes

     The developer may have made an obvious mistake in the code of the test itself. The test code should be shown in the overview of the failing test. The developer can then quickly spot the failure and fix the issue inside the tool. The test code can also be used as an information source to find out the intent of the failing test.

   - **Code Under Test**

     3.7. As a developer, I want to know what parts of the production code the test is targeting

     3.8. As a developer, I want to know the likely locations of the issue by reviewing fragments of the code that are both covered and changed

3.9. As a developer, I want to see a full overview of the changes in a file

3.10. As a developer, I want to edit the code under test from within the tool to make quick fixes

The developer may have made a mistake in the code under test. The tool should try to show all relevant code under test that has changed in the commits leading up to the failing build. The developer should be able to quickly browse through the code snippets of the code under test, instead of having to navigate to all related code files manually, to spot the issue and create a fix.

- **Flakiness**

3.11. As a developer, I want to know if flakiness is the reason for a failing test

The reason that the test fails might actually not be related to the code change at all, as a result of flakiness. The tool should suggest this to the developer when no code under test has changed. The suggestion is supported by executions of the test in the past that may have resulted in the same failure.

4. **Evaluating the Test Suite Quality**

4.1. As a developer, I want to know which tests fail often to improve the quality of my test suite and production code

4.2. As a developer, I want to know which tests impact the build time to improve the quality of my test suite

As a developer, one wants to keep the overall quality of their production and test code in a good shape. The tool should provide feedback on two quality measures: failure rate (requirement 4.1) and test run time (requirements 4.2).

Tests that fail often could, among others, indicate the following issues: the test code might be too tight to the production code (capturing how the code works instead of what the code does), the production code might be too tightly coupled (the developer could conclude that the test often fails for changes that are not related to the test), or the test might be flaky.

The test run time could be used to identify long-running tests. The perspective of test execution data provides the opportunity to show warnings about the run time of tests, making it possible to address such tests.

2.2.2 Non-Functional Requirements

The quality of the tool is determined by non-functional requirements and makes the difference between a well-received product and an unused one [50]. These requirements constraint the design choices during the implementation and determine the structural properties of the tool. Amriola et al. state the following six categories for non-functional requirements which we apply to the requirements of the tool [3].

1. **Efficiency** TESTAXIS must be an efficient tool in terms of performance: the build results should be available quickly after the CI build has finished, and exploring the results in the tool should be fast enough to not interrupt a developer's workflow.

2. **Portability** TESTAXIS must provide support for Windows, macOS, and Linux as a part of the IntelliJ IDEA platform IDEs. The tool must support Java and Kotlin.

3. **Maintainability** The code of TESTAXIS should be well maintainable. By choosing a modular architecture that allows for change and extension, maintainability should be achieved. Moreover, testing and static analysis tools should ensure that the quality of the code is maintained.

4. **Expandability** The modular architecture of TESTAXIS should ensure easy expandability and open the possibility to create alternative clients for other platforms. Within the codebases, TESTAXIS should have an architecture that allows for easy extension. Also, it should be possible to add additional programming languages and test frameworks with reasonable effort.

5. **Robustness** The tool should be able to deal with incomplete, corrupt, or failing builds and process the results accordingly. Missing information should not hinder the developer's user experience.

6. **Safety** The data stored by TESTAXIS should be stored securely. The data per user should be separated and no information about unauthorized projects should be accessible. TESTAXIS should implement a proper authentication mechanism to identify a user.

Additionally, we also consider the following two non-functional requirements:

7. **Usability** The tool should be easy to use for developers and match their way of working. Developers are a different type of target audience than end-users and typically quickly understand the intuitions of a software application. Therefore, the focus should be on displaying information as concise and quick to understand as possible without any interference.

8. **User Experience** The experience of using the tool should resemble the fixing process of a local test failure as closely as possible. The tool should integrate seamlessly into the local development environment.

2.3 Summary

In this chapter, we laid out the feature set of TESTAXIS, consisting of five major features, that form the conceptual design of the build result inspection tool. TESTAXIS displays the current CI build results but also allows browsing through past builds. For test results, it displays the meta-information, such as the name of a test, and gives access to an interactive stack trace that can be used to explore the test failure. The changed code under test feature

provides clear hints of the potential location of the issue causing the test to fail based on code coverage and VCS changes. Furthermore, the developer is notified of failing builds through build notifications in the local development. Finally, TESTAXIS raises so-called health warnings about the quality of the test suite about slow, often-failing, or flaky tests.

To confirm the usefulness of these conceptual features, we analyze the requirements of the implementation of a tool with these features. The first part of this analysis describes user stories explaining the potential use cases. The second part is about the non-functional requirements of the application, such as security and user experience.

Chapter 3

Development and Implementation

We created a prototype implementation of TESTAXIS that implements the conceptual design of Chapter 2. The implementation of this prototype tests the practical realizability of a CI build test result inspection tool as described in the previous chapters. We also use this prototype as part of the user study to expose users to TESTAXIS.

We present an overview of the system in Figure 3.1. TESTAXIS consists of two main parts: the backend and the IDE plugin. The backend is an application that runs on a server. It receives and processes incoming reports from CI build providers and prepares them for usage by a client. The application exposes an API that clients can use to retrieve the build results, among others.

An example of such a client is the IDE plugin we developed. We developed a plugin for IDEs built on the IntelliJ Platform, such as IntelliJ IDEA or Android Studio. The plugin presents the build results to the user for inspection. A demonstration of the plugin is available at `https://youtu.be/4sfnKsvqwKw`.

In this chapter, we discuss how the build and test information is gathered, processed, and distributed between the four different components in Figure 3.1. This follows the information flow from the source to the user.

3.1 From a CI Service to the TESTAXIS Backend 1

The first component in the information flow is the CI build provider. As part of a CI build, the CI service is responsible for sending the build results to TESTAXIS. The build results consist of two parts: the test reports and the code coverage reports.

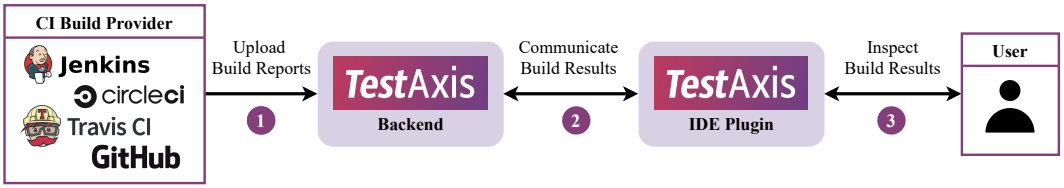


Figure 3.1: System Overview of TESTAXIS.

```
- name: Upload test results to TestAxis
  run: bash <(curl -s https://testaxis.io/testaxis-upload.bash) -s build/test-results/test/ -c build/coveragepertest/xml -p ${{ job.status }}
  if: always()
  env:
    TESTAXIS_TOKEN: ${{ secrets.TESTAXIS_TOKEN }}
```

Figure 3.2: Example of the new build step to add TESTAXIS support to GitHub Actions.

3.1.1 CI-Service-Independent Communication

To start using TESTAXIS in a software project, developers need to add a new step to their build scripts. In this step, the CI build runner downloads a bash script from the TESTAXIS backend. The bash script is based on Codecov's bash uploader[1]. The script detects the CI build provider and gathers meta information such as the current build, commit, branch, repository, etc. Then, it uploads the build reports (see Section 3.1.2 and 3.1.3) to TESTAXIS.

The bash script can auto-detect 27 CI build providers such as Jenkins, Travis CI, or GitHub. For those providers, it will automatically detect all the meta-information. For unsupported providers, these meta properties can be set manually using command line arguments. The report upload phase of TESTAXIS therefore does not depend on any specifically supported CI build providers.

The new build step must be included after compiling the code, running static analysis tools, and executing the tests. Therefore, the developer needs to ensure in their build configuration that the upload step is executed even when the build failed in an earlier step. Otherwise, TESTAXIS will not receive any information about the build and would thus be unaware of the existence of the build. In Figure 3.2, we show an example of what the build step would look like when one would implement it in a GitHub Actions build script. The uploads are protected by the user's access token, see Section 3.3.1.

3.1.2 Collecting Test Reports

Once a build has finished, the test result reports are sent to the TESTAXIS backend. These reports are XML files in the JUnit XML format [30]. This format is supported by many test frameworks, such as JUnit, Jest, pytest, and PHPUnit. Implementation-wise we focused on the JUnit reports outputted by Gradle. While the reports from other tools are supported in theory, small deviations from the structured format may require customizations for specific test frameworks or build tools.

The XML reports include the name, location, failure message, stack trace, and run time of each executed test. Figure 3.3 shows an example of such a report.

The user is responsible for setting up their build tools to generate these test reports. A path to the reports can be specified as an argument to the upload script (see Section 3.1.1). Gradle outputs the test reports by default and places them in the `build` directory. Figure 3.2 shows the corresponding argument for the upload script.

[1] The bash script by Codecov is available open-source at `https://github.com/codecov/codecov-bash`

```
<?xml version="1.0" encoding="UTF-8" ?>
  <testsuites id="20140612_170519" name="New_configuration (14/06/12 17:05:19)" tests="225" failures="1262" time="0.001">
    <testsuite id="codereview.cobol.analysisProvider" name="COBOL Code Review" tests="45" failures="17" time="0.001">
      <testcase id="codereview.cobol.rules.ProgramIdRule" name="Use a program name that matches the source file name" time="0.001">
        <failure message="PROGRAM.cbl:2 Use a program name that matches the source file name" type="WARNING">
WARNING: Use a program name that matches the source file name
Category: COBOL Code Review – Naming Conventions
File: /project/PROGRAM.cbl
Line: 2
        </failure>
      </testcase>
    </testsuite>
</testsuites>
```

Figure 3.3: An example of a JUnit XML test report [30].

3.1.3 Collecting Coverage Per Test Reports

To enable the code under test and test health warning functionality, TESTAXIS can also collect coverage reports. Regular code coverage tooling, however, aggregates the covered lines of all tests of the test suite into a single report. This way, it is not possible to determine which test covered which lines. We need this information to highlight fragments that are both covered and changed in the code under test feature and to analyze potential flakiness for the test health warning feature. Some tooling does allow collecting coverage per test [57, 42] but does not integrate with build tools and cannot output XML reports. SonarQube used to have support for inspecting such coverage but has dropped this feature [25].

Therefore, we developed a custom Gradle plugin[2] that allows collecting coverage and generating reports per test using JaCoCo. It implements an approach similar to the technique proposed in [21]. The plugin registers a JUnit 5 extension with a before-each and after-each callback that gets run whenever a developer executes their tests[3]. The extension communicates with JaCoCo's `IAgent` interface. This agent exposes methods to set a custom session ID and to get the raw execution data. The before each callback sets the session ID to an identifier based on the method name of the test so that the results can be traced back to the right test. The after-each callback gets the current coverage execution data from the `IAgent` in raw bytes and writes it to an `.exec` file. JaCoCo typically uses this file type to output its full results, however, we create such a file per test. We now run the regular Gradle JaCoCo coverage report task to take the raw output and convert it into an XML report. Instead of doing this just once, our plugin re-runs this task for each test.

Similar to the test reports, developers can specify a path to the output location of the coverage reports as an argument to the upload script. TESTAXIS accepts JaCoCo coverage reports with session IDs in the format `fully.qualified.class.name##methodName`. The Gradle plugin outputs the reports in this format, however, a similar plugin outputting the same format could be created for other build tools as well.

[2]The Coverage Per Test Gradle plugin is available at `https://github.com/testaxis/coverage-per-test-gradle-plugin`

[3]The extension implements lifecycle callback for JUnit 5, see `https://junit.org/junit5/docs/snapshot/user-guide/index.html#extensions-lifecycle-callbacks-before-after-execution`

3.2 From the TESTAXIS Backend to the IDE Plugin 2

The next part in the information flow is communicating the build and test results from the TESTAXIS backend to the IDE plugin. The TESTAXIS backend is a Spring Boot application written in Kotlin. The application is responsible for processing incoming build reports, performing analyses on the results, and exposing the gathered information through an API.

3.2.1 Processing of Incoming Reports

TESTAXIS processes two types of incoming reports: test reports (see Section 3.1.2) and coverage reports (see Section 3.1.3). Once a test report is uploaded, TESTAXIS is made aware of the existence of a new build, even when the report is empty or absent. It creates a build in the database linked to a project and a user's account. For a new build, we persist the build status, the branch name, the commit hash, the repository name, the tag name, the PR number, the CI build service, and the external CI build ID and URL. TESTAXIS returns the internal build ID after the test report upload which can be used to upload coverage reports for the same build.

The incoming XML reports are parsed to an internal representation. For each test case execution, we persist the test name, the test class name, the run time, the passed status, the failure message, and the failure stack trace. This information is persisted in the database of the backend application and is linked to a user's account and one of their projects. If the CI build service indicates that the build has failed, the failure reason is either set to tests or to unknown when no reports are present or all tests passed.

The incoming coverage reports are large in size because they also contain information about non-covered lines of code and several statistics. Therefore, we store the coverage information in a more condensed format where we leave out all information that is irrelevant to TESTAXIS. Each incoming coverage report contains the coverage results for a single test. In that report, we collect all the covered lines per source file, ending up with a mapping from source file names to covered line numbers. Using the session ID in the coverage reports (see Section 3.1.3) we find the corresponding test case execution of the build in the database and append the coverage mapping.

3.2.2 Communication

The TESTAXIS backend communicates with the IDE plugin using two different techniques. It uses WebSockets to instantly push messages about finished builds for build notifications and a REST API for all other communication.

The REST API exposes all information gathered by the TESTAXIS backend[4]. This includes a listing of projects and their builds but also the information of all test case executions corresponding to a build. The API also features endpoints to retrieve the full details of a test case execution like the covered lines or the stack trace information. Furthermore, it offers the possibility to conduct a test health analysis on a given test case execution (see Section 3.2.3).

[4]The API documentation of TESTAXIS is available at `https://documenter.getpostman.com/view/14162304/TVzSjGs1`.

The API requires authentication using a JWT Bearer token. This token can either be retrieved by logging in using GitHub or creating a user account in TESTAXIS itself. Therefore, the API also features endpoints to register an account or log in. The project, builds, and test information can only be retrieved by users that have access to the project.

3.2.3 Backend-Side Test Health Analysis

We conduct the test health analysis partly in the backend application and partly in the IDE plugin. Since the backend application does not interact directly with the git repositories, we perform the analyses that need access to change information in the IDE. The backend application conducts the health checks for often-failing or slow tests. The TESTAXIS backend conducts the analyses upon explicit request of the client (see Section 3.2.2).

Slow Tests To determine whether a test is performing slower than average, we perform a run-time analysis. We first compute the average run time of a test of the current build. Then, we check if the test performs more than two times slower than the average run time. If this is the case, we raise the following warning: *"The performance of your test suite may be improved by speeding up this test. It performs slower than twice the average. The average test execution time is X ms."*

Often-Failing Tests We consider a test to fail often when it has failed in more than 10% of the last 50 builds. Contrary to CI providers, TESTAXIS has a more granular history on the level of tests instead of builds. This allows us to analyze whether a test fails often by investigating the history of a particular test case across multiple builds. It considers a test from a previous build the same when it has the same name and is in the same class. If we find that the number of times the test has failed exceeds the threshold, we raise the following warning: *"This test is failing often (X times in the last 50 builds). This may be an indication that your test is too tightly coupled to your production code or that the test may be flaky."*

3.3 From the TESTAXIS IDE Plugin to the Developer 3

The IntelliJ Platform IDE plugin is the last step in the information pipeline. From the plugin, TESTAXIS communicates the build information to the developer. Figure 3.4 shows an example of a failing test in the IDE plugin. Interaction with the plugin is likely to start upon receiving a build notification. Then, the developer can use the build and test overview to select a (failing) test to get access to the failure information, test code, and code under test. We show the interaction flow in Figure 3.5 and describe the implementation of the shown features below. However, we first describe how to set up TESTAXIS on the first usage.

3.3.1 Getting Started

Before the IDE plugin can be used, the user first needs to log in or register an account. This can be done in the settings screen of IntelliJ, see Figure 3.6 for an example. The user can either choose to log in through GitHub or to create an account using their email address.

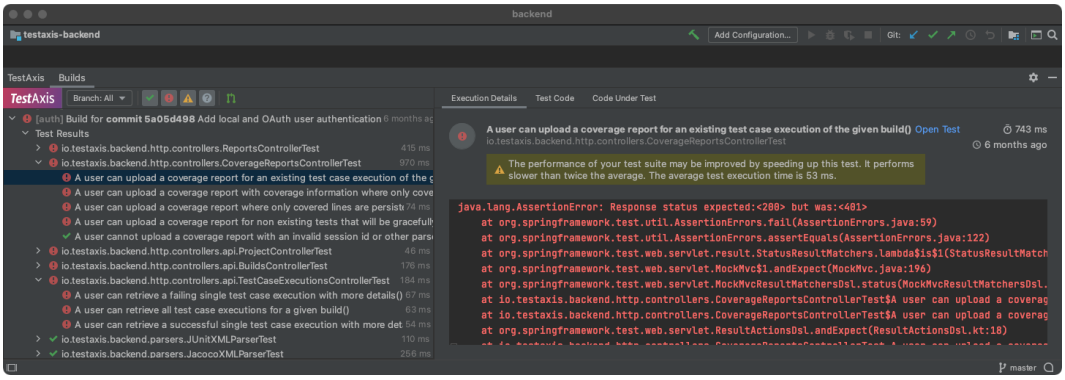


Figure 3.4: Example of a failing test in the TestAxis IDE plugin.

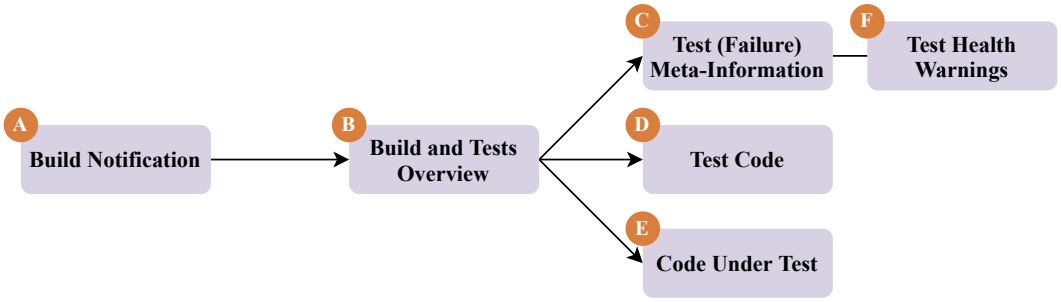


Figure 3.5: The expected user's flow of interactions with the TestAxis IDE plugin.

After the user has set up their CI build to upload results to TestAxis and the first build of a project has been completed, the project will show up in the settings screen. The user can select this project to link it to the currently active IntelliJ project.

To set up the upload script as a part of the CI build steps, an access token is required. After the user is logged in, TestAxis shows this access token in the settings screen.

Under "Advanced Settings", the user can set up the host of the TestAxis backend server. This way, a user can decide to run their own backend infrastructure and connect it to the IDE plugin.

3.3.2 Build Notifications A

The starting point of a user's journey through TestAxis is likely to be a build notification indicating that a new build has finished. By default, TestAxis notifies the user of both successful and failed builds but this behavior is configurable. From the build notification, see Figure 2.6 for an example, a user can click "Inspect Results" to open the TestAxis tool window and analyze the results of the CI build.

TestAxis shows a build notification as soon as it gets notified over the WebSocket connection (see Section 3.2.2). When such a message comes in, the plugin also updates the build and tests overview so that the build information is ready to be inspected by the user.

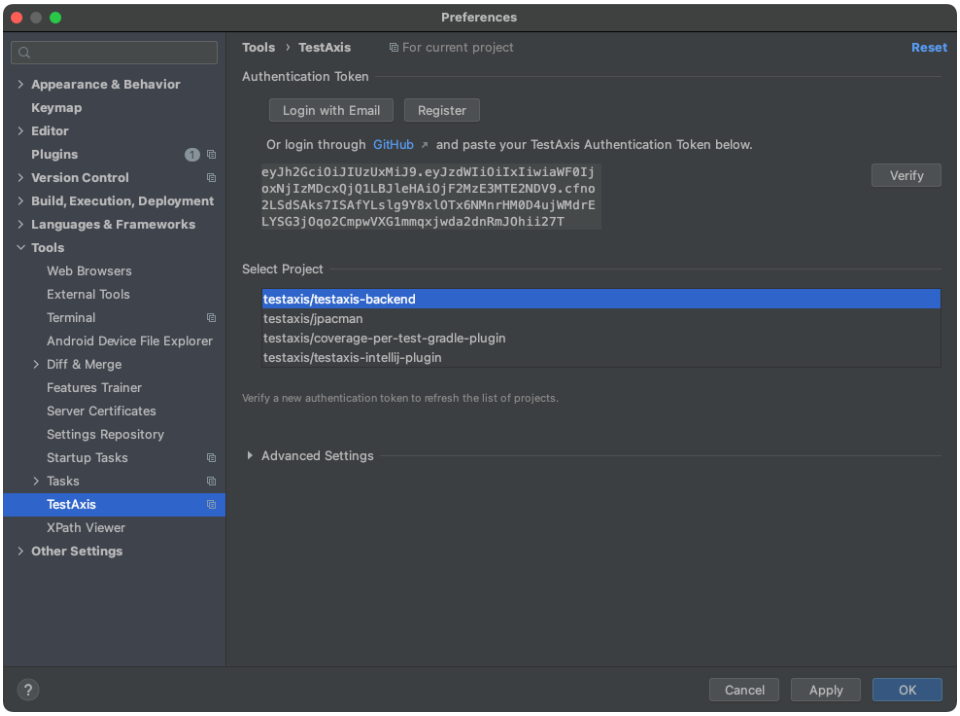


Figure 3.6: The settings screen of the TESTAXIS IDE plugin.

3.3.3 Build and Tests Overview B

On the left of the tool window (see Figure 2.1), TESTAXIS shows an overview of past builds. The information in this list is retrieved from the TESTAXIS backend through the REST API. When a user selects a build, the list expands to a tree and shows all the test classes of which tests are executed during the build. This information is loaded asynchronously to avoid retrieving lots of unnecessary information and therefore making it possible to show a larger history of builds. The class names can be expanded to show the tests per class. In this overview, we also show the run time and the pass/fail indicator. This looks similar to locally running multiple tests in IntelliJ.

To make the builds recognizable for the developers, TESTAXIS also shows the name of the commit that triggered the build. This information is not available when the test reports are uploaded to TESTAXIS and thus we must retrieve it at a later stage. We leverage the availability of the git plugin in IntelliJ which abstracts over native git command-line interactions. We communicate with this plugin to retrieve the name of the commits and, in a later stage, also to retrieve the build history for the (changed) code under test feature and the test health warnings.

Above the builds tree, the plugin shows filtering options (see Figure 2.1). It is possible to filter by branch, build status, and build type (PR builds). The filtering is performed at the client-side in the plugin.

3.3.4 Test (Failure) Meta Information C

Once the user selects a test in the builds overview, the tool window will show more information about the test case execution. The information is split in three tabs: *Execution Details*, *Test Code*, and *Code Under Test*. The first tab shows meta-information about the test failure, see Figure 2.2 for an example. This includes the name of the test, the name and package of the test class, the run time, the pass/fail status, and the execution date. It also shows a button to navigate to the test in the main IDE window. Clicking this button will open the test class at the start of the test method. In Section 3.3.5, we discuss how we find the method in the codebase based on the class and method name originating from the uploaded test reports.

Moreover, the meta-information also includes the stack trace. The stack trace originates from the test reports uploaded by the CI build runners. TESTAXIS displays the stack trace in an interactive manner similar to how IntelliJ presents the results of native test runs. This means that all entries in the stack trace are clickable. Clicking such an entry will navigate the user to the mentioned file at the right location. Also, TESTAXIS hides internal Java or platform calls to keep the focus on the relevant entries. In terms of implementation, we re-use the UI component used in native parts of IntelliJ to make this possible.

The *Execution Details* tab also shows the test health warnings. We discuss these warnings in more detail in Section 3.3.7.

3.3.5 Test Code D

The second tab with context of the test case execution is the *Test Code* tab. This one shows the code of the test that was executed. To allow putting the test in the right perspective, TESTAXIS shows the surrounding test class as well. It highlights the specific test method of the test that was executed and opens the file at the start of this method, see Figure 2.3.

To show this file, we first need to find it based on the package name and class name of the test method. From the uploaded test reports, we know the fully qualified class name and the name of the test. To find the file that implements this class, we make use of the Program Structure Interface (PSI) of IntelliJ. The PSI allows us to find a reference to the file in IntelliJ's internal file system by providing the fully qualified class name. Furthermore, it exposes program structure information allowing us to get a reference to a specific method, in our case the test method, as well. This allows us to identify the lines within the file that need to be highlighted or to navigate to the test method.

These interactions with the PSI are language-specific since the constructs of namespaces and modules differ per language. TESTAXIS offers support for Java and Kotlin projects. This is the only part of TESTAXIS where the support for programming languages is limited and thus where customization is needed when support for other languages would be added.

TESTAXIS shows the local version of the test file. This means, when the user is investigating a build of a different commit or branch than they are currently working on, the shown test code may be inaccurate. The plugin makes the user aware of this by showing a warning indicating that the currently checked-out revision does not match the build revision. The warning is shown at the top of the editor and offers the options to check out the right branch or commit by clicking a button in the warning message.

The shown editor is an extension of the `LanguageTextField` provided by the IntelliJ platform. In this editor, most IDE functionality (auto-completion, navigation to references, refactoring, etc.) is available. This should give developers a user experience that is familiar to them. The code shown can immediately be edited from our editor as well.

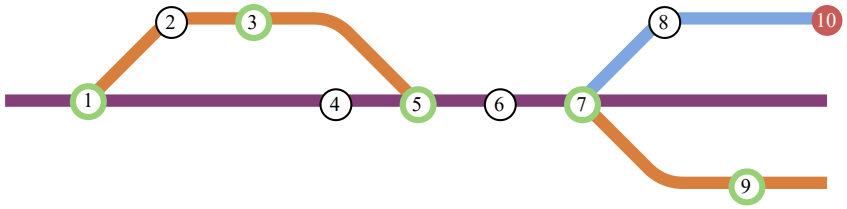
3.3.6 (Changed) Code Under Test E

The *Code Under Test* tab shows the files that were covered by the test, see Figure 2.5. Files that were changed in the commits leading up to the build run are labeled and shown first in the list. When a user selects a file from the list, they see the contents of the file on the right. The file is shown in the same type of editor as in the *Test Code* tab, see Section 3.3.5.

The lines of code in the file are highlighted using three different colors. Yellow indicates the lines that are covered by the test. The coverage information originates from the coverage reports uploaded to TESTAXIS during the CI build run. Green indicates the lines that were changed between the previous build and the current build. Lines that are both covered and changed are highlighted in purple. These are presumably the most relevant lines to the developer since it is likely that the issue causing a test to fail is in a part of the codebase that has changed and is tested by the current test.

Since we want to highlight all changes between the previous build and the current build, we first need to determine the predecessor of the current build. Establishing this predecessor relationship is not trivial. While the history of executed builds is linear, the history of commits triggering the builds is not (see Figure 3.7a). Thus, the previous build is not always the predecessor of the current build. Hence, we need to map the builds to the commit history and find the predecessor build in that mapping. In our implementation, we reverse this problem. From the perspective of a single commit, there is a linear history that led up to that commit. Instead of looking at which build belongs to which commit, we request git to provide this linear history. Using this timeline of commits, we start at the newest commit and go back in time and check for each commit whether it triggered a CI build. If it did, we have found the predecessor. Figure 3.7b shows the linear timeline based on the commit history in Figure 3.7a. In this example, we do not mark commit/build 9 as the predecessor since it is not in the history of commit 10, our starting commit. The first commit that has both triggered a CI build and is present in the history of commit 10, is commit 7. If the predecessor build passed and the current build failed, we know that it is likely that the changes between the two builds caused the issue which made the build fail.

Once we know the commit that triggered the previous build, we request the git plugin of the IntelliJ platform to provide all the commits between the current and the previous build. For all of these commits, we compute the code change diff resulting in a list of all added, modified, or deleted lines. We combine this list with the coverage information to determine which lines need to get highlighted with the color depending on whether the line is only covered, only changed, or both covered and changed. In the editor, we do not show deleted code fragments since they are not part of the file that is under test. However, TESTAXIS does indicate when code fragments have been deleted at the bottom of the editor. It also provides a button to open the full code change diff to compare all changes between the current revision of the file and the revision corresponding to the predecessor build.



(a) Example of a Commit History

(b) Example of a Linear History

Figure 3.7: Finding the predecessor build in the history of commits. The circles indicate commits. Green circles represent commits that triggered a CI build. The red circle is the commit with a build for which we are looking for its predecessor.

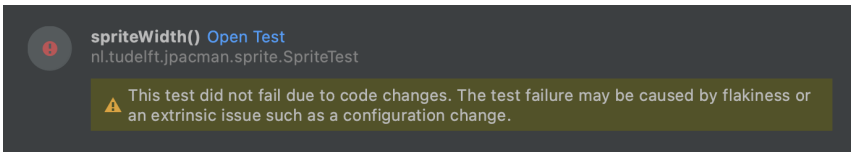


Figure 3.8: An example of a test health warning about a potentially flaky test.

Similar to the *Test Code* tab, a warning is shown when the user is inspecting a build of a different revision than the one that is checked out locally to ensure the state of the files matches the files during the build.

3.3.7 Client-Side Test Health Analysis F

In Section 3.2.3, we described the first part of the test health analyses that are conducted server-side. The flaky test analysis, however, is performed client-side in the IDE plugin. From the user's perspective, this does not result in any visual difference in how the tool shows the warnings. Figure 3.8 shows an example of how the flakiness warning is presented to the user. TESTAXIS shows these warnings on the *Execution Details* tab of the plugin.

Potentially Flaky Tests To determine whether a test is potentially flaky, we perform a code change and coverage analysis. Using the same information that we use for the code under test feature, we can detect potential flakiness. We consider a test to be potentially flaky when it does not cover any of the changes. In this case, there is either an extrinsic issue [51], such as a configuration issue, that causes the test to fail, or the test shows flakiness. We alert the user about the potential flakiness with the following warning: *"This test did not fail due to code changes. The test failure may be caused by flakiness or an extrinsic issue such as a configuration change."*

3.4 Summary

We developed a prototype of TESTAXIS. The prototype consists of two main parts: the backend application and the IntelliJ IDE Plugin. The CI build providers share the build results with TESTAXIS using a bash upload script that developers can include in their builds. The backend application then processes and analyzes these results. The backend shares the results with the IDE plugin using a REST API and a WebSockets connection for instant build updates. The plugin for the IntelliJ Platform shows all build and test information to the user. In the implementation, the IDE plugin re-uses components of IntelliJ to get easy access to code files or change information. The code under test feature highlights all changes between the previous and the current build. To establish the previous build, it uses git's ability to give a linear history of a single commit and finds the first commit in that history that has a CI build attached. The test health analysis gives interpretation to the collected information and is partly performed in the backend and partly in the IDE plugin.

Chapter 4

User Study Design

TESTAXIS attempts to improve the workflow of discovering and fixing failing builds. It introduces new functionality that has not been available in earlier tools by aggregating build and test information, by presenting new types of information such as the changed code under test, and by moving it to the context of the local development environment.

Because such a tool is new to developers, we do not know if the proposed features in fact help towards improving a developer's workflow and decreasing the time needed to fix a failing build. Therefore, we conduct a user study in which we ask developers to try out TESTAXIS and reflect on their experiences. The results of this user study should help us provide an answer to the research questions on performance and usefulness.

In the study, participants fill out a number of questionnaires and conduct assignments *with* and *without* TESTAXIS. We give an overview of the experiment in the first section. The experiment consists of two parts. The first part has a pre-experimental one-group pretest-posttest design. The participants conduct the second part of the experiment, which has a pre-experimental within-subjects design, in between the pretest and the posttest of the first part. We describe the design of both parts in the second and third section. Before we conducted the experiment, we first ran a pilot to evaluate the design of the experiment. Based on the pilot we made a number of improvements. We then carried out the experiment with 16 participants. In this chapter, we describe the design of the user study and explain how we conducted the experiments.

4.1 Overview of the Experimental Design

The goal of the experiment is to answer research questions RQ1-RQ6. In the experiment, we let a group of users try out and evaluate TESTAXIS to measure the potential performance improvement and perceived usefulness (see Section 4.1.1). On the one hand, we want to measure the quantitative performance improvement in time needed to solve failing tests using TESTAXIS. We collect this information in the pre-experimental within-subjects study. This information is useful to answer RQ1-RQ3.

RQ1

What is the influence of presenting a test failure in the IDE over a CI build log on the time a developer needs to fix a failing test?

RQ2

What is the influence of showing the test code on the time a developer needs to fix a failing test?

RQ3

What is the influence of showing the code under test, where the changed code is highlighted, on the time a developer needs to fix a failing test?

On the other hand, we want to capture qualitative usefulness feedback on the use of a CI build test result inspection tool like TestAxis. We collect this information in the pre-experimental one-group pretest-posttest study. This qualitative feedback provides insights to answer RQ4-RQ6.

RQ4

To what extent do developers prefer to be actively notified of CI build failures in the IDE over their current approach?

RQ5

To what extent do developers find it useful to be warned about the health and history of a failing test?

RQ6

To what extent do developers consider a CI build test result inspection IDE plugin useful?

Figure 4.1 shows an overview of the experiment. The first part of the experiment has a one-group pretest-posttest pre-experimental design (A), which we describe in detail in Section 4.2. It consists of two main phases: the pretest and the posttest phase. After the pretest established a baseline, TestAxis is introduced and the second part of the experiment, the pre-experimental within-subjects study (B) (see Section 4.3), is conducted. In this part the participants perform a number of programming assignments. Once that part is completed, we use the posttest to measure the improvement of TestAxis over the baseline from the pretest to evaluate the usefulness.

During the pretest, we ask participants to fill out a questionnaire (1) about their past experiences and their expectations of a tool like TestAxis (see Section 4.2.2 for an overview of the design of all questionnaires). The participants then conduct four assignments *without* TestAxis and four assignments *with* TestAxis (2). The ordering is mixed to avoid the influence of any learning effects on the results. We describe the assignment ordering in more

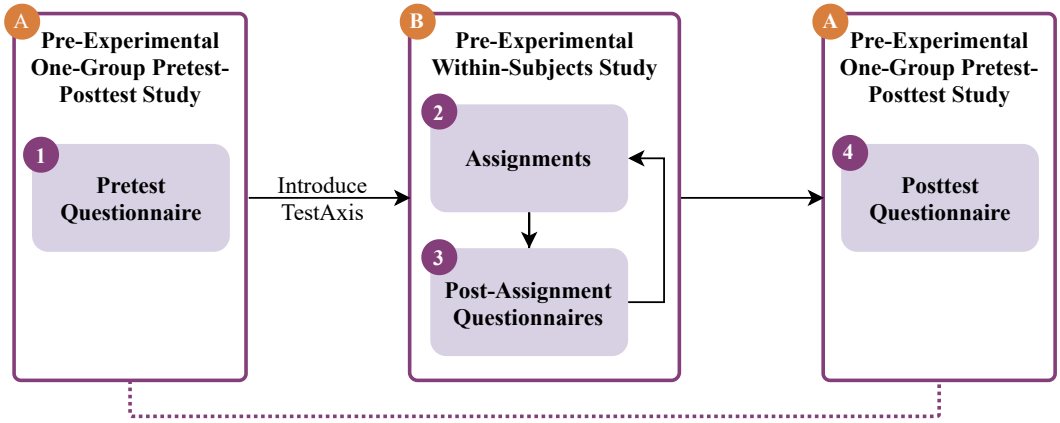


Figure 4.1: Overview of the relation between the two parts of the experiment.

detail in Section 4.3.4. After each assignment, the participants fill out a post-assignment questionnaire 3. At the end of the experiment, they share their feedback and experiences in the posttest questionnaire 4. The information gathered from the questionnaires provides the qualitative feedback that we use to answer RQ4-RQ6.

In total, the experiment has eight assignments divided over four categories (see Section 4.3.3). All assignments have two variants: *with* and *without* TESTAXIS. For each assignment, we present a CI build that failed due to failing tests. The participants have to find out which tests fail and why. Then, they have to come up with a fix. We time the performance of the participants on these assignments to gather the quantitative data to answer RQ1-RQ3. Per category, we divide the participants in two groups so that each participant conducts one assignment per category *without* and one assignment *with* TESTAXIS. We can then compare the results of the two variants of the first assignment (performed by different groups) and see whether it shows the same trend as the second assignment (to rule out sampling biases). During the assignments, we also take notes that may help explain the results or provide additional qualitative feedback over the questionnaires.

Before the participants start the assignments, they first watch two instruction videos. One shows the architecture and structure of the codebase of JPacman, which is the software project the participants use for the assignments (see Section 4.3.2). The other one shows and explains the functionality of TESTAXIS.

The experiment is approved by the Human Research Ethics Committee (HREC) of the Delft University of Technology and follows the guidelines set by the committee. At the start of the experiment, participants read and sign an informed consent form (approved by HREC, see Appendix B.1) indicating that they understand what data will be collected and how it will be used. As the experiment was carried out during the COVID-19 pandemic, all sessions with the participants were fully remote.

4.1.1 Measured Variables

Using the quantitative and the qualitative feedback by the participants of the experiment, we can measure the following dependent variables:

- **Performance** The time needed to fix a CI build that fails due to one or more failing tests.
- **Usefulness** The perceived usefulness and usability of aspects of a CI build test result inspection tool.

For the three main features of TESTAXIS (presentation of CI build results, test code, and code under test) the outcome of these variables provides a direct answer to the performance questions RQ1-RQ3 and a part of the answer to usefulness question RQ6. The usefulness variable also captures the perception of build notifications (RQ4), test health warnings (RQ5), and the overall usability (RQ6).

The independent variable of the experiment is the use of TESTAXIS. Therefore, we measure the dependent variables before and after the introduction of TESTAXIS.

4.2 Pre-Experimental One-Group Pretest-Posttest Study

The first part of the experiment is a one-group pretest-posttest pre-experimental study. This part of the study is mostly focused on evaluating the usefulness of TESTAXIS and its features. In this section, we give an overview of the design of the study and the questionnaires that capture the participants' opinions before and after using TESTAXIS.

4.2.1 Study Design

The experiment has a one-group pretest-posttest pre-experimental design [16]. This is an alternative to the classical controlled experiment where you compare the results of a control group to an experimental group. In our design, we only have a single group of participants, the experimental group. All participants are therefore exposed to the same elements of the experiment. The results of the control group are replaced by a so-called pretest. This pretest measures the dependent expectations on the usefulness variable, before introducing the independent variable, TESTAXIS, to the participants, see Figure 4.2. The introduction of the independent variable and execution of the assignments is called the intervention (this includes our second study, see Section 4.3). The posttest is conducted as the last step of the experiment after the intervention.

From the results, we may conclude that *there is* an effect after the intervention, however we cannot conclude that this effect is necessarily *caused by* the intervention [38]. If we can conclude that such an effect exists, we can provide the basis for and justify a more in-depth experiment. We choose a pre-experimental design due to a limitation of resources (in terms of time and staffing) that rule out the possibility for any larger scale experiment types such as a controlled experiment (requires a high number of participants) or a case study (requires a long period of time). Since the cost of these "true" experimental studies is significant, a more explorative study, like ours, can be useful to determine whether the cost of such a future experiment is justified.

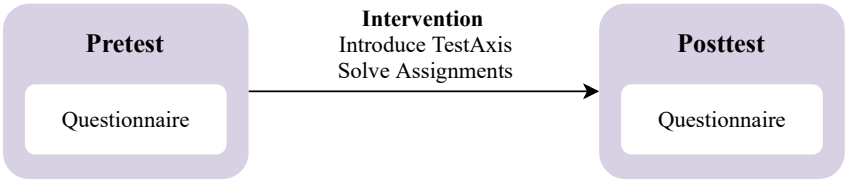


Figure 4.2: An initial pretest is followed by the introduction of TestAxis after which the posttest is performed.

In the pretest, participants fill out a questionnaire and conduct several assignments *without* TESTAXIS. This questionnaire has three parts: the first part asks about a participant's past experience, the second part asks about their views on software testing and continuous integration, and the third part asks about the participant's expectations of a CI build test result inspection tool.

The posttest also features a questionnaire. It asks about the participant's experience with using TESTAXIS and their perception of how the experiment went. Moreover, the third part of the pretest questionnaire is compared to the first part of the posttest questionnaire, which asks the same questions.

4.2.2 Questionnaire Design

While the dependent performance variable can be measured quantitatively by timing the assignments, the usefulness variable can only be answered using qualitative information. Therefore, we ask participants to fill out two questionnaires. The first questionnaire is at the start of the experiment, during the pretest. We use this questionnaire to ask the participant about their past experience, opinions on a number of matters, and establish a baseline in what they expect of CI build test result inspection tools. We compare the latter to the information we gain from the posttest questionnaire. Furthermore, the posttest questionnaire also asks about the experience with the tool and the experiment itself.

The questionnaires mostly contain statements that participants rate on a 1-5 Likert scale [33], where 1 means "strongly disagree" and 5 means "strongly agree". There are also a few open questions. Appendix B shows the full questionnaires.

Pretest Questionnaire

The pretest questionnaire contains questions on four different topics. The information gained from the topics about a participant's personal background and experience may be used to explain some of the results in the experiment. For example, a more experienced developer may find the assignments easier to solve, or may easier fall back to their own habits. To be able to provide insights into what a developer's normal test-fixing habits are, we also ask developers about their attitude towards software testing, continuous integration, and their current workflow. The last topic poses statements about what a participant expects from a CI build test result inspection tool so we can measure whether TESTAXIS meets a user's expectations.

- **Personal Background**
  Open and dropdown questions about a participant's education level, professional occupation, and years of programming experience.

- **Developer Experience**
  Statements about a participant's experience with developing software applications, software testing, and CI usage. Some of the statements are about specific tools or libraries used in the assignment to measure the impact of this specific knowledge on the results.

- **Attitude Towards Software Testing and Continuous Integration**
  Statements regarding a participant's software testing and CI behavior, and their view on their current workflow. The questions are about a participant's workflow when a build fails (open), how the participant usually becomes aware of a failing build, when and where they usually run tests, how they interpret CI build logs, and the attitude towards the concepts for which test health warnings are raised.

- **Expectations of a CI Build Test Result Inspection Tool**
  For the statements in this topic, the following high-level introduction of a CI build test result inspection tool was given:

  "A visualization tool for the results of tests executed during a CI build should improve the build fixing experience of the developer. The tool should show test failures in a more approachable manner than build logs.

  Furthermore, the tool should show information relevant to the test failure such as an interactive stack trace, the test code, and the modified code under test, in addition to information usually available in a build log.

  This should help fix failing tests quicker and make the fixing experience closer to a locally failing test. Moreover, the tool should actively notify developers of failing builds."

  The statements are about the participant's expectations of such a tool and whether they would consider using a tool like this in their development workflow.

Posttest Questionnaire

The posttest questionnaire is conducted after the participants have completed all assignments and have been introduced to TESTAXIS. This questionnaire contains questions in the following eight topics:

- **Verification**
  Statements about the meta aspects of solving the assignments. We ask the participants whether the assignments were challenging and interesting, whether external factors influenced their ability to fix the test, and whether the presented cases match issues they ran into in their own projects. Note that we also ask participants about the difficulty of individual assignments in the post-assignment questionnaire.

- **Usefulness of Informational Elements**
  Statements regarding the usefulness and the adequacy of the different context elements that TESTAXIS shows to the developer.

- **Build Notifications**
  Statements about the usefulness of the build notifications participants encountered during the experiment. We ask participants about how they perceived the TESTAXIS build notifications and whether they could replace their current solution

- **Test Health Warnings**
  Statements about the usefulness of the presented test health warnings, and whether participants would act on them.

- **TESTAXIS IDE Plugin User Experience**
  Statements about the user experience and integration within the IDE of TESTAXIS.

- **Expectations of a CI Build Test Result Inspection Tool**
  The same set of statements as in the pretest to measure whether TESTAXIS meets a user's expectations. The results can be compared to the results from the statements in the pretest.

- **General Comments**
  Open questions to leave general comments on TESTAXIS, potential missing features, and the experiment specifically.

- **Experiment**
  Statements regarding the quality and enjoyment of the experiment.

4.3 Pre-Experimental Within-Subjects Study

In the pre-experimental within-subjects study, we evaluate the quantitative dependent variable performance. We measure the performance in terms of failure-fixing time in seconds. All participants solve eight assignments for which we measure the time.

In this Section, we first explain the within-subject design of the study. Then, we describe the selection process of JPacman, the software project that the participants use in the assignments. Finally, we show the designed assignments and explain the per-participants ordering of the assignments.

4.3.1 Study Design

The study has a pre-experimental within-subjects design [46]. We conduct this experiment with the same group of participants as in the pre-experimental one-group pretest posttest study (see Section 4.2). Both studies are pre-experimental, which means that the performance results can only show whether the usage of TESTAXIS has any influence on the results

at all but not that the influence is necessarily caused by TESTAXIS. While the within-subjects design could also be employed in a true experimental setting, the limited number of participants in our study makes a pre-experimental study more appropriate.

In a within-subjects experiment, all participants are exposed to the same treatment which, in our case, means that all participants solve the same assignments [46]. The order in which they complete the assignments varies to counterbalance learning or order effects that could occur due to execution of previous assignments. We explain the assignment order in more detail in Section 4.3.4 The participants conduct four assignments *with* TESTAXIS and four assignments *without* TESTAXIS. These two sets of assignments are also mixed and balanced to avoid learning effects caused by working with the same software project.

The participants solve assignments *without* TESTAXIS to determine the baseline of the dependent performance variable. To compare the performance results of an assignment *without* TESTAXIS to an assignment with the use of TESTAXIS within the same group, there are two options. The first option would be to split the group in two and ask one half to solve the *without* assignment and one half to solve the *with* assignment. This is very much like a controlled experiment, and, given the small sample size, would be influenced too much by the experience of individual participants. The second option is to have participants solve both the *with* and *without* assignment, however, this requires two very similar assignments. At the same time, the assignments can also not be *too* similar because the results could then be influenced by a learning effect. It is not feasible to design assignments that meet these requirements.

Therefore, we opt for a design that is a mix of these two approaches. We design two assignments, say *A* and *B*, for each of the four assignment categories (see Section 4.3.3). Per category, we split the group in two. The first group conducts assignment *A without* TESTAXIS and assignment *B with* TESTAXIS, and the second group conducts assignment *A with* TESTAXIS and assignment *B without* TESTAXIS. Per assignment, we can now compare the performance of the first group against the second group. When we see the same effect for both assignments, we know that the composition of the group did not influence the outcome.

After each assignment, we ask the participants to fill out a questionnaire regarding their behavior while performing the assignment and their reflection on the assignment itself (see Appendix B.2). The statements in this questionnaire ask the participant whether the assignment was too difficult and on which tasks (for example, finding out which tests failed or looking at the test code) they spent the most time. The participants can also share any general remarks to offer a possibility for direct feedback. For the assignments *without* TESTAXIS this questionnaire measures the “experience of fixing failing test without the tool” and for the assignments *with* TESTAXIS it measures the “experience of fixing failing test with the tool”.

4.3.2 Software Project Selection

The assignments of the experiment ask participants to fix failing test cases that are designed to mimic test failures that occur while working on real software projects. Obviously, the simulation of realistic test failures requires that the designed test cases are part of a software project that is sufficiently complex and close to the real-life projects the participants have

experience with. We picked JPacman, a simple Pacman-style game implemented in Java that is written for software testing education at the Delft University of Technology.

We used the following criteria to select the software project:

- **Easy to Understand Within a Short Time** Since many of the participants will have no or little prior experience with the software project, they must be able to grasp the project within a short time. The understanding of the software project may influence the results when the participants spend more time understanding the general structure of the codebase of the project than working on solving the assignment. Therefore, it must be possible to explain the project in a short time and participants must quickly feel confident working in the software project.

- **Non-Trivial Codebase** While the project must be easy to understand, it must also not have a trivial implementation. A project that is too simple may not drive developers to show their usual development behavior. It also does not allow for designing suitable test failures for the assignments. This could result in different behavior while using TESTAXIS than what a developer would do outside the experiment in their own projects.

- **Only Common Language Features and Dependencies** To enable a quick understanding of the project and to support quick familiarization, the chosen project must be a Java project that only uses common language features. It also should not use any libraries or other types of dependencies that are uncommon or influence the way code is written. Obviously, the project does need to use at least a testing framework such as JUnit.

- **Open-source Software** To enable reproducibility of the experiment, we require the software to be an open-source project.

- **High Variety of Tests** The software project must have tests of different types. To evaluate the different features of TESTAXIS, we need a project that has unit tests, integration tests, and system/end-to-end tests.

- **Realistic Test Cases** To ensure the generalizability of the results, the failing test cases in the assignments must be realistic. This requires the test cases in the chosen software project to be realistic and of a quality level comparable to an industry project.

JPacman fits these requirements. The project only uses basic language features and apart from static analysis dependencies only makes use of JUnit, AssertJ, Mockito, and list helpers from Google Guava. It features a variety of tests of different types with high code coverage. The project is available open-source on GitHub[1]. We created a fork of the project where we made the project compatible with newer Java versions and where we made modifications to the project to support more interesting cases for the experiment. Our fork is also available on GitHub[2].

[1] JPacman is available at `https://github.com/SERG-Delft/jpacman`

[2] Our JPacman fork is available at `https://github.com/testaxis/jpacman`

4.3.3 Assignment Design

We designed eight assignments in four categories. These categories cover the three main features of TestAxis (build result, test code, and code under test inspection) and enable collecting feedback about build notifications and test health warnings as a side-effect. Each category has two assignments. The assignments ask participants to fix the test that failed in a given CI build. The categories are separated by the location where the issue causing a test to fail can be spotted most easily.

All eight assignments have two variants: one *without* TestAxis and one *with*. For both variants, the participants first read the description of the change. This description is written in a style that reflects typical pull request (PR) descriptions. The goal of the descriptions is to convey the intent of the code change. This is important to rule out the trivial solution of fixing a failing test: changing the assertions to match the state of application just before the test fails. For the without-variants, the participants cannot use TestAxis but are given access to a GitHub PR introducing the change. This PR also provides access to the GitHub Actions CI build log and gives an overview of the changed files. In the with-variants, we give the participants access to TestAxis to solve the assignment. For these assignments, the participants do not have access to GitHub or the CI build log. In both cases, the participants may use any (other) feature of IntelliJ and any other tool that they would normally use in their workflow.

We designed the assignments with the intent to be similar to issues that may occur in real-life projects. We verify this by asking participants if the cases were indeed similar to the ones they encountered in their projects in the posttest questionnaire. We describe the assignments per category below. Appendix C.2 lists the descriptions of the changes that the participants read before starting the assignment. We list the categories and the corresponding assignments below.

Category 1: Test Failure Metadata

For the assignments in this category, the reason for the failure can be spotted from the test failure metadata (the name of the test and the stack trace).

Assignment 1a — **TIME LIMIT** 5:00

In this assignment, a **test resource file is renamed** "for consistency" while the references to the file in the tests are not updated. This causes multiple tests to fail. The participant can spot this issue from the stack trace of the test failures. By **clicking** the **entries in the stack trace**, the participant can find the location of the reference to the file and update the filename.

`https://github.com/testaxis/jpacman/pull/3`

Assignment 1b TIME LIMIT 5:00

The maps of the levels of JPacman are described in text files that the game parses. Each element on the map is represented by a different character. In this assignment, a **new map element** with a corresponding character is added to the game. However, while this new element is placed in the map on several locations, the new **parser** code contains a small **mistake** causing it to throw an **exception**. The exception indicates that the map character is not recognized. This is because the author of the change accidentally wrote code that recognizes a different character. The participants can spot this issue in the **stack traces** of multiple failing tests or by looking at the changed files. The **first entry** in the stack trace refers to the **parser method** with the mistake.

`https://github.com/testaxis/jpacman/pull/4`

Category 2: Test Code

For the assignments in this category, the reason for the failure can be spotted in the test code.

Assignment 2a TIME LIMIT 5:00

The changes for this assignment add a **new method with tests** to find an element on the map by a relative position to another element. The added **production code** is relatively **complex** and it takes a lot of time to fully understand it. The **issue**, however, is **in the test code**. In the test, a small board is created with three elements. "By accident", the **same element is added twice**, simulating a copy/paste error, and the map thus has only two unique elements. The test code then tries to fetch an element relative to the third element that is not present on the map, **resulting in a** `NullPointerException` somewhere in the production code. The participants can spot and fix this issue by careful inspection of the **test code**.

`https://github.com/testaxis/jpacman/pull/5`

Assignment 2b TIME LIMIT 5:00

In this assignment, a **method** to retrieve the winning player is added together **with** a number of **tests**. These tests involve mocking and have a relatively **complicated setup**. The issue is in the test code where one of the **stubbed methods** indicating the *alive status* of a player returns `false` instead of `true`. The participants can spot and fix this issue by careful inspection of **the test code**.

`https://github.com/testaxis/jpacman/pull/6`

Category 3: Code Under Test

For the assignments in this category, the reason for the failure can be spotted in the code under test.

Assignment 3a TIME LIMIT 5:00

A pacman game typically contains little dots a player can collect to earn points, called pellets. This assignment adds a **new type of pellet with a random score**. The changes include modifications of the parser, a **new overloaded method** to create this pellet type, and a number of tests. The overloaded method is a copy of the original method and while it accepts a **parameter for the amount of points**, it is **never used.** The participant can spot this issue by inspecting the **code under test**.
`https://github.com/testaxis/jpacman/pull/7`

Assignment 3b TIME LIMIT 5:00

To refactor existing code, this assignment adds a **new method** to get an **opposite cardinal direction**. In the implementation, the method returns the **wrong direction** for the south and west directions. The participant can spot this issue by inspecting the **code under test**.
`https://github.com/testaxis/jpacman/pull/8`

For both assignments in this category, the assignment variant *without* TestAxis requires a manual analysis of the changed files that may be relevant. This is a tedious task because the changes also include code additions and modifications that are not related to why the test is failing. In TestAxis the issue is part of the set of changed code under test fragments making it easier to spot the issue.

Category 4: Advanced Code Under Test

For the assignments in this category, the reason for the failure can also be spotted in the code under test. However, these assignments are more advanced.

Assignment 4a TIME LIMIT 10:00

The participant's task in these assignments is different than in the previous assignments. In this assignment, a number of **power-ups are added** to the game throughout the whole codebase. The power-ups are active within certain score or steps ranges. Because of the new game behavior, the high-level **smoke test fails**. The task is to find out **which of the three added power-ups** causes this test to fail. Once a participant has found the right power-up, they need to **change the range** in which the power-up is active to make sure it is not triggered during the failing smoke test. All power-ups are implemented as an if-statement. With access to TestAxis the issue can be spotted by looking at the **changed code under test** and checking which of the bodies of the if-statements are covered.
`https://github.com/testaxis/jpacman/pull/9`

Assignment 4b TIME LIMIT 10:00

This assignment is the **same as assignment 4a** but has a different set of **newly added power-ups**.
`https://github.com/testaxis/jpacman/pull/10`

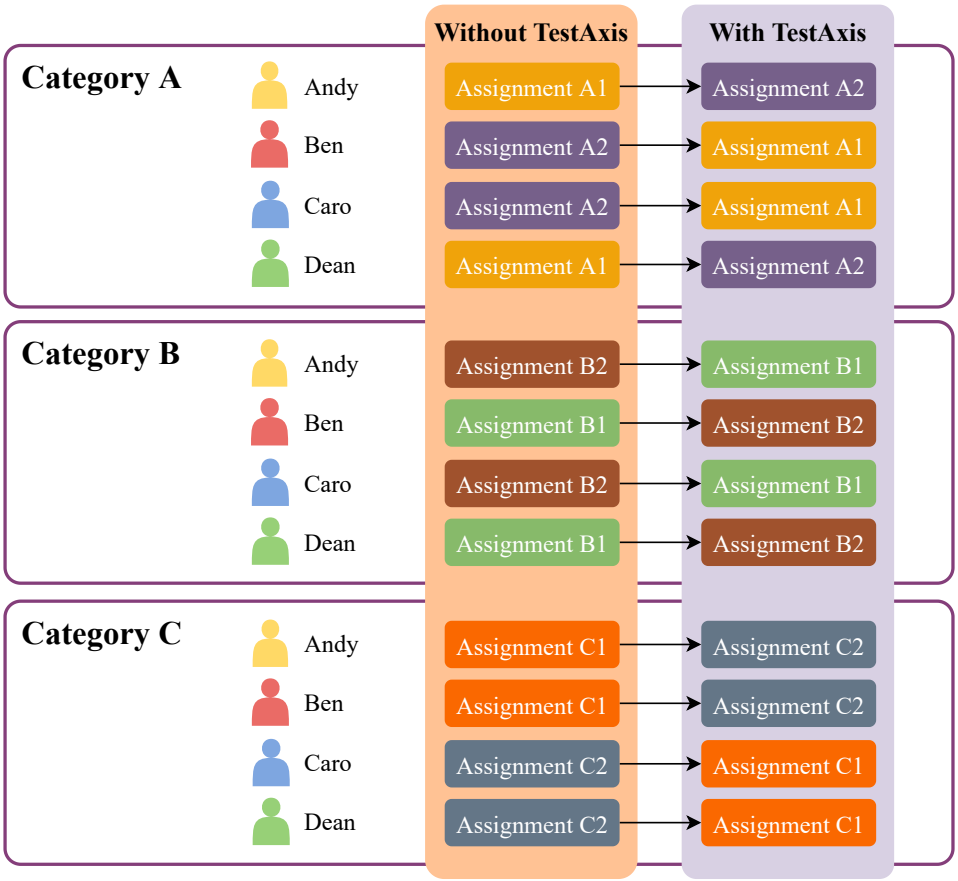


Figure 4.3: An example of how assignment variants are distributed for an experiment with three categories and four participants.

4.3.4 Assignment Ordering

The order in which the participants perform the assignments is an important factor when trying to minimize the learning effect on the results. If the assignments are not distributed carefully, a situation may occur in which the participant first does all the assignments *without* TESTAXIS and then all assignment variants *with* TESTAXIS, for example. In such a situation the participant would have learned about the codebase during the first round and could therefore be faster while solving the assignments *with* TESTAXIS. We would then not be able to conclude that the introduction of TESTAXIS caused the improvement. We try to mitigate the learning effect by varying the assignment order in a way that effects caused by the order are counterbalanced [46].

First, we remove the strict separation between assignments *without* TESTAXIS, which serve as the baseline, and the assignments *with* TESTAXIS. This allows us to create an ordering of assignments in which the assignments *with* TESTAXIS are interleaved by the assignments *without* TESTAXIS.

Second, as described in Section 4.3, for each category the participant solves two different

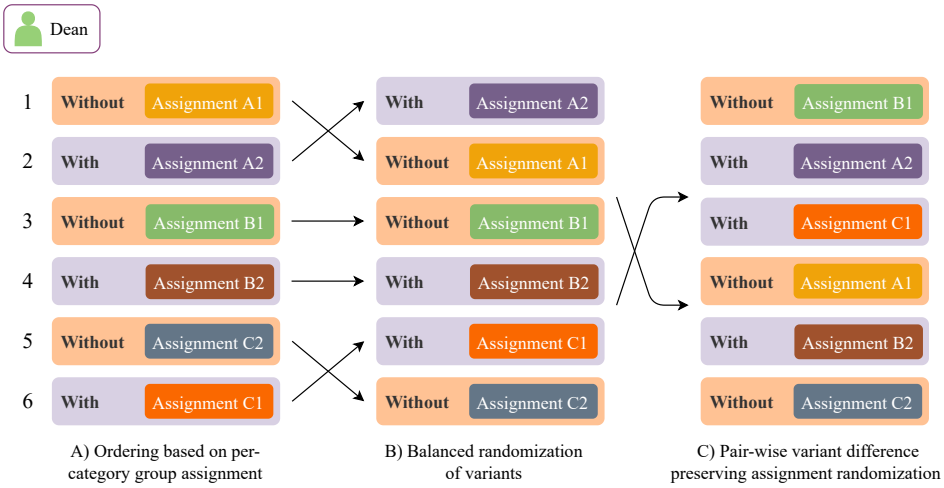


Figure 4.4: The three steps that lead to the final ordering of assignments for a single participant.

assignments. In general, each category has two assignments, and each assignment has two variants. Thus there are four assignment variants per category. A participant only solves one variant of the assignments. For each category, the participants are randomly distributed into two groups of equal size. The first group performs the first assignment *without* TestAxis and the second assignment *with*. The second group performs the first assignment *with* TestAxis and the second assignment *without*. Figure 4.3 shows an example of how the group distribution per category works.

At an individual level, we now have the starting point of the ordering of assignments. Figure 4.4a shows an example for the ordering of assignments for Dean, based on Figure 4.3.

Third, we randomize the order of the *with*/*without*-variant per category, see Figure 4.4b. This reduces any advantages and learning effects of always having performed a somewhat similar assignment *without* TestAxis before an assignment *with* TestAxis.

Finally, we shuffle the whole ordering while preserving the difference in variants per two assignments to remain the previous mitigation of the learning effect. To further reduce this effect, this step re-orders the assignments to reduce the effect of having seen a similar test failure in the previous assignment. The randomization favors assignments from categories that are not equal to the category of the previous assignment.

The ordering of assignments is pre-generated before the experiment to ensure the assignment is fair and balanced[3]. Appendix C.1 lists the assignment order per participant.

4.4 Selection of Participants

To conduct the experiment and gain useful insights about the results, we needed to recruit a large enough number of participants. Although the assignments are not extremely difficult,

[3]The script to generate the ordering is available in the replication package [12].

we required participants to at least have experience with Java and CI. This ensures a somewhat equal baseline and the ability for participants to reflect on and compare with their CI build fixing workflows.

At the same time, we also wanted a diverse group of participants and therefore used a phased participant recruitment process with different target audiences per step. We first reached out to acquaintances, which are mostly (PhD) students. Then, we placed a message on the internal messaging platform of Computer Science teaching assistants of the Delft University of Technology, with a similar target audience. To target industry developers, we posted a collection of tweets on Twitter illustrating the capabilities of TESTAXIS and asking for a developer's help to improve the project. Finally, we also posted on LinkedIn with the same target audience in mind.

In the promotional posts and messages, potential participants were directed to a website with detailed information about the experiment and the ability to sign up. Appendix D shows the information on this website. To thank the participants for their time and to increase engagement, we raffled four 15 euro gift cards among the participants. In total, 16 participants signed up for the experiment. Before the experiments, we also conducted a pilot with 1 participant.

4.5 Pilot

To ensure a smooth run of the experiments, we first conducted a pilot. This pilot was set up in the same way as we planned the main experiments. The only difference in the setup was that we did not apply any time limits to the assignments yet to determine how much time is reasonable for each assignment. Based on the feedback provided by the participant and our own observations, we made the following changes:

- **Assignment Order** In the pilot, the assignments were ordered chronologically per category. Also, the *without* variant always came before the *with* variant. This ordering benefited the results for the *with* variants because the assignments within a category are somewhat similar, and the *without* variant always came first. Therefore, we improved the assignment order to be less predictable, as described in Section 4.3.4.

- **Technical Experiment Setup** The pilot took 136 minutes in total while we estimated 90 minutes. While this difference is partly caused by the lack of time limits, the switches between assignments also took longer than expected. Thus, we automated these switches to reduce the amount of time needed for the experiment and to smoothen the experience from a participant's perspective, see Section 4.6.2.

- **Introduction to TESTAXIS and JPacman** Before the participant starts with the assignments, we introduce them to TESTAXIS and JPacman by giving an overview of the features of TESTAXIS and the code structure of JPacman. We observed that repeating these explanations for each experiment would take a lot of time of the experiment and create inconsistencies in the base knowledge that is provided. Therefore, we created two explanation videos in which we can keep the information concise.

- **Code Under Test Assignments** It was too easy to quickly spot the issue in the code under test assignments, both in TestAxis and on GitHub. This was because the failure-introducing change touched very few lines of code. We added more (related) changes to the PRs belonging to these assignments to make the PR more realistic.
- **Minor Tweaks** We made minor tweaks in several parts of the experiment based on the pilot. In the questionnaires, we fixed accidental double questions and added a question about Gradle experience, as suggested by the participant. In the assignment descriptions, we fixed a few small mistakes. Finally, we fixed one of the build logs by rerunning the build because the log showed an unrelated TestAxis authentication error.

4.6 Experiment Execution

We conducted the experiment in March 2021, for three weeks in multiple sessions per day. Due to the COVID-19 pandemic at the time, the experiment was fully remote. All sessions were individual and guided by an observer. During a session, the observer took notes of interesting things that happened or were said during the experiment. The observer also timed the assignments. A session took about 90 minutes depending on the time needed to fill out the questionnaires or solve the assignments. Participants could request a break whenever they wanted.

The execution of the experiments went well and without any major issues. Midway through the experiments, we updated the assignment description of the assignments of category 4. The description mentioned that certain thresholds needed to be *increased* while it should have said *changed*.

4.6.1 Structure of a Session

Figure 4.5 depicts the flow of the experiment. All experiment sessions were performed according to the following script:

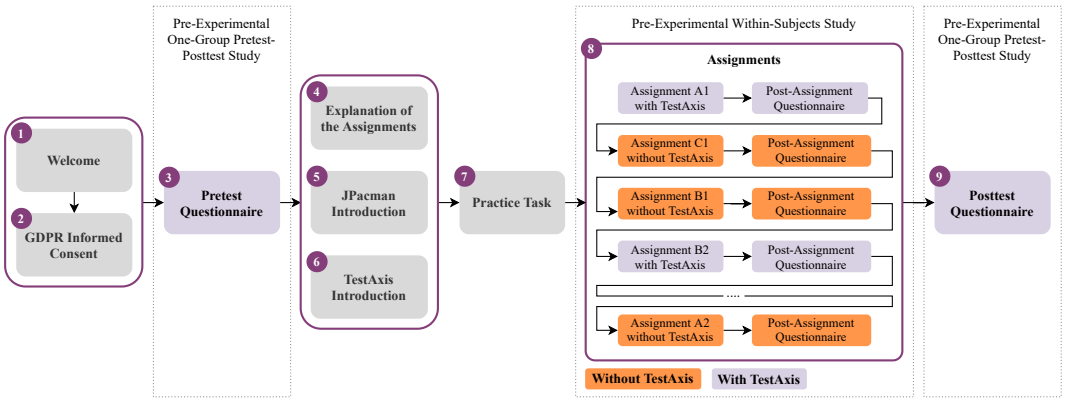


Figure 4.5: The structure of an experiment session.

1. **Welcome** Explanation of the general structure of the experiment and optimizing the technical setup (for example, screen resolution or key bindings).

2. **GDPR Informed Consent Form** Explanation of the privacy aspect of the collected research data and filling out and signing the informed consent form, see Appendix B.1

3. **Pretest Questionnaire** Filling out the pretest questionnaire. The questionnaire was presented on the experiment machine the participant had remote access to.

4. **Explanation of the Assignments** Overview of the goal and rules of the assignments. We gave participants the following information:

   You will get 8 assignments in total. In these assignments you have to fix failing CI builds that fail due to a failing test. You will complete these assignments with or without TestAxis, which is the tool we developed. We will measure the time you use to find the issue of the failing build. The issues could be in the code under test as well as the test code itself. The issue is always a test failure, never compilation or build tool issues. After each assignment, we ask you a few questions about your experience.

   The difficulty varies, some assignments will take more time than others. Some cases can be spotted quickly, others need more work. The order of the assignments is random, so the difficulty may jump up and down. If you have not found the solution after a certain amount of time which varies per assignment, we may ask you to stop and move on to the next assignment.

   Your task is to find out why the tests are failing. Once you have found the issue, try to come up with a fix. The fixes should never take more than a few lines of code. When you have applied the fix, please say this out loud. For two assignments, the task is a bit different but this will be explained once we are there.

   We ask you to think out loud while performing the assignment. When you can use GitHub, you are not allowed to use TESTAXIS*. When you are allowed to use* TESTAXIS*, you cannot use GitHub. Other than that, you are free to use any tool you want to solve the failing test.*

5. **JPacman Introduction** An introduction video of JPacman[4] explaining the highlights of the architecture and structure of the project.

6. **TESTAXIS Introduction** A introduction video of TESTAXIS[5] explaining the main features.

7. **Practice Task** A small task in which participants have to find the `Launcher` class of the software project and start the game. This task is included to get participants more comfortable with the remote setup and ensure that everything is working correctly.

[4]The JPacman introduction video is available at `https://youtu.be/Pzv1d2AzpJM`
[5]The TESTAXIS introduction video is available at `https://youtu.be/2y3RzwfCnRQ`

8. **Assignments** Execution of the assignments. For each assignment, the participants first read the assignment description. Afterward, they either switch to the browser to review the PR on GitHub or to the IDE to see the TestAxis output. Participants find the GitHub PR by clicking the link in the assignment description. In assignments *with* TestAxis, they receive a build notification in the IDE which they can click to open TestAxis. Between each assignment, the development environment is reset to a default state and prepared for the next assignment. This includes, for example, undoing all code changes, closing all tabs and tool windows, and checking out the right branch. After each assignment, we ask the participant to fill out the post-assignment questionnaire. If the participant was not able to solve the assignment, we tell them the issue and the fix after filling out this questionnaire.

9. **Posttest Questionnaire** Filling out the posttest questionnaire once all assignments have been completed. The questionnaire was presented on the experiment machine the participant had remote access to.

4.6.2 Technical Experiment Setup

The experiments were conducted through Zoom. The participants had remote control of the experiment machine and were not required to have anything installed other than Zoom. The experiment machine ran macOS and since Zoom does not convert keybinding, the participants had to be careful when using a keyboard without the macOS keyboard layout.

To support smooth transitions between the different elements of the script and between assignments, we created an application that could run actions on the experiment machine

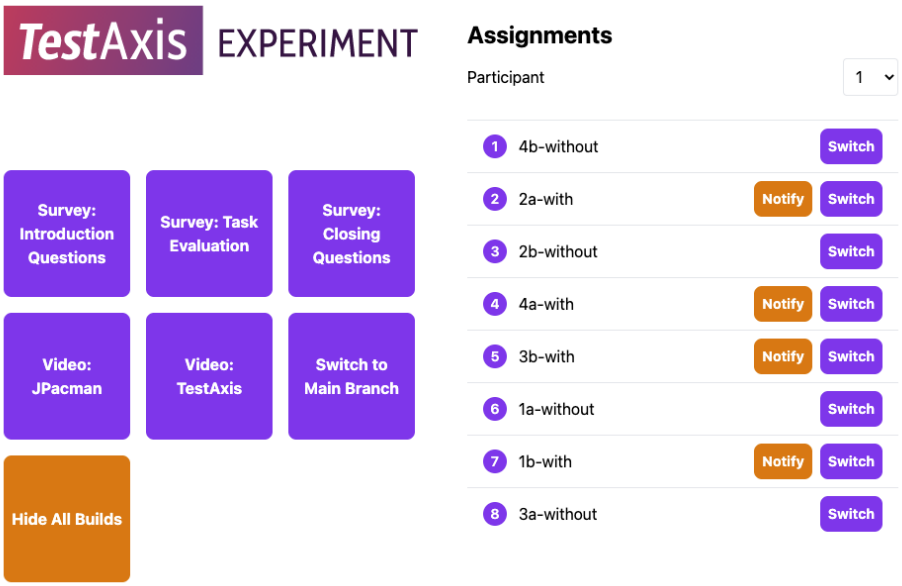


Figure 4.6: The support tool assisting the observer during the experiments.

remotely. This way we could show the explanation videos or one of the questionnaires with a single button press from another computer. The application also supported assignment switching which would reset the state of the IDE by closing any windows or tabs, undoing code changes, and checking out the right branch. It also shows the assignment description PDFs to the participant. Furthermore, it offers the ability to fire TESTAXIS CI build notifications for predefined builds corresponding to the assignments. The application also showed the pre-generated assignment order for the participants. Figure 4.6 shows the client of the support application. The server controlled the experiment machine through AppleScripts, git interaction, and TESTAXIS API communication.

The participants had access to IntelliJ Community Edition with TESTAXIS installed, Google Chrome, and any application that is available in a standard macOS installation.

4.7 Summary

We presented the design of the experiment. The experiment consists of a one-group pretest-posttest study and a within-subjects study which we use to measure the performance improvement and usefulness. The pretest and posttest of the first part contain questionnaires. In between, TESTAXIS is introduced and the participants perform a number of programming assignments. The study has a pre-experimental design that can be used to determine whether the introduction of TESTAXIS has any effect at all and may justify the need for more in-depth experiments.

The experiment features eight different programming assignments in four categories: test failure metadata, test code, code under test, and advanced code under test. Each assignment has two variants: in one the use of TESTAXIS is not allowed (pretest) and in the other, it is allowed (posttest). For each category, the group of participants is divided into two. Each participant performs one *without* variant and one *with* variant per category. The assignments are ordered randomly per participant.

The experiment takes 90 minutes per participant and is conducted remotely. The participants were recruited through communication with personal acquaintances, internal university communication channels, and social media. The experiment design and setup were improved after a pilot was conducted.

Chapter 5

Results and Analysis

To answer the research questions posed in our study, we conducted the experiment as described in the previous chapter. In this chapter, we present, analyze and discuss the results. First, we provide an overview of the results from the time measurements of the assignments and the questionnaires. Then, we analyze and discuss the failure-fixing time performance results of the assignments. We also discuss the usefulness results, based on the questionnaire feedback. In both discussions, we formulate an answer to the six research questions based on the findings in our study. To put the results in the right context with the constraints of the study, we also discuss threats to the validity of the results.

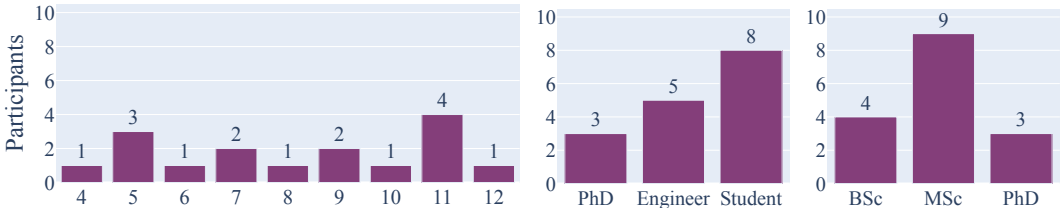
5.1 Results

We present the questionnaires and assignment performance results in six categories. Based on the pretest questionnaire, we show the demography, experience, and background of the participants in Section 5.1.1. In Section 5.1.2, we use the performance results and the post-assignment questionnaires to present the results of the execution of the assignments. We show the participants' opinions on the tool itself and its usefulness in Section 5.1.3. We also asked participants to reflect on the build notifications and test health warnings, which we present in Sections 5.1.4 and 5.1.5. Finally, we present the participants' perception of their behavior during the assignments and of the quality of the experiment in Section 5.1.6. Appendix E shows the full results of the experiment.

5.1.1 Participants

The following results are part of the outcomes of the one-group pretest-posttest experiment.

The pretest questionnaire asked about a participant's personal background, their experience with software testing and continuous integration, and their opinions on several statements about software testing. Appendix E.1 shows the full results of the pretest questionnaire. In this section, we describe the group of participants using the information gathered in the pretest questionnaire. In total, 16 people participated in the experiment.



(a) Years of Programming Experience (b) Occupation (c) Education Level

Figure 5.1: Demography of the participants of the experiment.

Demography

The participants of the experiment have at least four years of experience in programming. The most experienced participants have programmed for 12 years. The other participants are relatively equally distributed between 4 and 12 years, see Figure 5.1a. 31.3% of the participants work in industry as a software engineer, while the main occupation of the remaining participants is student or PhD student. Figure 5.1b shows an overview of the professional occupations of the participants. All participants have an academic background. The current or highest education level of most participants is MSc (56.3%) as shown in Figure 5.1c. The other participants were either BSc (25%) or PhD students (18.8%) at the time of the experiment.

Experience

Figure 5.2 shows the experience in software development of the participants. The figure shows the questions from the pretest questionnaire, the average score, and the distribution over the Likert scale. We visualize the Likert-scale results with *diverging stacked bar charts* [49]. In these charts, we plot the frequency as a percentage of the ratings 1 and 2 (strongly disagree and disagree) left of the center, rating 3 (neutral) on the center, and ratings 4 and 5 (agree and strongly agree) right of the center. The 16 participants consider themselves to be experienced software developers (●4.0[1]). They are experienced with developing Java applications (●3.8) in IntelliJ (●3.9). One participant indicates that they are not experienced developing software in Java. Some of the participants have experience developing software applications professionally (●3.4), whereas others do not have any experience in this area.

While the participants are experienced with software development, they are less experienced with software testing (●3.6, see Figure 5.3). We observe that some of the participants indicated to be very experienced in testing, while others indicated to not be experienced at all. We see similar results for specific elements of software testing. The participants rate themselves as averagely experienced with using mocks (●3.4). We see a comparable distribution for the experience with inspecting code coverage reports after running tests with coverage collection enabled (●3.6).

[1]To present the Likert-scale results we show the average score indicated by the purple-colored dot. The small bar chart gives a rough indication of the distribution of the answers that the participants gave.

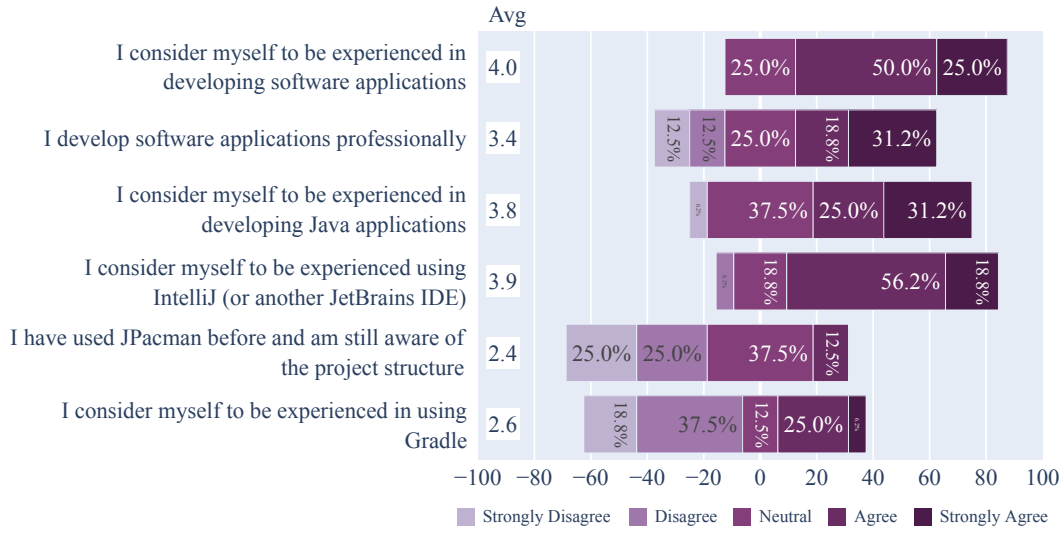


Figure 5.2: Participants' experience with software development.

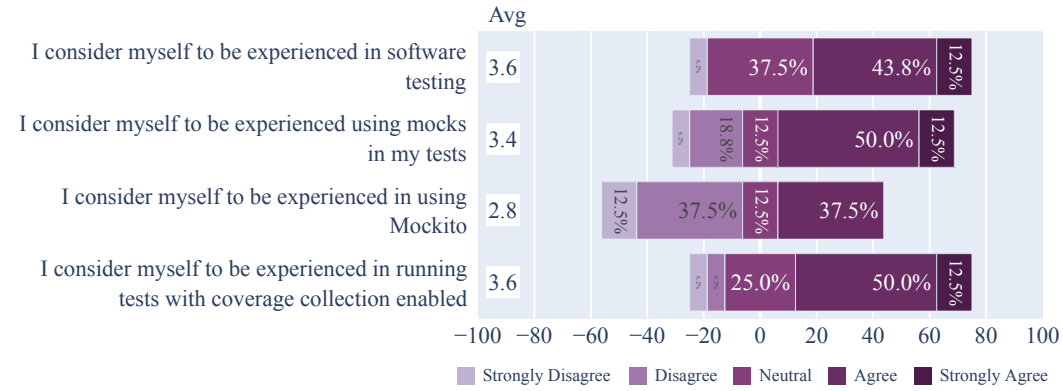


Figure 5.3: Participants' experience with software testing.

The results on the experience with CI show a different trend, see Figure 5.4. The participants rate themselves as highly experienced in using CI build tools (like Travis CI, GitHub Actions, or Jenkins; ●4.2) and inspecting the output logs when a build fails (●3.9). They also indicate to have regularly worked with pull requests on a git platform like GitHub (●4.4).

In terms of some of the tooling used in the experiment, we observe mixed levels of experience. The minority of participants (43.8%) indicate to have some experience in using GitHub Actions (●2.2, see Figure 5.4). Gradle, the build tool that produces the CI build logs, is rated slightly higher (●2.6) but also has a significant number of participants (56.3%) that have no to very little experience using the tool (see Figure 5.2). Mocking plays a role in one of the assignments. JPacman uses the Mockito framework in its tests. The participants have varying degrees of experience in using Mockito (●2.8, see Figure 5.3).

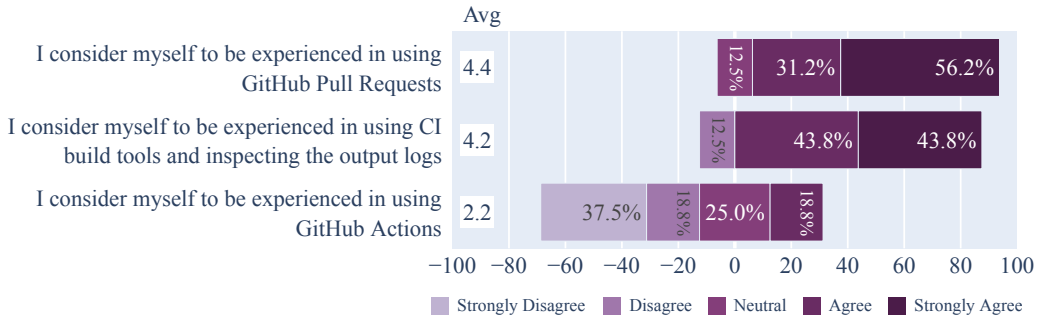


Figure 5.4: Participants' experience with CI.

The experiment was conducted using JPacman: a simple Pac-Man style game used for software education at Delft University of Technology. Some of the participants have used JPacman earlier in their studies. However, none of the participants indicated to still be very aware of JPacman or its project structure (see Figure 5.2. In general, a minority of the participants have some previous experience with the software project (● 2.4 ▃▆▁▂).

Opinions and Habits

We cover the opinions and habits of the participants in three categories: testing behavior, CI behavior, and build failure awareness.

Testing Behavior Most participants usually run their tests inside their IDE (● 4.1 ▂▂▃▆, see Figure 5.5) and can therefore compare using TestAxis to solving a test failure using the IDE. When a CI build breaks due to a failing test, the majority of the participants investigates the build log to find the specific failing test and rerun it locally (● 3.9 ▂▂▃▆). A smaller group reruns the entire test suite locally instead of only the failing test (● 3.0 ▆▂▆▂). Participant 10 notes that *"Build logs can be quite verbose and therefore it may be hard to find which test is failing. Sometimes it is easier to just rerun all tests locally."*

We asked participants about their opinions on the importance of integration or system/end-to-end over unit tests to see whether that influences their view on TestAxis. Most participants do not think that writing integration or system/end-to-end tests is more effective at catching issues than unit tests (● 2.5 ▆▂▂▁). A possible explanation could be the cost of writing and maintaining higher-level tests as participant 9 mentioned that *"System tests often take more effort to set up even though they are useful, and therefore I think that unit tests are a better time investment."*

CI Behavior To support our assumptions of how developers handle failing tests they encounter in broken CI builds, we asked them to describe their workflow after encountering such a test. Appendix E.1 shows the full results. We expected that developers would first inspect the CI build log to see which test is failing. Then, if the available error message would not provide enough information to immediately come up with a fix, rerun the test locally. By inspecting the stack trace, they would fix the issue, then optionally locally verify

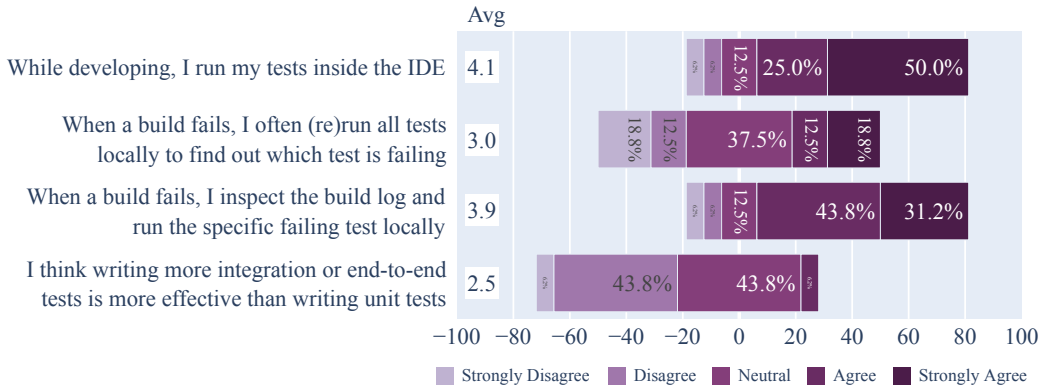


Figure 5.5: Participants' software testing behavior and opinions.

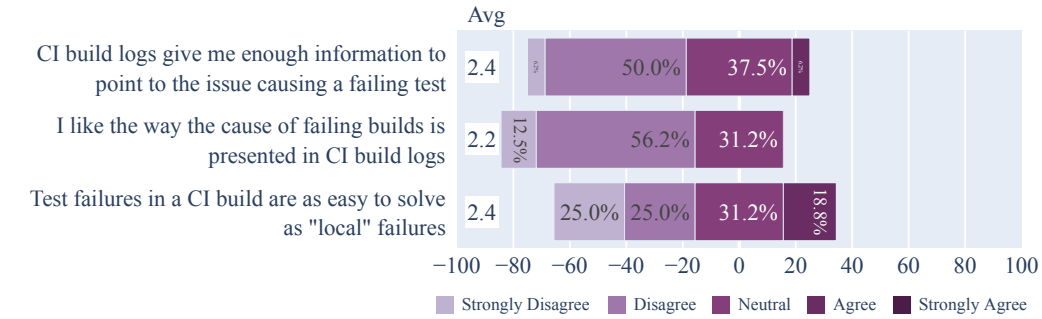


Figure 5.6: Participants' opinions on CI builds.

whether the test passes. Finally, we expected they would push the changes and as participant 4 put it *"hope for the best"*. All participants confirmed our hypothesis in the description of their workflow.

Interestingly, three participants indicated they would first write a new test locally to reproduce the issue. Also, three participants indicated that they use the debugger to find the issue of the failing test. Participant 12 described an alternative to the hypothesis using additional tooling: *"I look at the aggregated TeamCity logs to see what tests fail (if tests failed) and amend a fix to the commit and push it directly. TeamCity also offers the ability to see if the test is "flaky" - in this case, I simply re-start the build."*

In Figure 5.6, we show the results of the participants' opinions on CI builds. The participants indicate that they do not like the way failure information is presented in CI build logs (●2.2) and that the information does not help them enough to fix the issue (●2.4). One of the participants did, however, mention that Azure DevOps provided better-formatted test result reports than a raw build tool output, which they preferred. The majority of the participants also think that test failures they encounter in CI builds are more difficult to fix than the ones they encounter in their local development environment (●2.4).

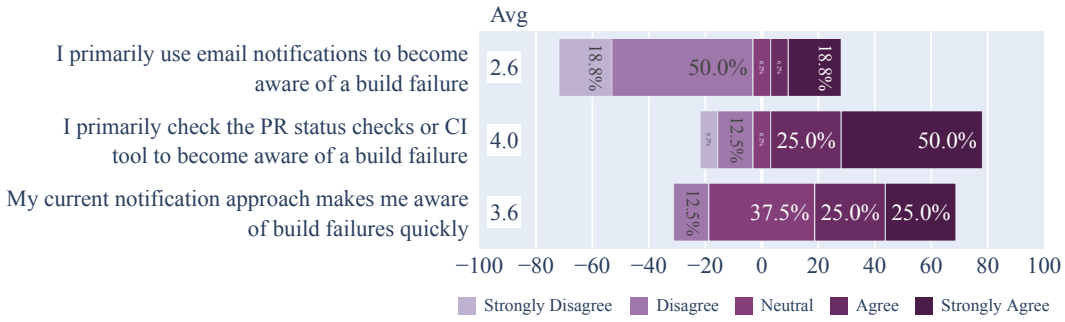


Figure 5.7: Participants' build awareness behavior and opinions.

Build Failure Awareness To verify our assumptions on the need for improving build failure awareness, we asked participants about their current approach of getting informed about build failures. Figure 5.7 shows the results. Platforms like GitHub often offer email notifications as the default option to be notified outside of the platform. However, only 25% of the participants indicate that this is their primary method of becoming aware of build failures (●2.6). Most participants primarily perform manual checks on their pull request to see the status of the build and are thus not actively notified (●4.0)

The participants also indicated alternative ways of getting informed about build failures. Three participants are informed by colleagues, six indicate to use notifications in chat services like Slack or Microsoft Teams, and one participant uses mobile push notifications.

The participants are not unhappy about their current approach of becoming aware of build failures (●3.6) but there is still room for improvement. During the pretest questionnaire, participant 14 stated that *"While I'm still working on a PR, I don't really need to be made aware of build failures quickly, only once it's ready to go to production."*

5.1.2 Failure-Fixing Performance in the Assignments

The following results are part of the outcomes of the within-subjects experiment.

The 16 participants conducted a single variant of all 8 assignments. Each variant was thus solved by 8 participants.

We measured the time between starting an assignment variant and fixing the issue. Figure 5.8 shows the results per assignment per variant. In all cases except 4a, we see that the median time to fix the issue is lower for the *with* variant than the *without* variant. For the assignments in the second and fourth category, we observe a high variability in the results. In Appendix E.3, we show the individual timing results for each participant.

Table 5.1 shows the mean failure-fixing time of all assignment variants. The percentages show the difference in performance between the *without* and the *with* variant. For category one, we see an overall improvement of 13.4%. For category two, this is 13.8%. The assignments in category three show the greatest performance improvement, on average 48.6%. Although assignment A of category four shows a performance decrease, on average

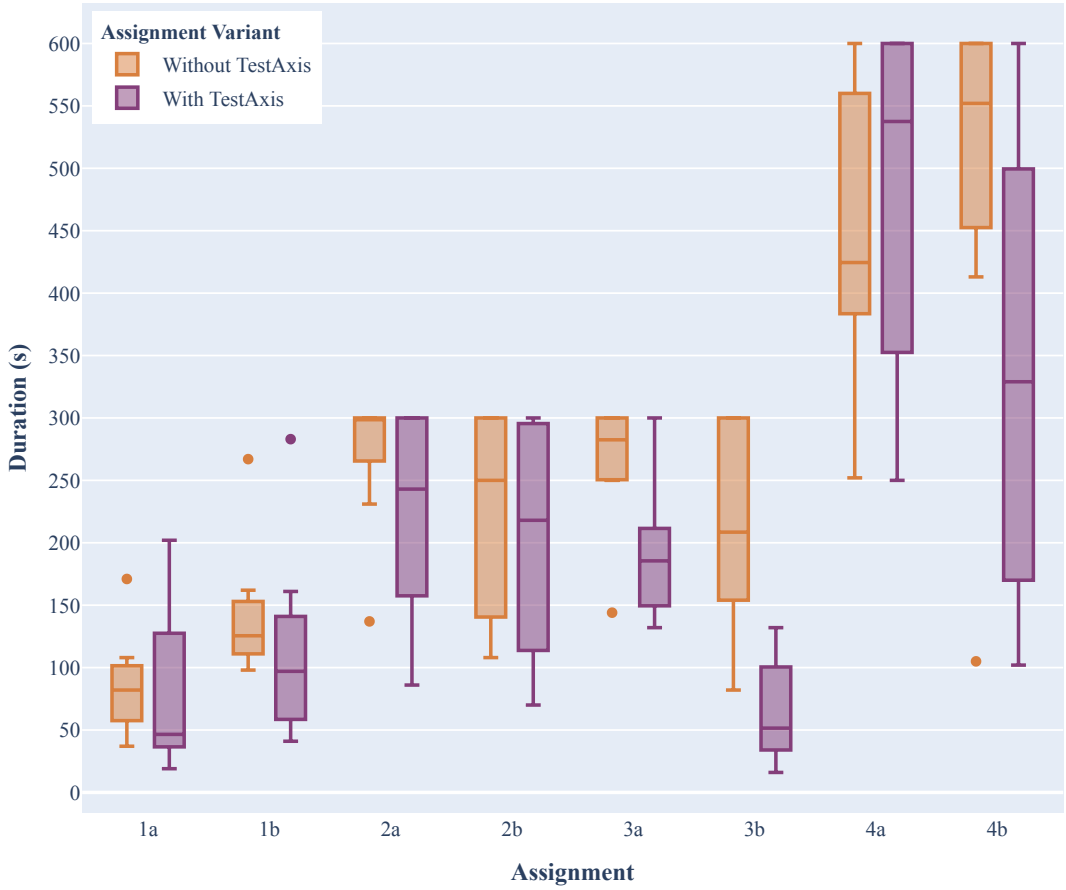


Figure 5.8: The failure-fixing time in seconds of both the *without* and *with* variant of all assignments. The first six assignments have a time limit of 5 minutes (300 seconds). For the last two assignments the limit is 10 minutes (600 seconds).

the performance difference of category four is 12.1%. Overall, the four assignment variants *with* TESTAXIS are conducted 22.0% faster.

The participants did not manage to solve all assignments within the time limit. When a participant hit the limit, we consider their failure-fixing time to be the maximum time of 5 minutes for categories one-three and 10 minutes for category four. Figure 5.9 shows an overview of how many times the time limit was hit per assignment variant. We observe a high number of hit time limits for category two. In general, we see a lower number of hit time limits for the *with* TESTAXIS assignment variants, except for assignment 4a. This corresponds to the results shown in Figure 5.8.

In the results, we left out one timing result for assignment 2b *with* TESTAXIS due to a miscommunication in the experiment introduction. The participant understood that the issue could never be in the test code and that it was not allowed to change the test code. As a result, participant hit the time limit for this assignment. The miscommunication was solved before assignment 2a was conducted.

Table 5.1: Average failure-fixing of the assignment variants in seconds with the performance improvement (%) per assignment.

	Assignment A			Assignment B		
Category	Without	With	%	Without	With	%
1	86	80	7.0%	143	115	19.8%
2	271	223	17.6%	224	201	10.1%
3	264	191	27.7%	231	65	69.5%
4	448	479	-6.9%	489	337	31.0%

Figure 5.9: The number of hit time limits per assignment (lower is better).

Another correction was made for one of the timing results of assignment 4a *with* TestAxis. The assignment description mentioned that the thresholds of the power-ups introduced in this assignment should be *increased* to fix the issue, while it should have said *changed*. After encountering the right location of the issue during the assignment, the participant tried to *increase* the threshold but this did not result in the right fix. The participant hit the time limit of the assignment. After we showed them the correct fix, they pointed out the error in the assignment caused them to not try this fix but instead *increased* the threshold. Therefore, we corrected the timing result to the point where the participant *increased* the threshold. We updated the assignment description for the remaining participants.

Finally, one of the participants concluded they did not have enough knowledge about the codebase of the software project and therefore they gave up on one of the assignments (assignment 4b in the *with* variant). The participant stopped after 9 minutes. We corrected this to 10 minutes, the time limit of this assignment.

After each assignment, we conducted the post-assignment evaluation questionnaire. See Appendix E.2 for the full overview of the results of these questionnaires. One of the participants erroneously did not submit the questionnaire for assignment 4 *without* TestAxis which is therefore not included in the results. We explore some of the results in more detail in the next section where we attempt to explain the timing results. Overall, an interesting result is the decrease in the average score of having to run the test locally to get more information

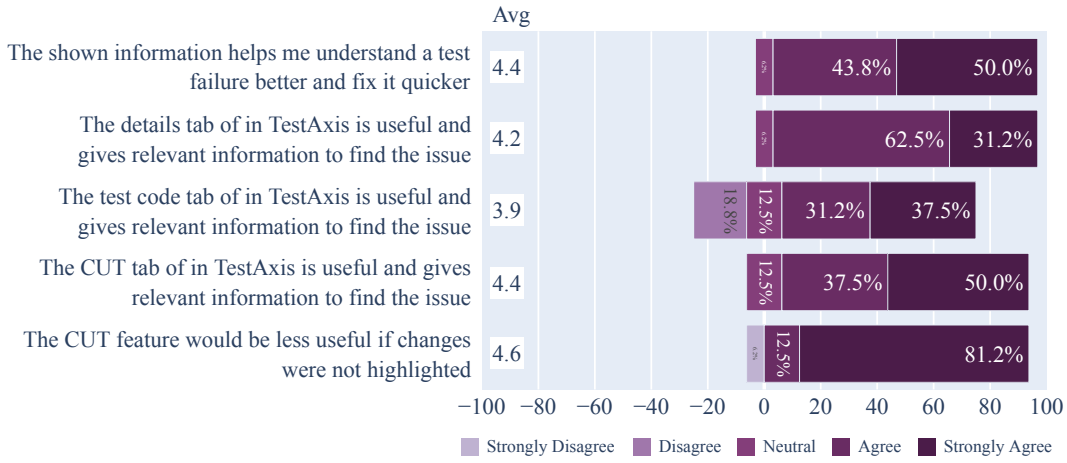


Figure 5.10: Participants' opinions on the usefulness of the main informational TESTAXIS features.

from 3.4 to 1.6. Also, the perceived time spent on finding out *which* test(s) failed dropped from 2.1 to 1.2, on average.

5.1.3 Usefulness of The Tool

The following results are part of the outcomes of the one-group pretest-posttest experiment.

We evaluate multiple aspects of the implementation of TESTAXIS that the participants used in the experiment: how useful the participants found the different informational elements, whether the tool meets the expectations of the participants, how the users experienced the tool, and what features they missed.

Usefulness of the Informational Elements

The majority of the participants find that the information provided by TESTAXIS in the various features helps them understand a failure better and fix it more quickly (●4.4, see Figure 5.10). The participants consider the details tab containing meta-information such as the test name and the interactive stack trace and the changed code under test feature most useful (●4.2 and ●4.4, respectively). Participant 5 even indicates that they *"already like just having the overview of failed tests a lot over a build log where I can see the name of the [failing] test but not much more"*. However, the participants rate the usefulness of the test code feature slightly lower (●3.9). Participant 8 mentioned that the test code tab may be unnecessary since there is already an "Open Test" button that opens the test in the main window of the IDE. The participants signal the importance of highlighting changes in the code under test tab and consider it an important part of the code under test feature (●4.6). Participant 13 explained why they think this feature is relevant: *"The highlights of CUT are very important since that's what you would normally do manually by thinking*

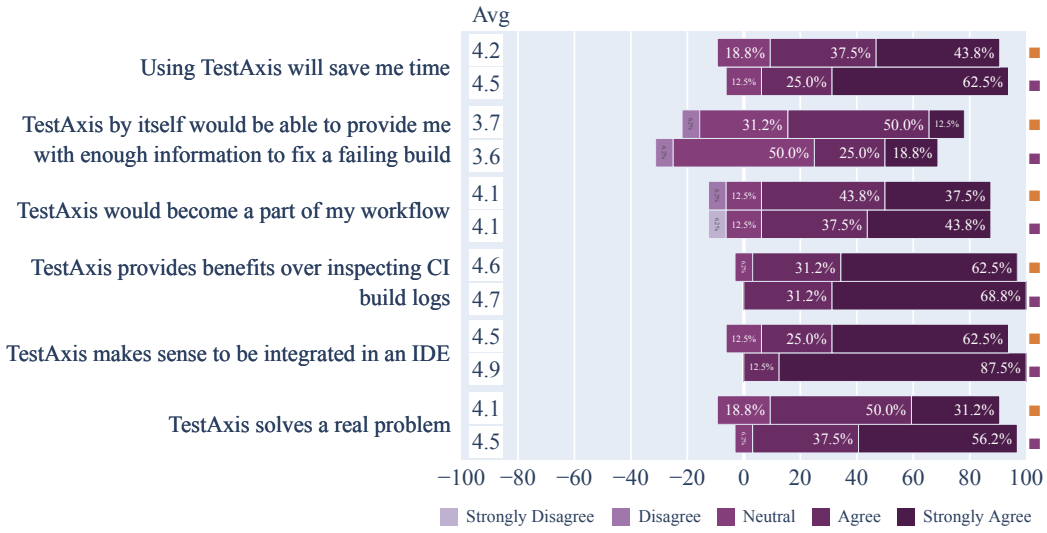


Figure 5.11: Participants' expectations before using the tool (orange square) versus the perceptions after using the tool (purple square).

about what changed and what the test could have covered. And this shows you everything automatically without any margin for error."

Expectations versus Perceptions

Figure 5.11 shows the results of the questions that were present in both the pretest as the posttest questionnaire. In the pretest, the questions were about an abstract description of a CI build test result inspection tool, whereas in the posttest they were explicitly about TestAxis. This measured the expectations of the participants and how they perceived TestAxis.

The results show that TestAxis meets or exceeds the expectations almost completely. After using the tool, the participants became more aware of how TestAxis solves a real problem and could save them time. Also, they found that integrating a CI build test result inspection tool into an IDE made more sense. However, after having seen an implementation of the features that were described abstractly in the pretests, participants are slightly less convinced that TestAxis would give them enough information to fix a failing build. Participant 12 mentioned that they considered it to not be enough because their builds *"often fail due to configuration issues or issues with Docker containers for instance"*. While TestAxis captures such extrinsic issues [51], it does not offer much support to fix these issues.

User Experience

In the experiment, participants got to use TestAxis extensively. Based on their experiences, we asked them about the usability of the IDE plugin. Figure 5.12 shows the results.

Overall, the participants found that TestAxis was easy to use (●4.5) and thought that most people would learn to use TestAxis quickly (●4.6). They did not think TestAxis should be a standalone (web) tool (●1.6) and did think that TestAxis was

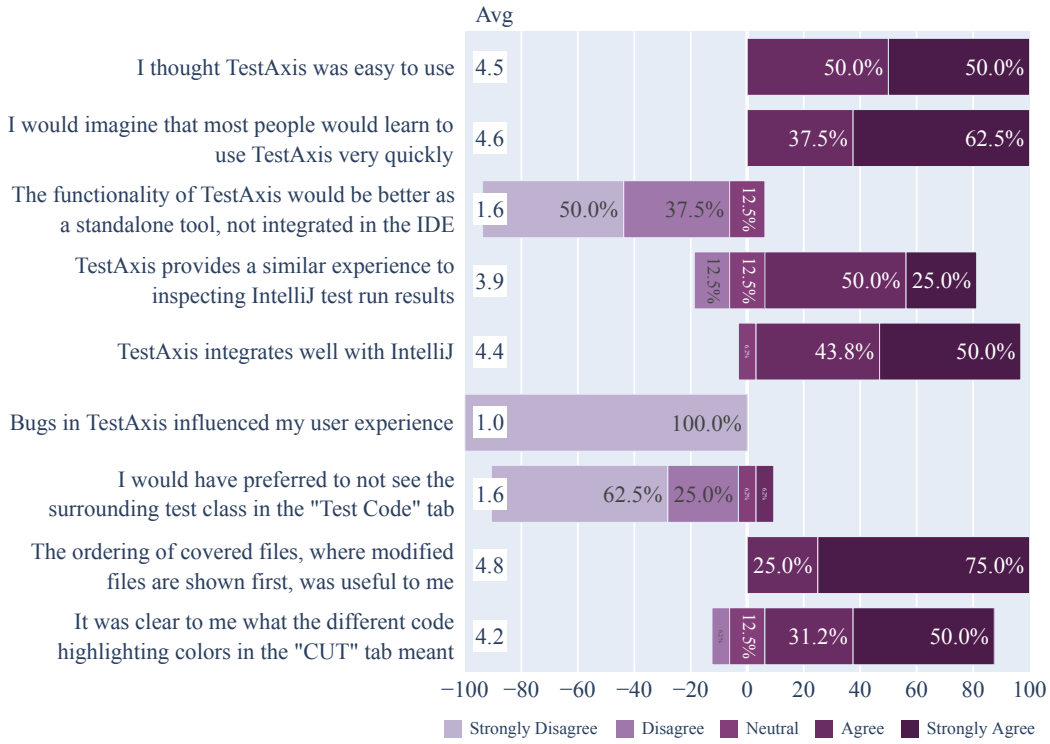


Figure 5.12: Participants' opinions on the user experience of TESTAXIS.

well-integrated with IntelliJ (●4.4 ▁▁▃▇). The participants rated the statement asking whether TESTAXIS provides a similar experience, in terms of presented information, as locally running a test in IntelliJ a bit lower (●3.9 ▁▃▃▇). However, some of the participants explained their reasoning out loud and stated that TESTAXIS has more features and that they, therefore, rated the statement negatively. None of the participants were interrupted by any bugs that occurred in TESTAXIS (●1.0 ▇▁▁▁, unanimous).

In the design of the tool, we made several assumptions about how information should be presented. For example, we chose to show the entire test class of the failed test case on the test code tab, and not only the body of the test method. The test method is highlighted to still be visually recognizable. The participants indicate that they agree with the choice as they would not prefer to see only the test method (●1.6 ▇▂▁▁).

Another assumption is showing the changed files at the top of the list of covered files in the code under test tab before unmodified covered files. The changed files are also labeled as such. The participants thought this was a useful feature (●4.8 ▁▁▃▇). The different highlighting colors for covered, changed and covered and changed fragments were clear to the participants (●4.2 ▁▃▃▇). Although one participant found the difference in highlighting colors between the code under test and the test code feature confusing and was missing a legend in the test code tab. A different participant also missed the legend on the code under test tab for non-modified files. Finally, one participant would have liked to have the ability to toggle the different highlighting colors because they found the code under test feature overwhelming.

Table 5.2: Missing features in TestAxis indicated by the participants of the experiment.

Feature	Number of Times Mentioned
Re-run the failing test locally	8
Open code under test in main editor	5
Right-click a test in the builds overview for a context menu	3
Suppress test health warnings	3
Full diff (including removed code) on the CUT tab	2
Easier navigation to highlighted CUT	2
Navigate from the stack trace links to the CUT tab	1
Show full PR details	1
Navigation from a line of code to a PR	1

Missing Features

During the execution of the assignments, participants mentioned features they missed in TestAxis. Also, they entered suggestions for features that would have improved their abilities to fix a test failure in the posttest questionnaire. Table 5.2 shows the missing features the participants indicated, sorted by the number of times a participant suggested the feature.

50% of the participants missed the ability to re-run a test locally to verify whether their attempted fix was indeed correct. Five participants would like to be able to open the files shown on the code under test tab in the main editor, either directly or by double-clicking a file. To directly navigate to a certain feature of TestAxis or to re-run a test, 3 participants missed the feature to right-click a failing test case in the builds overview for a context menu. The test health warnings currently do not offer any interactions. However, 3 participants indicated that they would like to able to suppress the warnings. For example, for *"tests that I know will take longer (such as a smoke test)"* as participant 8 mentioned.

5.1.4 Build Notifications

The following results are part of the outcomes of the one-group pretest-posttest experiment.

For each assignment *with* TestAxis, the participants got shown a build notification. In the posttest questionnaire, we asked their opinions on these notifications, see Figure 5.13.

The participants did not think the notifications were too intrusive (● 1.1 ▁▁). They consider the notifications to contain enough information to recognize the triggering commit or PR (● 3.8 ▂▆). Most participants do not prefer their current approach over the TestAxis IDE notifications (● 2.3 ▆▃). Thus, most of the participants liked this method of being notified (previous average score inverted 3.7). However, not all participants like being notified in the local development environment. Participant 12 mentioned: *"The problem is that builds take 9, 20, 30 minutes and then I'm already working on something else. I would*

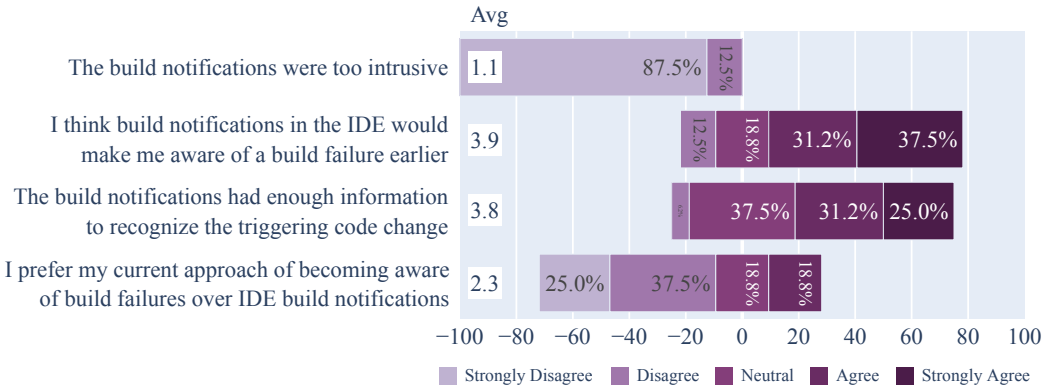


Figure 5.13: Participants' opinions on the IDE build notifications of TESTAXIS.

find it annoying to be distracted [by build notifications]." The majority of the participants do think that they would be notified earlier when they receive the notification in the IDE (● 3.9 ▁▃▅▇).

5.1.5 Test Health Warnings

The following results are part of the outcomes of the one-group pretest-posttest experiment.

In the pretest questionnaire, we asked participants about their view on the three test health aspects, see Figure 5.14. The participants care about the improved run time of their test suite (● 3.7 ▁▃▅▇) and think that flaky tests are difficult to recognize (● 3.8 ▁▂▅▇). We also asked whether the participants think that tests that fail often are as meaningful as tests that seldom fail. Since both the participants that thought these tests are more important and the participants that thought the tests are less important disagree with the statement, it is difficult to interpret the result (● 3.2 ▅▂▃▅).

We showed the participants several test health warnings during the experiment. In the posttest, we asked the participants how they experienced these warnings. Figure 5.15 shows the results.

The majority noticed the warnings (● 3.9 ▁▂▅▇) and a slightly smaller number of participants would act on them (● 3.5 ▂▅▇▇). The participants consider the warnings on often-failing tests and potential flaky tests to be useful (● 3.7 ▁▃▅▇ and ● 4.1 ▁▃▅▇, respectively). The participants neither agree nor disagree that the warnings on slow tests are useful (● 3.0 ▇▇▅▂).

5.1.6 Reflection on the Experiment

The following results are part of the outcomes of the one-group pretest-posttest experiment.

The first part of the posttest questionnaire asks participants to reflect on the assignments. This tells us whether the assignments are suitable for the experiment and if the behavior of participants in an experimental setting influenced the results. Figure 5.16 shows the results.

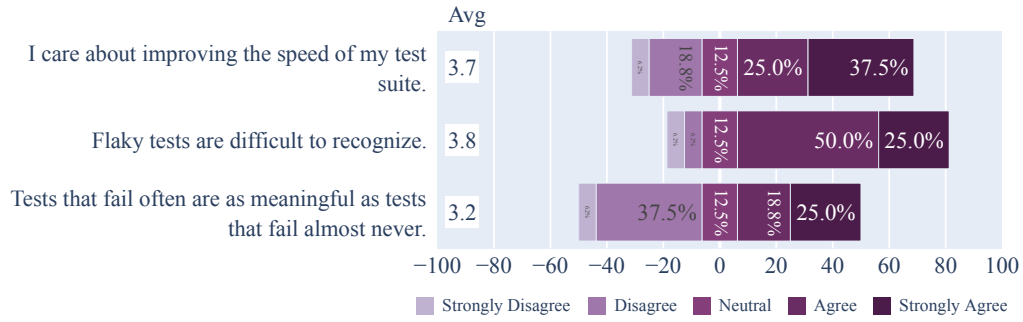


Figure 5.14: Participants' opinions on test (suite) health.

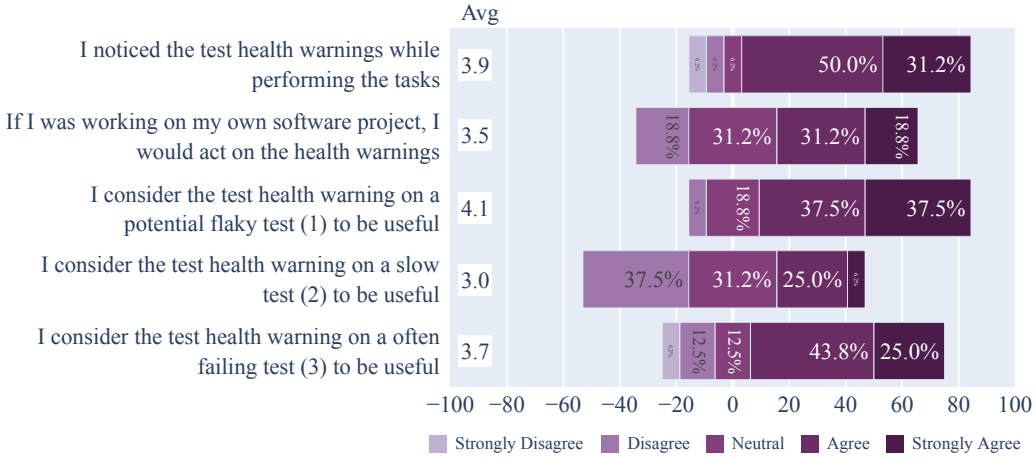


Figure 5.15: Participants' opinions on the test (suite) health warnings of TESTAXIS.

The participants consider the assignments challenging and interesting (●4.4), and sometimes similar to issues they encounter when working on software projects (●3.2). Participant 2 described the assignments as *"quite representative while still doable test tasks"*. Participant 8 agrees on this but also indicates that *"usually the fix usually requires more work than a small change"*. According to the participants, we provided enough guidance to conduct the assignments (●4.9). Participant 4 found that the assignments were *"sometimes really easy and did not always highlight the benefits of* TESTAXIS*"*. They found that some assignments could be fixed easily by a code review but that their opinion may be influenced by the fact that the participants act more as a reviewer than a code author in the assignments.

A minority of the participants would have solved the assignments easier or quicker outside the experimental environment (●2.8) or felt pressure due to the time limits (●2.6). During the experiment, participant 3 mentioned two times that they felt like they were changing their tactics *"to find the result quicker within the time limit"*. Most participants, however, applied the same tactics as they would have done in their own software projects (●4.0). Participant 9 indicated that they would have made more use of the

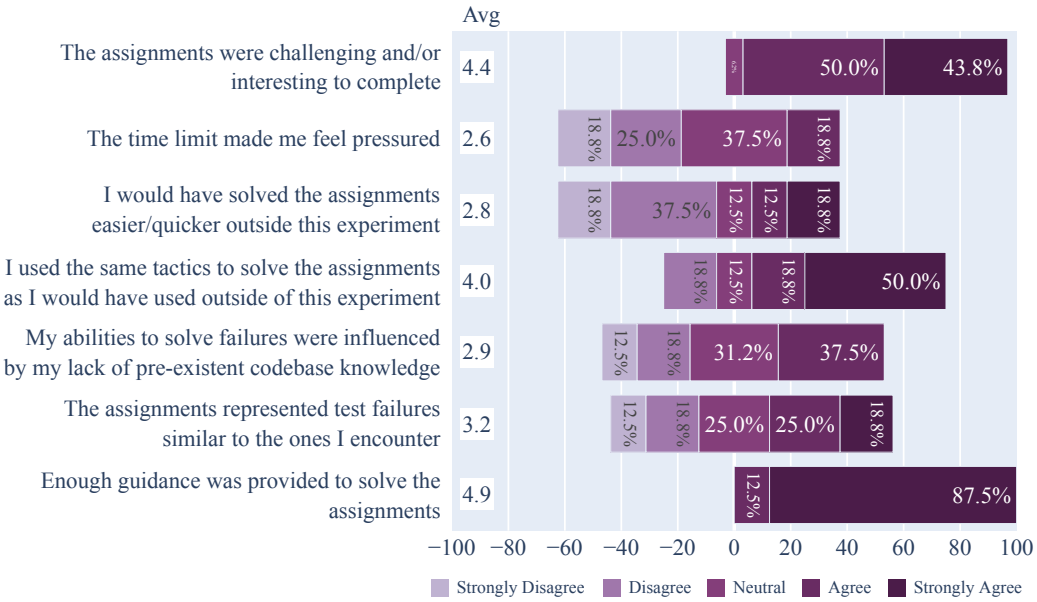


Figure 5.16: Participants' perceptions of external factors that may have influenced them during the experiment.

debugger outside the experiment but that they are *"not extremely familiar with the debugger in IntelliJ"*. A potential lack of pre-existenting knowledge about the codebase of JPacman influenced some of the participants' abilities to fix the assignments (●2.9▂▆▆_).

We also asked participants to reflect on the experiment as a whole, see Figure 5.17. The participants indicated that the quality of the assignments (●4.5__▆) and the questionnaires (●4.6__▆) was high. They also felt that the explanations of both the TESTAXIS IDE plugin and JPacman were clear (●4.8__▆) and that there were enough opportunities for feedback (●4.9__▆). Participant 13 mentioned that they were able to understand JPacman quickly. They also said that *"The project choice was really good for the experiment. It was not overly complicated like many open source projects. The scale of the project was not too simple but also not too complex."* In contrast, participant 14 indicates that because it is not very complicated it does not represent the *"much messy and less accessible"* codebases they usually encounter. However, most participants still think that JPacman allowed for interesting assignments that helped them provide an answer to the questions in the questionnaires (●4.3__▆). Overall, the participants enjoyed the experiment (●4.8__▆). Participants 2 and 14 indicated that they liked the assignments because they felt like solving little puzzles.

5.2 Performance Analysis and Discussion

Based on the results presented in the previous section, we now analyze the failure-fixing time performance results of the assignments. We explore these results by comparing them against several questions asked in the questionnaires. These comparisons may show a

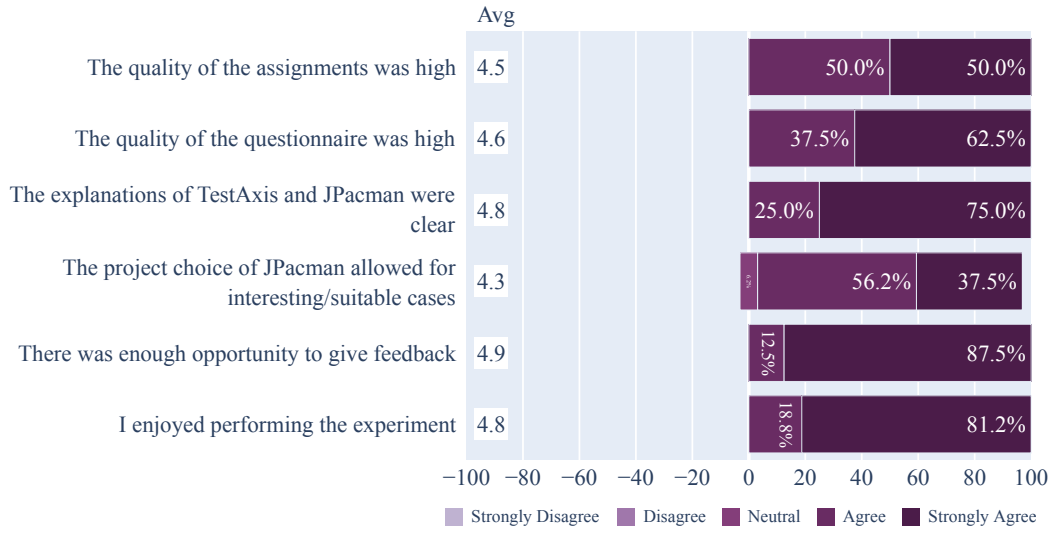


Figure 5.17: Participants' view on the quality of the experiment.

correlation between a participant's answer to a questionnaire question and their performance in a particular assignment, which may explain some of the timing results.

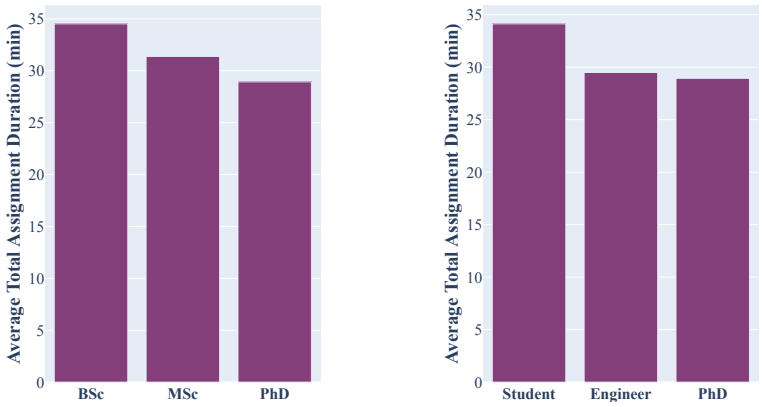
First, we investigate the influence of the background and the experience of the participants on the overall timing results to put the per-assignment results into perspective, see Section 5.2.1. Then, we analyze and discuss the assignments of the categories that belong to the three major informational features: meta information and test failure details (Section 5.2.3), test code (Section 5.2.4), and (changed) code under test (Section 5.2.5). These sections formulate an answer to RQ1-RQ3. Finally, we investigate the potential influence of the assignment ordering on the results in Section 5.2.6.

5.2.1 Influence of Background and Experience

The following analysis of the influence of background and experience on the performance results is based on the results of the within-subjects experiment presented in Section 5.1.2 and are compared with the results of the one-group pretest-posttest experiment presented in Section 5.1.1.

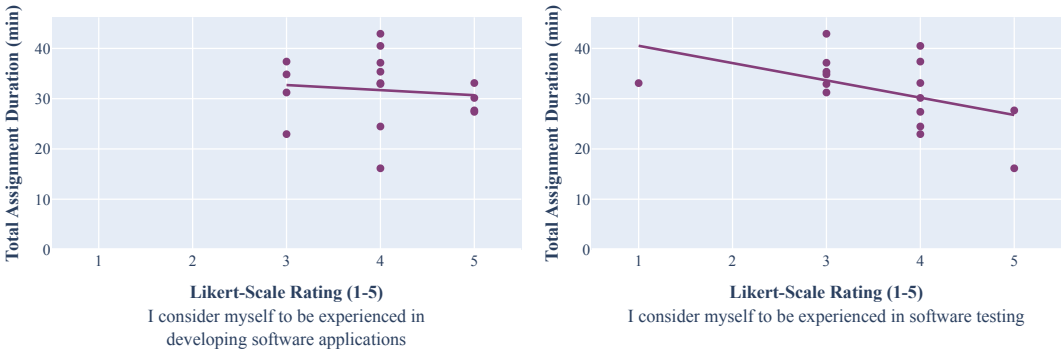
Before we analyze the results per assignment category, we first explore the overall influence of the experience and background of the participants. This shows us which per-assignment trends we will observe later are noteworthy because they differ from the global trends. To show this potential influence we compare the total duration of the programming assignments part per participant against their answers to the pretest questionnaire.

Figure 5.18 shows the average total assignment duration for the different backgrounds of the participants. In Figure 5.18a, we observe that the education level influences how quickly a participant is able to solve the assignments. The participants that are currently doing or have completed a PhD solved the assignments most quickly. This also follows from Figure 5.18b, where we see that the participants currently involved in a PhD were the



(a) Education (b) Occupation

Figure 5.18: The influence of the education level and professional occupation on the total duration of the programming assignments part of the experiment.



(a) Software Development (b) Software Testing

Figure 5.19: The influence of past experience of the participants on the total duration of the programming assignments part of the experiment.

quickest, closely followed by the software engineers.

Another trend we observe in the results is that more experienced developers were able to solve the assignments more quickly. This trend is visible for all pretest questions about the participant's experience. Figure 5.19 shows two examples: the influence of general experience with software development and the influence of software testing experience. Overall this shows us that the participants were able to rate their own experience accurately.

5.2.2 Statistical Significance of Performance Improvements

The following analysis of the statistical significance of the performance improvements is based on the results of the within-subjects experiment presented in Section 5.1.2.

In Section 5.1.2, we presented the performance results of all assignment variants. For all

Table 5.3: Statistical significance of the observed performance improvements.

Assignment	U	p	Reject H_0
1a	24.0	0.215	
1b	19.0	0.095	
2a	20.0	0.092	
2b	22.0	0.255	
3a	12.5	0.022	✓
3b	4.0	0.002	✓
4a	28.0	0.355	
4b	17.0	0.059	

assignments except one, we observe an improvement of the average failure-fixing time when using TESTAXIS. We now analyze whether these improvements are statistically significant.

We use the two-tailed Mann-Whitney U test [35] to attempt to reject our null hypothesis (H_0): "there is no difference between performing an assignment *without* or *with* TESTAXIS". Based on the evaluation of Q-Q plots of the data, we cannot assume normality and thus need a non-parametric test [40]. Moreover, our sample sample size (8 participants) per assignment variant is small. Therefore, the Mann-Whitney U test is an appropriate choice [40].

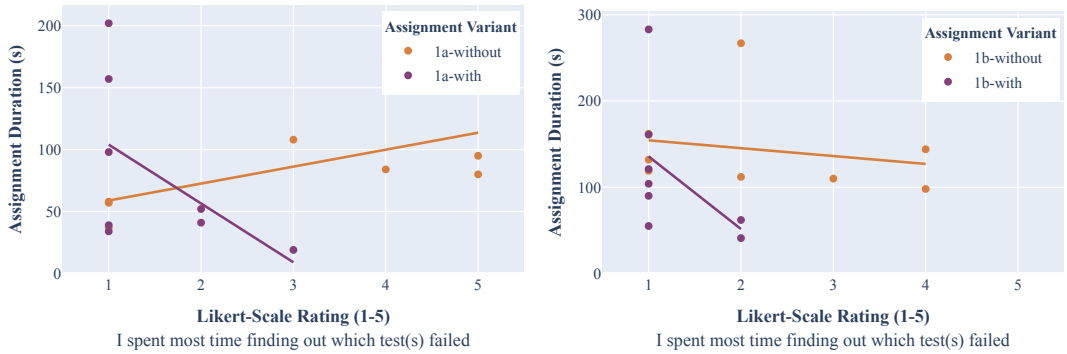
We want to reject our H_0 when $p \leq 0.05$. A p-value indicates the probability that the performance improvement occurred by random chance. The Mann-Whitney U test does not give a p-value directly but the U statistic that is based on a ranking of the results. This means that we reject the H_0 when $U \leq 13$ based on the critical values of a two-tailed Mann-Whitney U test with sample size 8 for both samples with the criterion $p \leq 0.05$ [55].

Table 5.3 shows the U values per assignment. It also shows p-values using a normal approximation. For assignment 3a and 3b, $U \leq 13$ holds, thus we can reject H_0 and conclude that these improvements are statistically significant. For all other assignments, we cannot conclude whether the performance improvement is statistically significant. This is mostly due to the small sample sizes, a higher number of participants with the same observed performance improvements would result in lower U/p values. For small sample sizes, the significance test only detects large effects [40].

5.2.3 Meta Information in the IDE

The following analysis and discussion of the performance improvements in the assignments of the first category is based on the results of the within-subjects experiment presented in Section 5.1.2.

To measure the influence of presenting a test failure in the IDE rather than in a CI build log, we conducted the assignments in category one: Test Failure Metadata (see Section 4.3.3). These assignments are solvable by just looking at the meta-information (the name of the failing test, and the stack trace of the failing test). Since the issues are relatively easy, the impact of showing an interpreted overview of the failed tests over scrolling through a build log should be visible in the amount of time spent on figuring out which tests failed.



(a) Assignment 1a

(b) Assignment 1b

Figure 5.20: Time spent on figuring out which tests failed indicated by the participant in relation to the failure-fixing time of the assignments of category one.

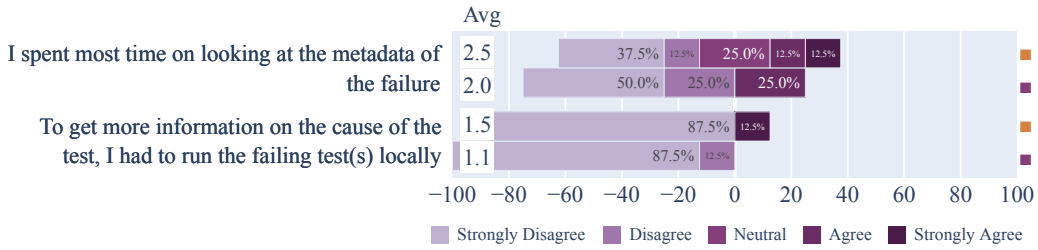
Other TESTAXIS features are less useful for these assignments putting the focus on the meta-information.

We measured the influence in terms of failure-fixing time performance. As explained in Section 4.3, we consider there to be an effect when both assignments show the same trend between the *with* and *without* TESTAXIS variants. Only in those cases, we know that the performance is not influenced by the group distribution per category. The results in Section 5.1.2 show that for both assignments the *with* variant was conducted more quickly. Thus, we can conclude there is a positive effect on the failure-fixing performance when using TESTAXIS for the assignments in category one. On average, the participants were 13.4% faster using TESTAXIS for the assignments in this category (test failure metadata).

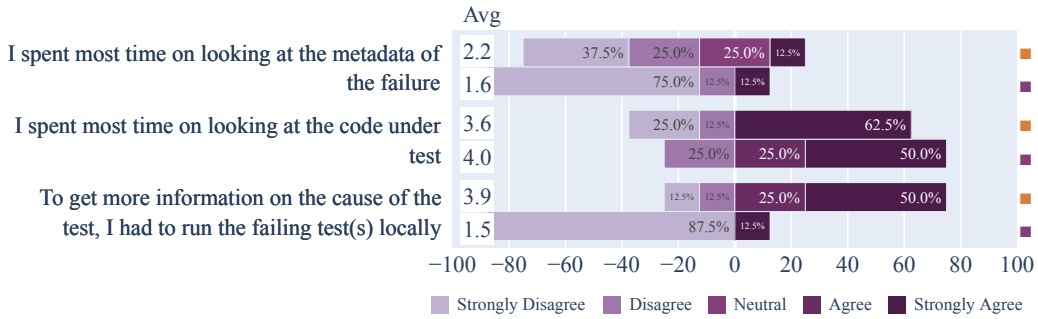
In both assignments, we see that for the variants *without* TESTAXIS, participants indicate that they spent more time figuring out which tests failed (see Figure 5.20). This suggests that it is easier to find which tests failed using TESTAXIS. In Figure 5.20a we observe that the participants of the *without* variant of assignment 1a that spent more time figuring out which tests failed, also took longer to solve the entire assignment. This trend, however, is not visible in the results of assignment 1b, see Figure 5.20b. This could be explained by the fact that assignment 1b is more involved and requires participants to spend more time investigating the code relative to figuring out which test failed.

Since the issues of these assignments can be easily spotted from the stacktraces, we would expect that participants spent the most time looking at the stacktrace and afterward quickly spot the issue in the code. However, the results in Figure 5.21 show that the participants rated the time spent on looking at the metadata relatively low for both the *with* and *without* variants. This may again be explained by the fact that the time they have to look at the stack trace is relatively short and that actually looking at the code and typing the change in the test code takes more time. For both assignments, we see that the participants spent more time looking at the location of the issue: the test code in assignment 1a (see Figure 5.21a) and the code under test in assignment 1b (see Figure 5.21b).

An interesting observation is that while in assignment 1a almost none of the participants ran the failing test locally, the majority of the participants in the *without* variant of assign-



(a) Assignment 1a



(b) Assignment 1b

Figure 5.21: Participants' post-assignment feedback on the *without* (orange square) versus the *with* (purple square) variants of the assignments of category one.

ment 1b did (see Figure 5.21). The issue in assignment 1a is a resource renaming issue that many of the participants could solve by intuition. Assignment 1b is more involved and requires participants to take a closer look at why and where the test is failing. Here it is clearly visible that the participants did not get enough information from the CI build log and had to run the tests locally to get a deeper understanding of the issue.

RQ1

What is the influence of presenting a test failure in the IDE over a CI build log on the time a developer needs to fix a failing test?

Key Points

- Developers solve test failures more quickly when the failure information is presented in the IDE over a CI build log. In the experiment, we saw an average performance increase of 13.4%.
- Developers indicate they need less time to find which test is failing using TestAxis.
- When the failure information is presented in an accessible format, developers do not need to run tests locally for failure details.

5.2.4 Failed Test Code in the IDE

The following analysis and discussion of the performance improvements in the assignments of the second category is mostly based on the results of the within-subjects experiment presented in Section 5.1.2. We compare some of the observations to the results of the one-group pretest-posttest experiment and in these cases state this explicitly.

We designed the assignments in category two to measure the impact of providing quick access to the test code of a failing test. This may help to spot mistakes in the test code itself and in understanding the intent of the test. Both assignments in this category can be solved by carefully reading the test code. Since the assignments add both new tests and new production code, the participants may also use the code under test feature to see what code the test is supposed to check.

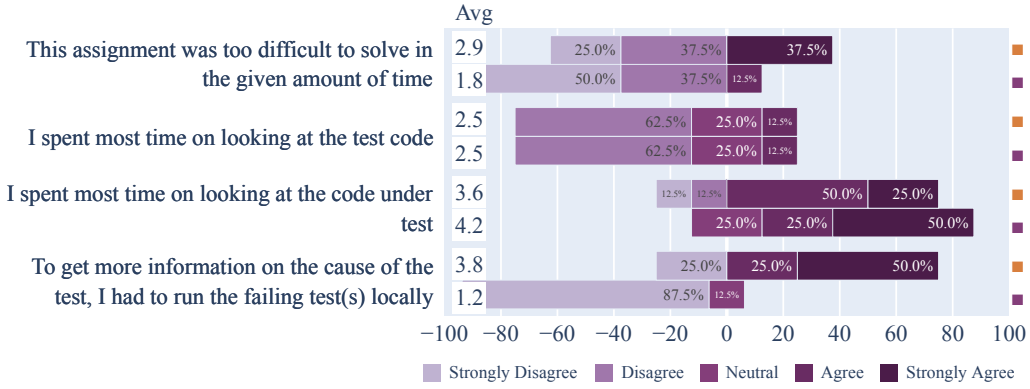
We again measured the failure-fixing time to determine the influence of showing the test code. In the results from Section 5.1.2, we observe an improvement in the performance for both assignments in this category. Therefore, we conclude that there is a positive effect on the failure-fixing time, that is not caused by the random participant distribution when the test code feature is made available. On average, the participants performed 13.8% faster in the assignment variants *with* TESTAXIS.

Assignment 2a has the highest number of participants that hit the time limit, see Section 5.1.2. While the issue to be spotted in the test code is simple, the added production code is more complex. This is reflected in the perceived difficulty of the assignment which is higher (average score 2.9, see Figure 5.22a) for the *without* variant (6 hit limits out of 8) than the *with* variant (average score 1.8, 3 hit time limits out of 8). Possibly, if given more time, the participants would have been able to solve the assignment, which would likely have increased the performance improvement since the *without* variant had a high number of time limits.

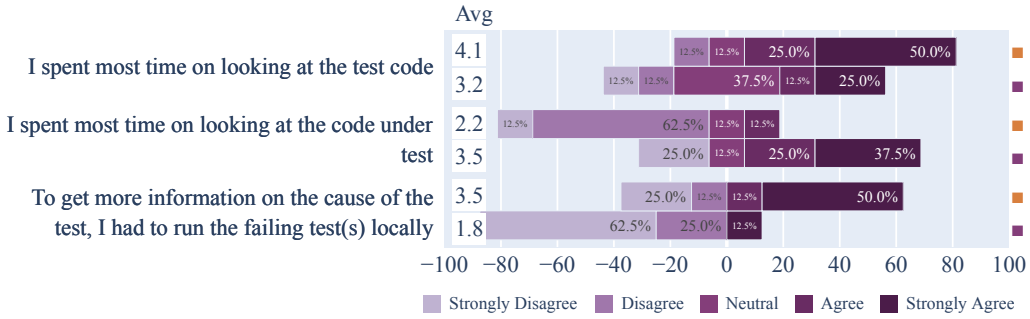
Since the production code of assignment 2a is complex, we also see that most participants spent the most time investigating the code under test rather than the test code where the issue is located (see Figure 5.22a). We observe the same trend for the *with* variant of assignment 2b in Figure 5.22b but not for the *without* variant. However, the relevant code under test highlighted by TESTAXIS consists of only three lines, making it unclear why the participants spent a similar (or more) amount of time on the code under test as on the test code. The participants that spent more time looking at the code under test also solved the assignment the slowest as shown in Figure 5.23. This may be because the participants could be more inclined to look for the issue in the production code than in the test code since there are only 2 out of 8 assignments where the issue is located in the test code.

An interesting observation is that the participants that spent the most time looking at the test code solved the assignments the quickest, see Figure 5.24. This may be because these participants simply focused more on the test code and therefore spotted the issue earlier which would make the relative amount they spent on the test code higher than for participants that also investigated the code under test. However, it also supports the hypothesis that it is useful to provide quick access to the test code to solve an issue earlier.

The newly added tests in assignment 2b make use of mocks. When we look at the pretest questionnaire of the one-group pretest-posttest experiment, we see that some of



(a) Assignment 2a



(b) Assignment 2b

Figure 5.22: Participants' post-assignment feedback on the *without* (orange square) versus the *with* (purple square) variants of the assignments of category two.

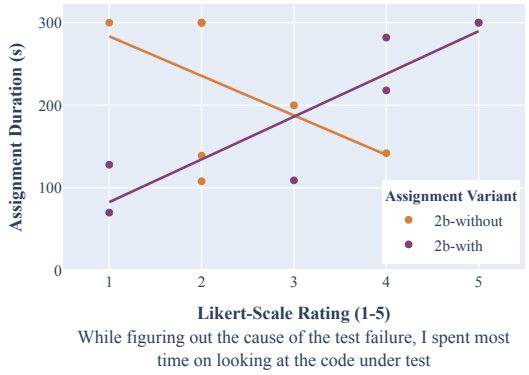
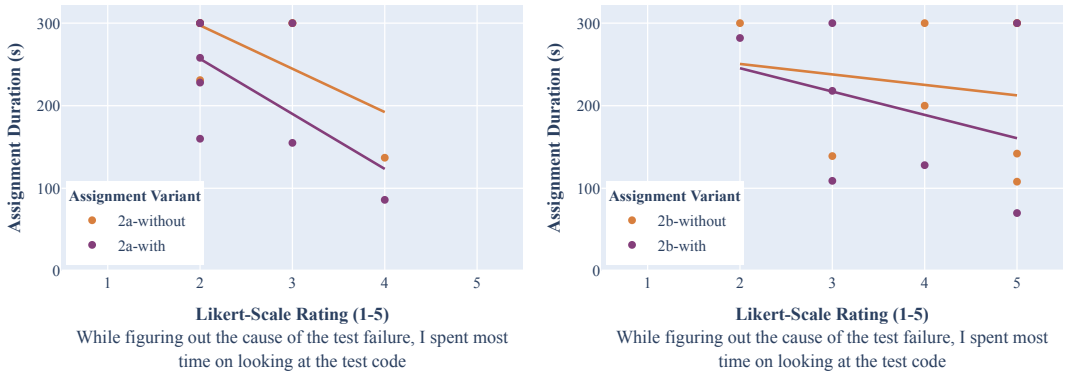
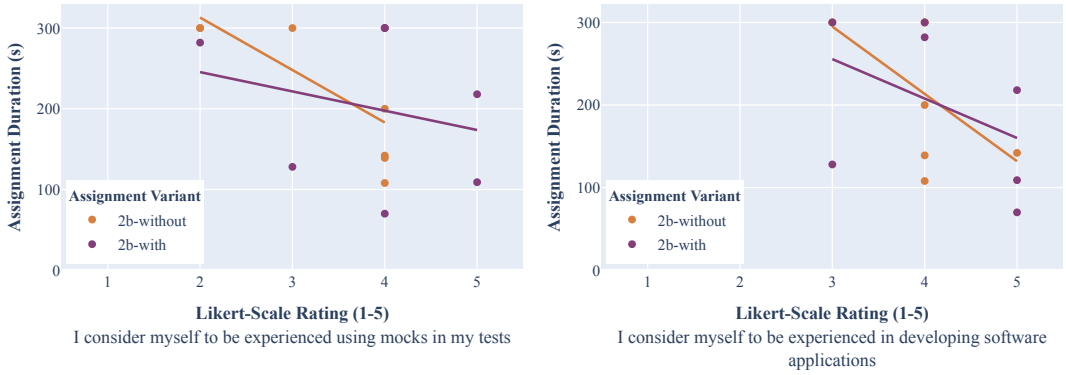


Figure 5.23: Time spent on investigating the code under test indicated by the participant in relation to the failure-fixing time of assignment 2b.



(a) Assignment 2a (b) Assignment 2b

Figure 5.24: Time spent on investigating the test code indicated by the participant in relation to the failure-fixing time of the assignments of category two.



(a) Mocking (b) Software Development

Figure 5.25: Participants' past experience in relation to the failure-fixing time of assignment 2b.

the participants are less experienced with the concept of mocking. We show the impact of this on the performance results in Figure 5.25a). However, this the trend aligns with general experience with software development indicated in the same pretest questionnaire (see Figure 5.25b) and is no different from what we observed in other assignments.

Similar to what we observed in assignment 1b, the participants rate the need for rerunning a failing test locally significantly lower when using TESTAXIS. Figure 5.22 shows that the average score for this need went down from 3.8 and 3.5, respectively, to 1.8 and 1.2.

RQ2

What is the influence of showing the test code on the time a developer needs to fix a failing test?

Key Points

- Developers solve test failures more quickly when they have quick access to the test code as part of the failure information. In the experiment, we saw an average performance increase of 13.8%.
- When there is an issue in the test code, developers almost never run tests locally when using TESTAXIS to gain more details.

5.2.5 (Changed) Code Under Test in the IDE

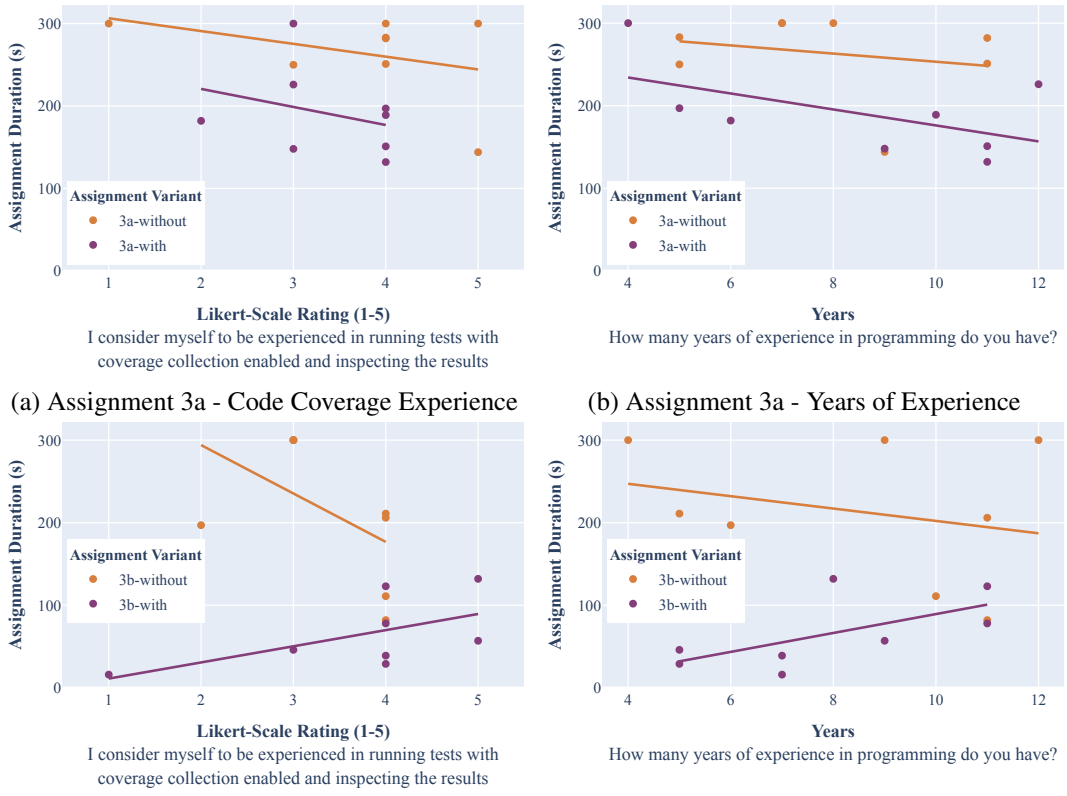
The following analysis and discussion of the performance improvements in the assignments of the third and fourth category is mostly based on the results of the within-subjects experiment presented in Section 5.1.2. We compare some of the observations to the results of the one-group pretest-posttest experiment and in these cases state this explicitly.

We evaluate the (changed) code under test feature using the assignments from the third and fourth category: code under test and advanced code under test.

(Simple) Code Under Test

We timed the execution of the assignments in category three: code under test. Section 5.1.2 presents the timing results. The participants solved the assignments in this category 48.6% faster *with* TESTAXIS than without (see Section 5.1.2). We observe an increase in performance for both assignments and thus conclude that TESTAXIS has a positive effect, not caused by the participant distribution, on the failure-fixing time on the type of failing tests in this assignment. In Section 5.2.2, we concluded that the performance improvement for both assignments in this category is statistically significant.

The way TESTAXIS presents the changed code under test may not be familiar to developers based on their experience. The combination of coverage data and change information was a new concept for most participants that they had to get used to. The closest existing feature is the inspection of code coverage information. Using Figure 5.26a and 5.26c we explored whether code coverage experience influenced the failure-fixing time. The participants indicated their experience level in the pretest questionnaire of the one-group pretest-posttest experiment. We found that, although the participants with more code coverage experience were able to solve the assignments quicker, this trend is also visible in all other assignments. It also aligns with the general experience in developing software applications indicated in the same pretest questionnaire (see Figure 5.26b and 5.26d), and therefore we may conclude that code coverage experience had no additional influence on the results over general experience. In fact, for assignment 3b we observe that the participants conducting the *with* variant that



(a) Assignment 3a - Code Coverage Experience

(b) Assignment 3a - Years of Experience

(c) Assignment 3b - Code Coverage Experience

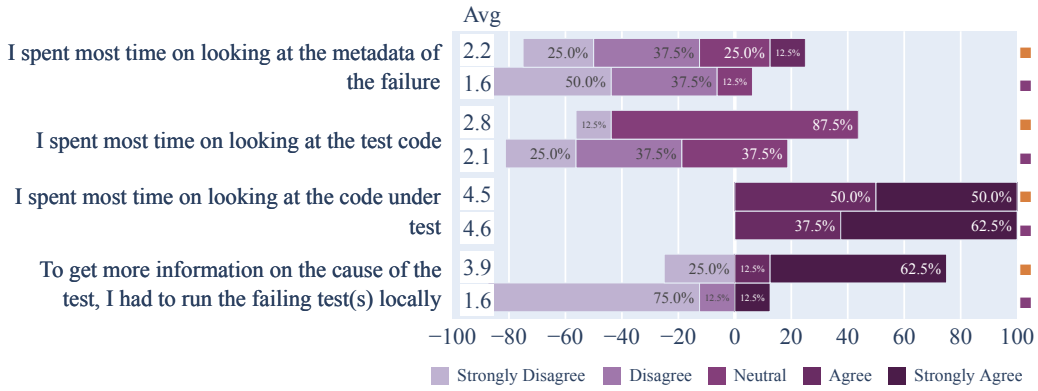
(d) Assignment 3b - Years of Experience

Figure 5.26: Participants' past experience in relation to the failure-fixing time of the assignments of category three.

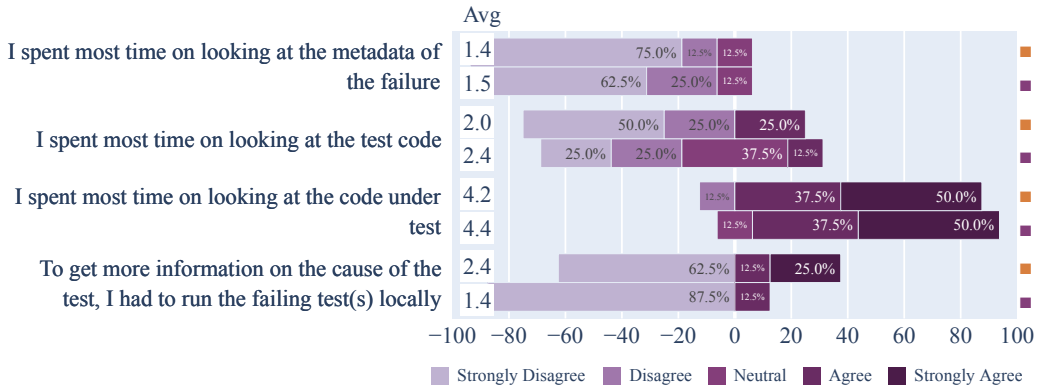
are more experienced with code coverage took longer to solve the assignment. However, this also follows the general experience trend in Figure 5.26d.

The participants indicated they spent the most time looking at the code under test for both assignments (see Figure 5.27). This is not surprising as the issues are located in the production code. The specific changes in the production code causing the issues, however, are more prominently highlighted in TESTAXIS because it only shows the changes covered by the test. The other changes are not all covered by the failing tests in the assignments. Thus reviewing the changes on a platform like GitHub will take more time. This explains that although the results show that developers spent a similar relative amount of time on the code under test (according to their own rating), their absolute failure-fixing time is lower when using TESTAXIS.

Like previous assignments, we see a significant drop in the number of participants that had to run the test locally in assignment 3a between the *without* and *with* variants (see Figure 5.27a. For assignment 3b, this drop is less obvious (from average score 2.4 to 1.4). This is likely because several participants were able to solve this assignment by careful review of the introduced changes without any additional information.



(a) Assignment 3a



(b) Assignment 3b

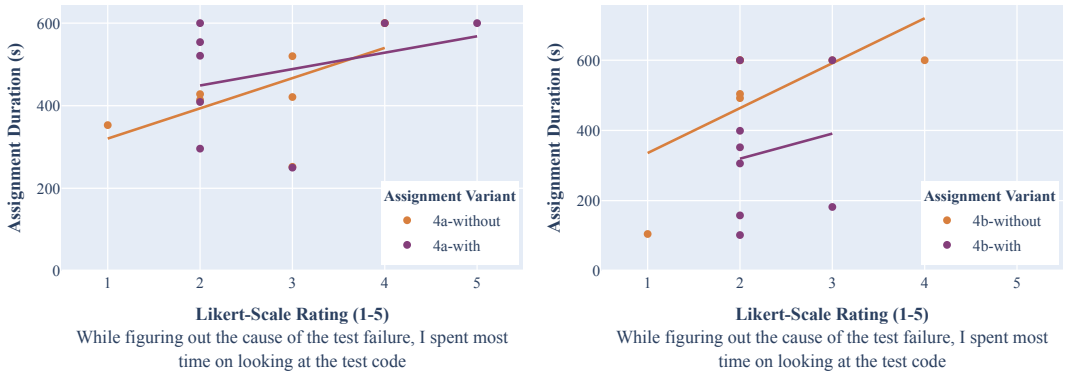
Figure 5.27: Participants' post-assignment feedback on the *without* (orange square) versus the *with* (purple square) variants of the assignments of category three.

Advanced Code Under Test

The timing results of the assignments in the fourth category are more surprising. While there is an average performance improvement of 12.1% (see Section 5.1.2), the participants of the *with* variant of assignment 4a were actually 6.9% slower than the participants of the *without* variant. Assignment 4b does show an improvement (31.0%). By the design of our experiment, this means we cannot conclude that the effect is caused by the introduction of TestAxis because the results of the two assignments do not show the same trend. This may indicate that the results are influenced by the random participant distribution. However, the levels of experience and background knowledge indicated by the participants in the pretest questionnaire are similar across the two groups. Thus, the results are less likely to be influenced by the background of the participant groups.

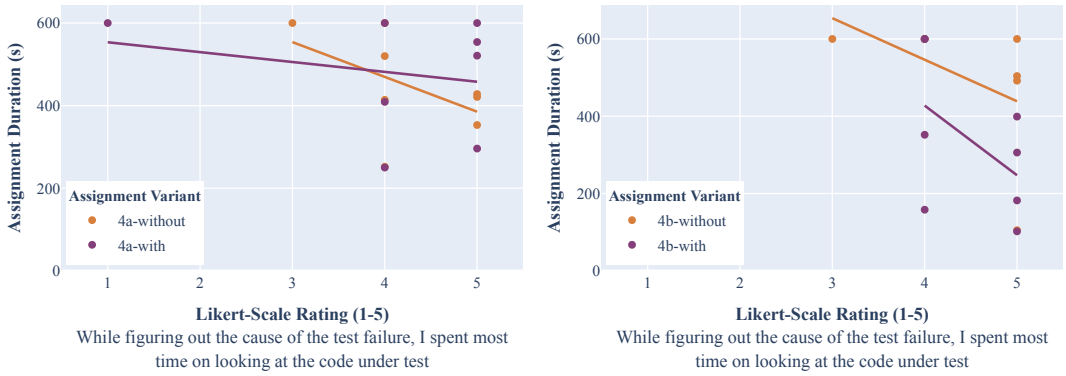
Similar to the assignments of category three, the experience with coverage does not have an impact on the timing results that does not follow the general experience trend.

In Figure 5.28a, we see that the participants of the *with* variant of assignment 4a, especially the ones that were the slowest, spent more time understanding the test code (while



(a) Assignment 4a (b) Assignment 4b

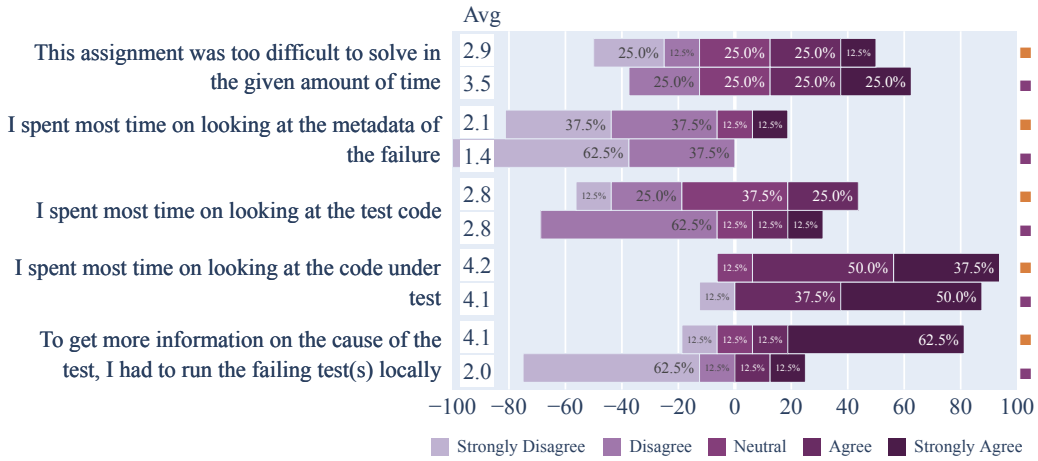
Figure 5.28: Time spent investigating the test code indicated by the participant in relation to the failure-fixing time of the assignments in category four.



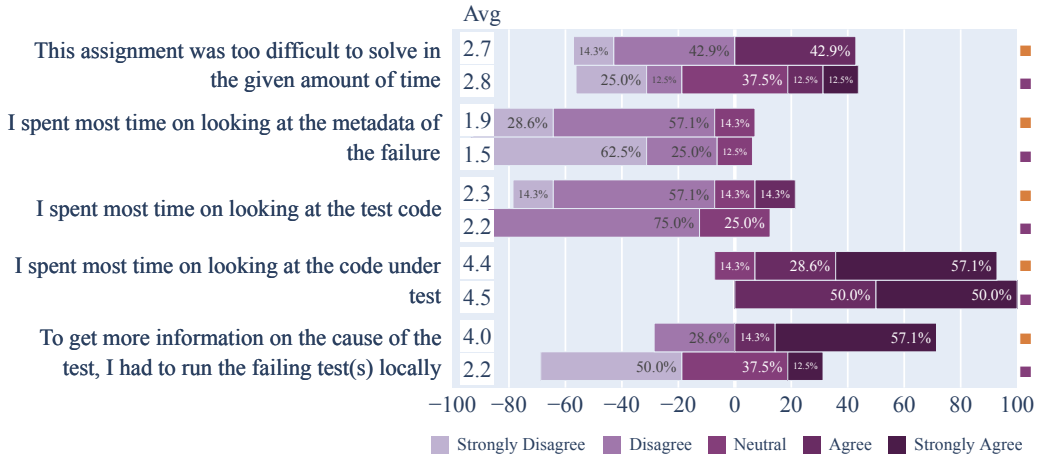
(a) Assignment 4a (b) Assignment 4b

Figure 5.29: Time spent investigating the code under test indicated by the participant in relation to the failure-fixing time of the assignments in category four.

the issue is in the code under test). A possible explanation could be that test code is more emphasized in TESTAXIS, while in GitHub you may be drawn more to looking at the Files Changed, which first shows the production code. An alternative explanation is that the participants that spent more time on the test code may have had more trouble understanding the problem. A reverse trend is visible in Figure 5.29a, where the participants that spent the most time looking a the code under test fixed the issue more quickly. In contrast to assignment 4a, we do not observe that the participants in the *with* variant of assignment 4b spent more time reading test code according to their own rating of how they spent their time (see Figure 5.28b). We do see the same trend that people that spend more time looking at the test code took longer to fix the issue. Again, we see the reverse trend for the code under test, see Figure 5.29b: the participants that spent the most time looking at the code under test found the issue more quickly. Figure 5.30 gives an overview of what parts the assignments indicated to have spent the most time on while figuring out the cause of the failing test. It



(a) Assignment 4a



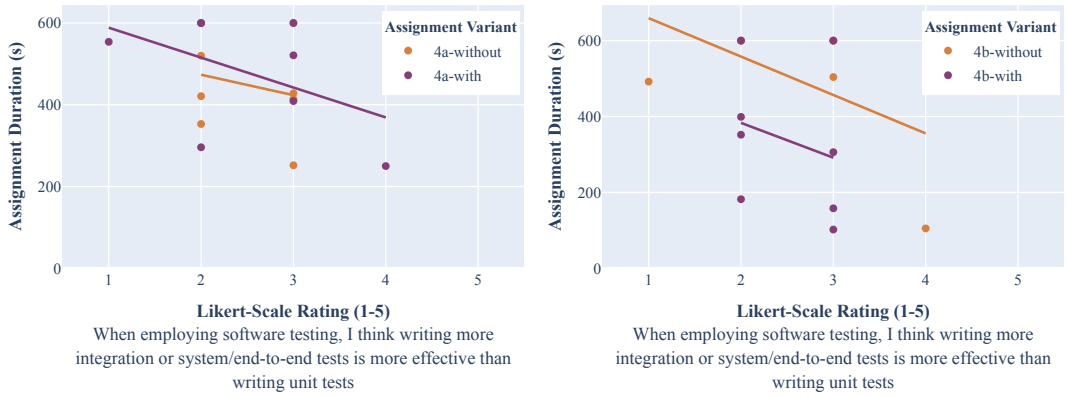
(b) Assignment 4b

Figure 5.30: Participants' post-assignment feedback on the *without* (orange square) versus the *with* (purple square) variants of the assignments of category four.

shows that most participants spent most time looking at the code under test, the topic of this category.

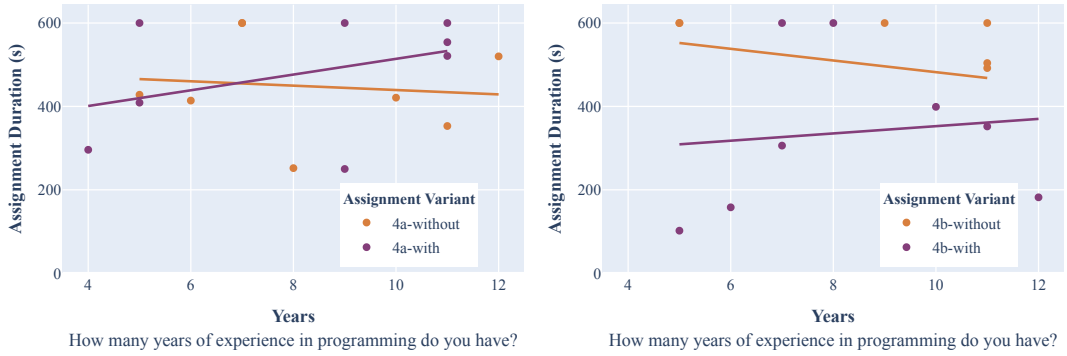
The failing test in the assignments in this category is an end-to-end smoke test of the application. Interestingly, we observe in Figure 5.31 that the participants that think writing such tests is more effective than writing strict unit tests were also quicker in solving the assignments. The participants gave their opinion on this matter in the pretest questionnaire of the one-group pretest-posttest experiment.

Another interesting observation is that we observe that the more experienced participants within the two participant groups of this category, perform worse *with* TestAxis than the less experienced participants, see Figure 5.32. The participants indicated their experience level in the one-group pretest-posttest experiment. The changed code under test feature is a



(a) Assignment 4a (b) Assignment 4b

Figure 5.31: Participants' opinion on effectiveness of integration/end-to-end tests in relation to the failure-fixing time of the assignments in category four.



(a) Assignment 4a (b) Assignment 4b

Figure 5.32: Years of programming experience in relation to the failure-fixing time of the assignments in category four.

new type of tooling that developers are likely not used to. A possible explanation might thus be that the more experienced participants stuck to their known inspection and debugging habits. The less experienced participants may find it easier to incorporate a new tool into their workflow.

Figure 5.30 shows that while still more participants had to run the tests locally when they did not use TESTAXIS, the number of participants that had to do this *with* TESTAXIS is slightly higher than in the previous assignments. This likely has to do with the difficulty of the assignments (average scores 2.9 and 2.7 for the *without* variant, 3.5 and 2.8 for the *with* variant) which is rated above average (2.1 for all variants *without* and 1.9 for all variants *with* TESTAXIS). During the execution of the assignments, we also observed that the participants that did run the test locally either did this to confirm whether their fix worked or when they had not found the solution after a given amount of time.

RQ3

What is the influence of showing the code under test, where the changed code is highlighted, on the time a developer needs to fix a failing test?

Key Points

- Developers solve simple test failures more quickly when they have an overview of the changed code under test. In the experiment, we saw an average performance increase of 48.6%. This improvement is statistically significant.
- The results cannot tell us whether there is a performance increase when using the changed code under test failure to solve failures of more complicated tests, such as end-to-end tests. In the experiment, we saw an average performance increase of 12.1% but cannot rule out the effect of the participant distribution per category.
- More experienced developers are less efficient than less experienced developers when using the code under test feature.
- When there is an issue in the code under tests, developers have a smaller need to run the failing test(s) when using TESTAXIS to gain more details.

5.2.6 Influence of Assignment Ordering

The following analysis and discussion of the influence of the assignment ordering on the results is based on the results of the within-subjects experiment presented in Section 5.1.2.

The ordering of the assignments may influence the results because the participants become more familiar with TESTAXIS and the codebase of JPacman over time. We analyze the duration of assignments per position in the ordering to see if such a learning effect is visible in the results.

Figure 5.33 shows the duration of the eight participants per assignment. It shows the duration per position in the assignment ordering, which is different for each participant. 11/16 of the plots suggest the influence of a learning effect because the assignment ordered later were completed more quickly. Obviously, we do need to take into account that the number of participants per assignment per position is small, about one.

In Figure 5.34, we show the mean and median time to complete all assignments ordered at a certain position. It is important to note that some assignments take a little over a minute to solve, on average, while others take about eight minutes on average. This explains the high variance (indicated by the vertical lines in the plot) per ordering position. The overall trend suggests that assignments at the end were solved more quickly than assignments positioned earlier in the ordering.

We conclude that there is an observable learning effect. In Section 5.4, we discuss the impact of this effect on the results and explain the measures we have taken to mitigate the impact.

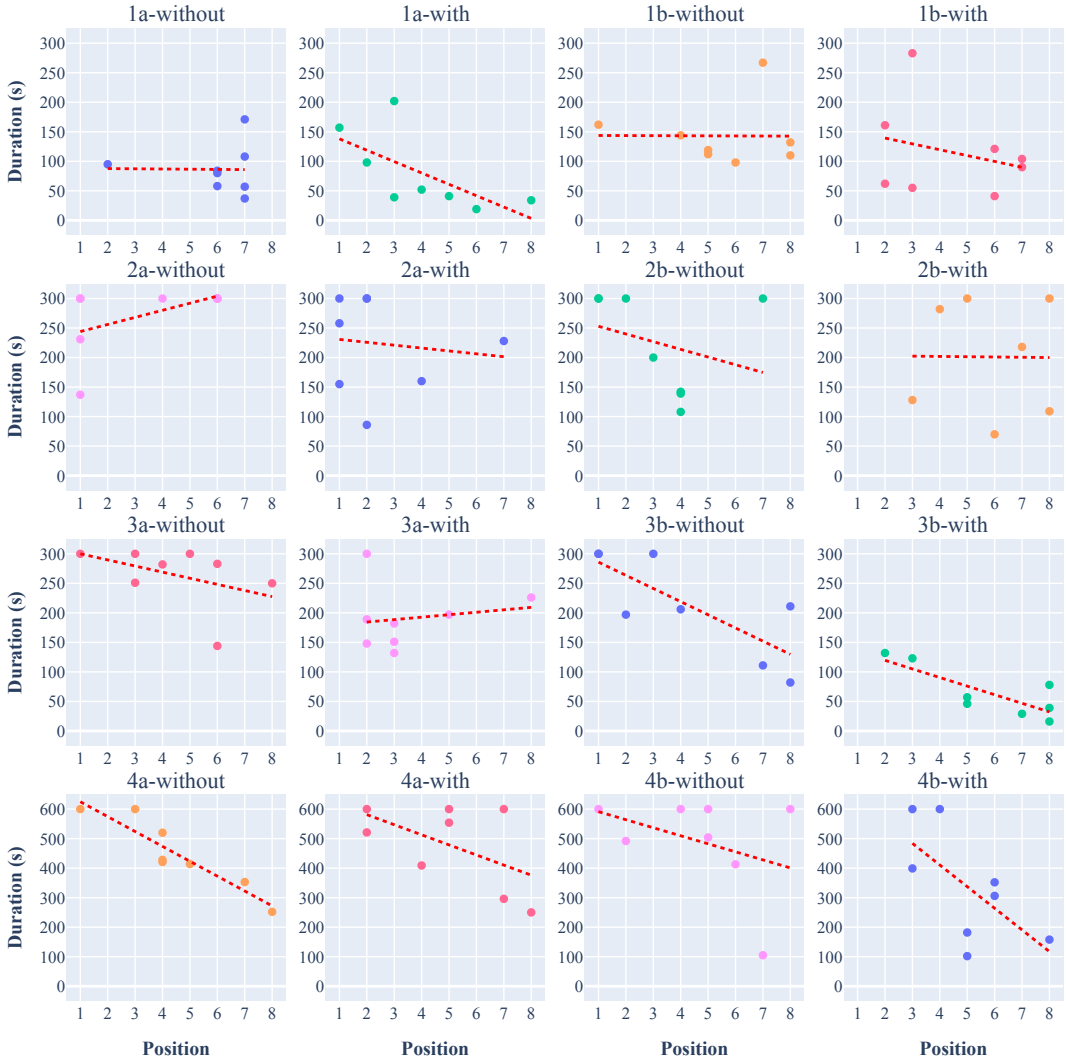


Figure 5.33: Influence of the assignment position in the participants' assignment ordering on the failure-fixing time per assignment.

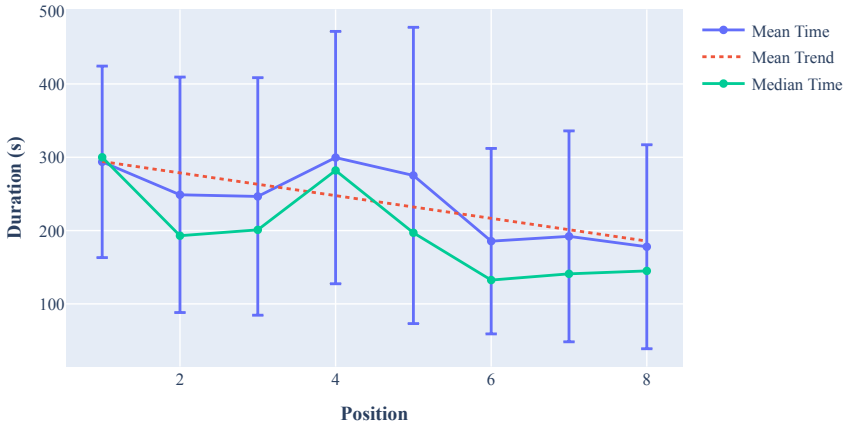


Figure 5.34: Influence of the assignment position in the participants' assignment ordering on the average failure-fixing time.

5.3 Usefulness Discussion

We evaluate the usefulness of three different aspects. First, we discuss the participants' opinions on the usefulness of having build notifications in the IDE in Section 5.3.1. Second, we explain the results of the questions on the health warnings and put them into perspective in Section 5.1.5. Finally, we discuss the overall usefulness of TestAxis as a CI build test result inspection tool in Section 5.3.3. In these sections, we also formulate an answer to RQ4-RQ6.

5.3.1 Build Notifications in the IDE

The following discussion of the usefulness of build notification in the IDE is based on the results of the one-group pretest-posttest experiment presented in Section 5.13.

TestAxis shows a notification in the IDE when a build has finished. During the experiment, we triggered such a notification for all assignments *with* TestAxis. In Section 5.13 we presented how the participants perceived such notifications.

Most participants prefer the TestAxis IDE notifications over their current approach of becoming aware of build failures. In their current approaches, most participants are not actively notified since they primarily manually check the build status (see Section 5.7). However, in the pretest questionnaire, the participants indicated their current approach does make them aware of the build failure quickly. Nevertheless, after having experienced the TestAxis notifications, the participants think that the build notifications in the IDE would make them aware of build failures earlier. One participant expressed the concern that they would not want to be distracted by these notifications while working on a different code change.

RQ4

To what extent do developers prefer to be actively notified of CI build failures in the IDE over their current approach?

Key Points

- Most participants prefer IDE notifications over their current approach.
- The most common current approach of becoming aware of build failures under the participants is checking the build status manually.

5.3.2 Health Warnings

The following discussion of the usefulness of test health warnings is based on the results of the one-group pretest-posttest experiment presented in Section 5.1.5.

By providing interpretation to the data collected by TESTAXIS, we can show test health warnings about potential flaky tests, slow tests, and often-failing tests. During the experiment, we presented several of these warnings to the participants. We described the full results in Section 5.1.5.

In the pretest questionnaire, we asked participants to give their opinions on the different health factors. The participants agreed most with the statement that flaky tests are difficult to recognize and after having experienced the test health warnings also indicate that they find the potential flaky test warnings the most useful. While the participants also care about improving the speed of their test suite, they did not find the slow test warnings very useful. During the experiment, participants indicated that they usually already know which tests are slow and that without any mechanism to suppress the warnings (explicitly considered a missing feature by three participants, see Section 5.1.3) they would consider the feature less useful because some tests are inherently slower than others. From CI build logs, it is difficult to inspect the past behavior of a specific test without going through a number of logs manually. Likely, participants therefore find the warnings on often-failing tests more useful and rated them higher than the slow test warnings.

In general, the majority of participants indicate that they would act on the test health warnings. However, 18.8% indicated they would ignore the warnings. Another 31.2% neither agreed nor disagreed with the statement. So, while there is a majority, not all participants are convinced the warnings would be useful enough to act on them. Obviously, the participants were only exposed to a small number of warnings in the limited time span of the experiment. It may be that such warnings in a developer's own software project are considered less or more useful.

RQ5

To what extent do developers find it useful to be warned about the health and history of a failing test?

Key Points

- Most participants would act on test health warnings, however, not all of them were convinced that the warnings are useful.
- The participants consider the test health warnings on potential flaky tests useful.
- The participants rate the test health warnings on slow tests neutral.
- The participants consider the test health warnings on often-failing tests useful.

5.3.3 Usefulness of TestAxis

The following discussion of the usefulness of TestAxis is based on the results of the one-group pretest-posttest experiment presented in Section 5.1.3.

We analyze the usefulness of TestAxis and its core features by looking at the posttest results on the usefulness of the different features and by comparing the participants' expectations of a CI build test result inspection tool with their perceptions (see Section 5.1.3).

For all test case execution results, TestAxis displays the information of its three main features in three tabs: the details tab, the test code tab, and the code under test tab. In general, the participants find that TestAxis is a useful tool to understand test failures better and fix them more quickly. The participants agree that all three main features are useful but do find some features more useful than others.

The participants find the test code tab that displays the source of the test case least useful. This feature provides quick access to the test code but does not offer additional insights that are not visible in the regular IDE window, therefore participants may perceive this feature least useful of the three main features.

The details tab shows which test failed and provides access to an interactive stack trace, similar to how stack traces are displayed in other parts of the IDE. The participants rate this feature as the second most useful one. Participant 13 mentioned the following about the failing test details feature: *"I think there is a significant difference in having* TestAxis *versus not having it. It shows me all the information I need. The stack trace shown in* TestAxis *is much less intrusive than in the build logs where it dumps the whole trace. Just having quick access is very useful."* Even though this feature mainly displays existing information developers already have access to, the participants still found it useful. This suggests that just providing easy access and obviating the need to inspect build logs can already help developers.

The code under test tab that shows the changed and covered code is considered most useful. The fact that it combines change information with code coverage data is appreciated

by the participants and they strongly agree that the feature would be less useful without it. This feature is the most innovative part of TESTAXIS and is not commonly available in IDEs such as IntelliJ. The participants likely consider this feature the most useful one because there is no similar alternative solution available.

TESTAXIS slightly exceeds the expectations of what the participants expected of the tool. After having used TESTAXIS, more participants were convinced that TESTAXIS solves a real problem. The participants think that TESTAXIS is useful enough to save them time. This perception aligns with our earlier analysis of the quantitative performance results. After the assignments, one participant thinks that TESTAXIS will be less useful in terms of providing enough information to fix a failing build. However, for all other participants, the expectations were met.

All participants either agreed or strongly agreed that TESTAXIS provides useful benefits over manually inspecting build logs. Most participants would consider making TESTAXIS a part of their workflow, both before and after having used it. All of them rated this statement neutral or higher except for participant 1 who strongly disagreed with the statement. They explained that *"*TESTAXIS *would not become part of my workflow because I don't do a lot of software testing and work in other languages than java"*.

RQ6

To what extent do developers consider a CI build test result inspection IDE plugin useful?

Key Points

- The participants find TESTAXIS useful in helping them understand a test failure better and fix it more quickly.
- The participants consider all three main features of TESTAXIS (failure details, test code, and code under test) to be useful. The (changed) code under test feature is considered to be most useful.
- The participants believe that TESTAXIS solves a real problem.
- The usage of TESTAXIS would save the participants time and they would make it part of their workflow.
- The participants strongly agree that TESTAXIS provides benefits over inspecting CI build logs manually.

5.4 Threats to Validity

The value of the results of the experiment is limited by the constraints of the experiment. We designed the experiment in such a way that as many external influences on the results could be avoided while not narrowing the experiment too much to avoid making it too specific. To

support the credibility of our results, we outline the threats to the validity of the study below. While some threats could be mitigated by the careful design of the experiment, others should be taken into account while considering the validity of the results. The three most important types of validity are internal, construct, and external validity [43].

5.4.1 Internal Validity

Internal validity indicates the reliability of the cause-and-effect relationship between the introduction of TESTAXIS and the observed effects in the results. The validity of the experiment results is threatened by any factors that influence the cause or effect. These factors are called confounding variables and the extent to which their impact can be minimized determines the internal validity [56].

As shown in Section 5.2.6, there is a learning effect visible in the results. Because the participants learn more about the tool and the codebase of the software project, they are able to solve assignments more quickly at the end of the experiment. We expected such an effect while designing the experiment and mitigated the impact of this learning effect on the results as follows. By randomizing the order of the assignments (see Section 4.3.4), we ensured that the results for each assignment are based on both early as late executions of the assignment in the experiment. The ordering of the assignment variants of categories is also randomized. Half the participants got the *without* variant of one of the two assignments belonging to a certain category first, and the other half got the *with* variant of one of the assignments of the same category first.

Creating two assignments that are similar enough to directly compare the difference in results when using/not using TESTAXIS is very complex (see Section 4.3.3). Obviously, participants can also not conduct the same assignment twice while using/not using TESTAXIS the second time because they would already know the solution. To mitigate this complexity, we do not compare the results of the two assignments per category that a single participant does. Instead, we compare the results of the *without* variant of a specific assignment against the *with* variant, executed by another group. This means that the composition of the two groups per assignment category could influence the results. While we randomized the group distribution process, one of the groups could, for example, be skewed in the number of experienced developers. For this reason, we only consider there to be an effect on the results after using TESTAXIS when the results of the *without* and *with* variant show the same trend for both assignments of a category. Thus, when one of the groups performs significantly better than the other group, we cannot conclude that the observed effect is not caused by the group distribution. This was the case for the results of category four.

The experiment environment was kept as consistent as possible between different participants and assignments. The participants all had access to the same computer with the same software installed. All participants used a remote connection to perform the assignments. For some participants, however, their keyboard layout did not match the layout of the experiment machine causing them to use different key combinations than they were used to. We tried to prevent any complications during the experiment caused by this issue by explaining the issue and by asking the participants to conduct a small task to get acquainted with working in a remote environment. Using our experiment control tooling described in

Section 4.6.2, we ensured that all software on the experiment machine was reset to the same state for each assignment. This mitigates any effects of easier access to certain tooling in later assignments. In the experiment, the participants had access to the same assignment descriptions and received the same amount of information about TESTAXIS and JPacman through the instruction videos.

Another factor in the experiment environment is whether the participants felt comfortable giving their honest opinions. The experiment was conducted in individual sessions with direct communication with the observer. The participants knew that their activity was observed while conducting the assignments and filling out the questionnaires. This could potentially have caused a Hawthorne effect resulting in participants answering certain questions more positively [1]. While it is not possible to show that this was not the case, we do observe negative answers to some of the questions of the questionnaire. This may suggest that the participants felt at ease and comfortable sharing their opinions. In the posttest questionnaire, some of the participants indicated that they would have solved the assignments easier or more quickly outside the experiment environment, while most of them did not.

The impact of maturation, the influence of other factors than the introduction of TESTAXIS between the observations, in this experiment is low. The full experiment is conducted within 90 minutes, for each participant, with an optional break of a couple of minutes. While we cannot rule out that anything other than TESTAXIS influenced the effect in the results, the exposure to external factors is very low. This makes it unlikely that such factors play a role in the results.

In the experiment, TESTAXIS is only used for a short period of time. A longer study of software development teams working with the tool in real projects is needed to measure its true impact. The short period of time is not long enough to build enough intuition to incorporate a new feature such as the (changed) code under test feature into one's workflow. Participant 14 confirmed this by saying *"For the best experience, requires a user to learn the intuitions of the tool"*.

5.4.2 Construct Validity

Construct validity is concerned with the degree to which a test actually measures the construct(s) it claims to be testing. In our case, we measure the performance and the usefulness in the experiment.

The performance results are based on quantitative data, the duration of assignment executions. The timing results may be influenced by different behavior induced by the experiment environment, previous questions, or previous assignments. However, the participants agree with the statement that they used the same tactics during the assignments *without* TESTAXIS as they would have done outside the experiment. We compared the performance results to the results from the post-assignment questionnaires in Section 5.2. These questionnaires contain ratings of how participants spent their time. During the experiment, we observed that the participants could not always judge their own time distribution correctly. Therefore, this questionnaire may not have captured the right results in all cases. The experiment notes that are part of the replication package contain our observations on the amount of time spent on certain tasks [12]. In a follow-up study, a more objective approach that monitors IDE usage,

such as WATCHDOG [9], could be considered to get a better indication of which tasks are most influenced by the usage of TESTAXIS.

We collect the participants' opinions on the usefulness of TESTAXIS through the pretest and posttest questionnaires. As described in the previous section, it is possible that the participants answered the questions more positively due to the one-to-one experiment setting. While we carefully formulated all questions, we can also not exclude any potentially leading questions that may cause us to not collect the true result.

5.4.3 External Validity

The external validity is concerned with the generalizability of the results. In our study, the following two factors influence this generalizability: the representativeness of the group of participants and of the programming assignments.

As shown in Section 5.1.1, the participants are a diverse group with mixed backgrounds. The participants have up to 12 years of programming experience and there is a mix between industry and academic software engineers. However, the group size is relatively small (16 participants) which may cause individual differences in background to have a larger effect on the results than in a larger group of participants.

The similarity of test failures in the programming assignments and real test failures developers encounter in real life are an important factor in the generalizability of the results. The assignments are conducted in JPacman (see Section 4.3.2). While this project is smaller than most applications, it does feature an extensive test suite, has a modern build pipeline, and employs design practices such as dependency injection. Although a larger project would have made the results more representative, the short duration of the experiment requires a project that can be understood quickly. The participants agree that JPacman allowed for interesting cases that were suitable to answer the questions (see Section 5.1.6). The cases we picked and designed are constructed to mimic test failures that could happen in any type of software project. We also designed them to be solvable using the functionality available in TESTAXIS, since we wanted to measure the impact of the individual features. However, that means we cannot conclude that the usage of TESTAXIS helps fixing failing builds faster in general, only in the specific cases. The participants neither agree nor disagree that the assignments are similar to the ones they encounter in their own projects, indicating that the generalizability of the assignments is a threat to the validity of the results.

5.5 Summary

In this chapter, we presented, analyzed, and discussed the results of the experiment. To discuss the performance improvement, we analyzed the quantitative timing results of the programming assignments. We observed an improvement in failure-fixing time when using TESTAXIS for the first three categories (test failure metadata, test code, and code under test). For the last category (advanced code under test), we cannot conclude that there is an improvement since it does not meet the requirement set by our experiment design that both assignments should show the same trend. When using TESTAXIS, the participants spent less time figuring out which test failed and had less need to rerun a failing test locally.

We discussed the usefulness based on the qualitative questionnaire feedback. We found that most participants prefer the TESTAXIS IDE build notifications over their current approach of becoming aware of build failures. Also, most participants agreed with the statement that they would act on test health warnings when they encounter them. They consider the warnings on potential flaky tests to be the most useful. In general, the participants find TESTAXIS useful in helping them understand a test failure and fix it more quickly. They consider the code under test feature to be the most useful. The participants think that there is a significant improvement in using TESTAXIS over inspecting CI build logs manually. Also, most participants would incorporate TESTAXIS in their workflow and think using it will save them time.

Chapter 6

Related Work

CI builds and test failures are well-explored topics in Computer Science. This chapter discusses a selection of the research done in these areas that is related or foundational to our work. In this chapter, we outline how developers currently consume information about failing CI builds due to test failures. We also investigate what types of additional information can be shown to developers to fix such failing tests more quickly. For these different types, we show how the process of collecting such a type of information, for example finding the code under test, relates to existing work.

6.1 Test Failures in CI Builds

CI is the practice of frequently integrating a developer's code change in the main code branch. Part of this practice is that every code change is built, verified, and tested by an automated service. CI is a best practice in both industry and open-source software development [15, 44]. Adopting CI in existing software systems, however, is not a trivial matter. One of the reasons that holds developers back is failing builds that rely on a developer's experience to be fixed while preventing the proposed change from being integrated [28]. With failure rates ranging from 13% to 35% [32, 65, 39], build failures are common. The fixing process is time-consuming, on average it takes around 42 [39] to 57 minutes [32] to find and fix an issue, which also makes it a costly process.

From discussions with developers, Amar et al. found that build logs are considered overwhelming due to the amount of details [2]. The authors conclude that developers want to *"find the most faults while investigating the fewest log lines possible"* [2]. A survey conducted by Hilton et al. [28] also concludes that there is a need for more assistance for fixing CI builds to prevent developers from having to work through *"hundreds of thousands of lines of output"*. Such assistance would improve the testing infrastructure of a project. In fact, a good testing infrastructure that makes it easier to add and maintain tests lowers the perceived complexity of the testing practices employed by a certain project [44].

Vassallo et al. developed a tool called BART that tries to solve this issue of overwhelming build logs by summarizing build failures and suggesting fixes based on information from external sources such as StackOverflow [66]. It reduces the time of fixing a failing build by

41%. However, its strength mostly lies in providing suggestions for failures related to code analysis, compilation, and dependencies. The authors conducted a user study and a survey to measure the effectiveness of BART. The results show that the tool is lacking in providing relevant and applicable suggestions for fixing build failures caused by failing tests. The tool shows the location, failure message, and optionally the stack trace of a failing test. Since this information is often project-specific, no fixes from external sources can be suggested. This indicates that more context is needed to fix failing tests.

Assisting a developer in finding and fixing the issue of a failing test in a CI build more quickly is important because the majority of build failures is caused by failing tests [7, 65, 48]. This could be explained by the fact that developers rarely run tests in their IDE [6], and thus only notice a failing test after a CI build has completed. Beller et al. have investigated the build logs of a sample of high-ranked open-source projects written in Java and Ruby. The authors conclude that failing tests were responsible for 59.0% and 52.3%, respectively, of the build failures [7]. Rausch et al. state that this was the case for up to >80% of the investigated builds [48]. However, Rausch et al. considered only 14 projects whereas Beller et al. used a subset of 1,359 projects from the TravisTorrent data set [8].

6.2 Assistance in Fixing Failing Tests

To assist developers in fixing failing tests, TESTAXIS provides context around the test failure that offers useful information and speeds up the fixing process. To know what context could be useful to the developer, it is important to know what information developers normally use while trying to fix a failing test. Beller et al. have monitored the behavior of developers after observing a test failure [6]. The results show that in more than 60% of the cases the developer immediately starts reading the production code (the code under test). Another 17% starts reading the test code first. However, after 5 seconds a significant number of users switched focus from the IDE to another window. A possible explanation is that developers reached out to external resources to help solve the issue.

The need for such external resources could be avoided by providing more context around the test failure. Zhang et al. proposed an approach that explains the reasons for test failures through comments in the test code [70]. It adds, for example, a comment before a line of code indicating which exception is thrown by that specific line. It also tries to suggest fixes by mutating the failing tests to see if it can find a variant that would pass. Using a statistical algorithm the lines most suspicious of causing the failure are determined and commented.

ReAssert also mutates test code to try to make the test pass [19]. The tool suggests mutations that result in a passing test as repair options. It can, for example, replace literals and change assertions. Because it only mutates test code, this method only works if the test code is no longer in line with the production code. If the failure is caused by a regression (the other way around), mutating the test code will not lead to the expected result because it will capture the wrong behavior of the production code.

6.3 Context of Failing Tests

To provide additional context to failing tests, TESTAXIS collects test results of CI builds to bring test failures into the developer's local development environment. TESTAXIS features multiple informational components that give context to a test failure. The most basic component shows the test name and failure message, which come directly from the uploaded test reports. We discuss related solutions that show test results in Section 6.3.1. There are also two more advanced components that try to find the test code and the relevant code under test. There exist multiple techniques to find the production code targeted by a test case which we discuss in Section 6.3.2. To identify only the relevant parts of the production code under test, we try to find the code changes leading up to a build failure. In Section 6.3.3, we discuss several methods to find such changes and argue why TESTAXIS can identify both tests that fail due to code changes as to external reasons, such as configuration issues. Finally, TESTAXIS notifies the developer of a failing build in the IDE to provide easy access to the information provided by the components described above. This information is provided by a rich interface in the IDE as a part of the TESTAXIS IDE plugin. In Section 6.3.4, we compare TESTAXIS to existing IDE plugins that provide build notifications or show build results.

6.3.1 Collecting and Showing Test Results

Before any context can be given, TESTAXIS first needs to know which tests have failed during a build. There are two main approaches to collect test results from CI builds: interpreting build logs or inspecting test execution reports generated by the test runner.

The first approach entails finding the relevant parts that show the cause of the build failure in a build log [14, 7]. This can be automated using various techniques. For example, by using a technique based on regular expressions [7, 48], by writing custom parsers [66] or by applying clustering techniques on the diff between failing and passing build logs [2]. All these techniques have or form knowledge about the structure of the build logs, making them error-prone when the format changes or specific to certain build tooling. If the techniques detect that the cause of the failure is failing tests, they can be used to parse the part of the build log that shows which tests are failing.

However, build logs often do not show which tests are passing and do not include information about test execution times. Moreover, the relevant parts of the build logs are outputted by various test runners such as JUnit, Jest, pytest, or PHPUnit, which do not output results in a uniform format. The second approach does not use the build logs but instructs test runners to output a test report file. In fact, most runners (including the aforementioned ones) can output the report in the same JUnit XML format [30]. This format includes for each test the name, location, execution duration, failure message, and stack trace. Section 3.1.2 shows an example of such a report.

Using these test reports, it is possible to show to the developer which tests have failed. There are CI platforms that already use these reports to display the test results of a CI build to the user. GitLab allows developers to upload JUnit XML reports as artifacts from within the build [45]. It then shows the test results to the developer from within the merge request interface, see Figure 6.1. Jenkins offers similar functionality through a plugin [23]. However,

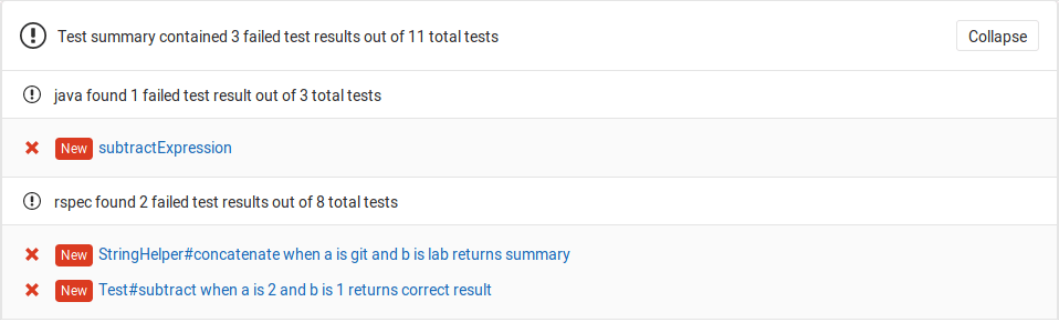


Figure 6.1: A summary of the failing tests in GitLab's Merge Request interface [45].

both platforms only show the name and failure messages of the tests, which does not always provide enough context to fix failing tests [66].

Brandtner et al. developed a stand-alone application that aggregates data from different CI tools (such as GitHub, Jenkins, and SonarQube) for a single build [15]. It combines all the aspects that the collected data covers into a single overview. The overview includes information about test results and coverage, see Figure 6.2. The information about test results is, however, limited because the tool only shows the number of tests that failed but not which tests those were.

6.3.2 Finding Code under Test

One of the ways to assist developers in finding the cause for a failing test earlier is by showing the production code that the test is targeting. However, this is not a trivial matter because there is rarely an explicit connection between tests and production code [52]. This results in the need for methods to trace production code to test code, in both directions. There exist static and dynamic approaches that accomplish this. The static approaches do not require code execution, which often makes them faster to use (because no tests have to be run). Dynamic approaches do require the execution of tests but can be more accurate. Since we collect the code under test during a CI build in which tests are already run, the overhead in terms of speed for dynamic approaches is negligible, making them a realistic option.

Static approaches There exist several static solutions. For example, approaches based on naming conventions [69, 52], call graphs [13, 52], or information retrieval [52].

Naming conventions are the simplest way to conclude a relationship. In this approach, the solution assumes that developers follow the convention of giving test classes the name of the class under test with a `Test` suffix [69, 52]. For example, when the convention is followed, the class `User` has a corresponding `UserTest` class. Obviously, this technique does not apply when developers do not follow this naming convention or when the tests are not at the unit level.

Another static approach is using static call graphs to find the interactions between test and production code [52, 13]. In this approach, all the methods that are called from a test are considered as being under test. This is the approach proposed by Van Rompaey

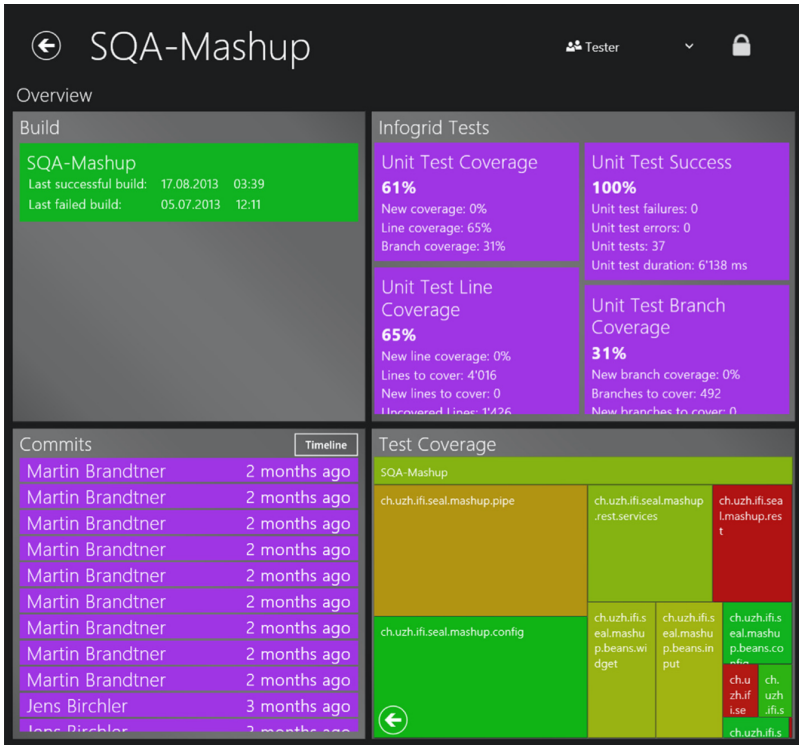


Figure 6.2: The interface of SQA-Mashup shows an aggregation of data of various CI tools [15].

and Demeyer [52]. It has the drawback that unrelated method calls, for example to helper methods, are also collected. To mitigate this issue, EzUnit, by Bouillion et al., lets developers select which of these methods are relevant, and registers this in a Java-annotation above the test [13]. Moreover, Van Rompaey and Demeyer also propose a variation to the static call graph approach: Last Call Before Assert (LCBA) [52]. This technique only considers the last method being called before the first assertion. While it is possible to implement this approach statically, the authors actually chose to use a dynamic approach for LCBA to avoid complexities with polymorphism and conditional logic.

Approaches using information retrieval techniques perform a form of textual analysis to determine which tests and production code methods are similar [52]. In the approach of Van Rompaey and Demeyer named Lexical Analysis, the assumption is made that the same vocabulary is used in the tests as in the code under test. The authors use Latent Semantic Indexing (LSI) to calculate the similarity which turned out to not be an effective approach.

Dynamic approaches In contrast to static approaches, dynamic solutions do require the execution of code. An example of a dynamic approach is TestNForce [29]. TestNForce tracks the relationship between test and production code. It, therefore, needs to know which tests are covering a certain part of the production code. It builds a (many-to-many) index containing which tests cover which methods (and which methods are covered by which tests).

The index is built by executing tests one-by-one and inspecting the covered methods using a process similar to typical code coverage collection. We also build such an index using coverage to find the code under test belonging to a certain test execution. However, our approach uses JUnit callbacks (see Section 3.1.3) to collect per-test coverage without having to resort to one-by-one test executions. Given that the coverage data must be collected during test runs and that all tests are already being executed during CI builds, the performance overhead is negligible, whereas this is a serious concern for TestNForce.

While the aforementioned approaches indeed find all code under test, the results may also include helper methods that are not always relevant to find the cause of the failing tests. Therefore, Qusef et al. propose a tool named SCOTCH+ that uses heuristics to discriminate between helper methods and relevant methods [47]. Like TestNForce, Scotch+ also executes the tests to identify the relationship between test and production code. SCOTCH+'s process of linking test code to production consists of two steps. The first step is dynamic slicing (a program decomposition technique to capture executed statements) and identifying the used classes that influence the outcome of the assert statements. Since these classes may include unrelated helper classes, the second step rules out classes with a low semantic similarity (using LSI) to the test class based on textual analysis. While using an LSI-based approach is not effective on its own [52], Qusef et al. show that it is effective for this second filtering step.

6.3.3 Finding Changes Causing a Build Failure

In Section 3.3.6, we describe how we collect relevant changes to a codebase that may have caused a build to fail. In summary, this involves two steps: finding the predecessor build and collecting all changed lines between the commit belong to the predecessor build and the commit belonging to the current build. For the first step, we ask git to provide the linear history of the commit belonging to the current build and find the most recent commit for which a build was executed. For the second step, we collect the changed lines by creating a code diff between the commit belonging to the predecessor build and the commit belonging to the current build.

Build Linearization and Commit Mapping The process of mapping builds to commits is what Beller et al. refer to as *"build linearization and commit mapping"* [8]. The authors state that while such a mapping may seem trivial, there are many cases in which it is not possible to establish a simple *"1:1 relationship"* between a commit and a build. Git's non-linearity and the CI build trigger conditions make establishing a predecessor relationship complex. The authors describe six scenarios that may occur while performing the build linearization and commit mapping process. Our approach is able to deal with four of these scenarios. The other two apply to so-called pull request builds (instead of push builds) that build a virtual integration commit of the pull request with the main branch which TESTAXIS does not offer support for since these commits are not present locally. TESTAXIS can deal with scenarios where multiple commits are included in a single push or where the predecessor build is not equal to the previously executed build. The diverging history of a merge commit is linearized by git and therefore also supported by TESTAXIS.

SZZ Algorithm A different approach to finding the commits is to use the SZZ algorithm [71]. This algorithm has a different starting point: it starts from the bug-fixing commit (*BFC*), which in our case does not exist yet. It identifies the *BFC* by linking commits to bug reports from an issue-tracking system using textual analysis on the commit messages to link the issue report and the commit. Using the *BFC* it finds the commit that introduced the bug by the following procedure. The SZZ algorithm first forms the set *LC(BFC)* containing all lines modified by the *BFC*. One of these lines must contain the fix for an issue introduced earlier, but irrelevant lines may be included. Therefore, the algorithm performs a Version Control System (VCS) `blame` operation for commit the parent commit of the *BFC*. For each line in *LC(BFC)*, the result of the `blame` operation shows which commit changed that line most recently before *BFC* did. Based on a few criteria described in [71], it then decides which commit caused the issue.

In our approach, the starting point is comparable to the parent commit of *BFC* since we expect the next commit to fix the issue causing the build to fail. From there, we also collect all previous changes. However, we have the advantage that we know when the system was still in good shape, namely when the predecessor build passed. Therefore, we have a fixed stop point to collect potential bug-introducing changes. While the SZZ-algorithm uses several criteria to then establish which faulty changes were fixed by the *BFC*, we leave this part up to the developer since the fix (and thus the *BFC*) does not exist yet.

Intrinsic vs. extrinsic bugs For our approach, we assume that *"a bug was introduced by the lines of code that were modified to fix it"* [51]. Rodríguez-Pérez et al. refer to this type of bug as intrinsic bugs [51]. However, there also exist extrinsic bugs: bugs that do not originate from an issue in the source code but from external issues. An example is an update in the behavior of an external dependency that is not reflected in the source code of the software. Rodríguez-Pérez et al. found that 9%-21% of the bugs they investigated were extrinsic bugs. These bugs were general issues that occurred during the execution of certain software, the types of issues that cause test failures may be different. Therefore, the 9%-21% extrinsic bugs rate is not necessarily transferable to our work. While our approach still captures extrinsic bugs (as explained below), we cannot always classify them correctly. Therefore, we cannot say whether our approach that only captures relevant code changes for intrinsic issues is effective enough at covering most test failures a developer encounters.

Rodríguez-Pérez Model Because current SZZ algorithms only capture intrinsic bugs, Rodríguez-Pérez et al. developed a model that generalizes SZZ algorithms to also capture extrinsic bugs [51]. The model is designed to evaluate the effectiveness of the current solutions that are all based on the earlier described assumption that *"a bug was introduced by the lines of code that were modified to fix it"*. The authors conclude that the extrinsic bugs that they found by applying their model to real-world projects were not detected by the SZZ-based algorithms they evaluated. Ideally, a tool like TESTAXIS can capture both intrinsic and extrinsic bugs as well. Therefore, we compare our solution to the model by Rodríguez-Pérez et al. and show that TESTAXIS identifies both intrinsic and extrinsic issues but cannot differentiate them.

The model describes the process of identifying *BFC*s and the corresponding first-failing commits (*FFC*s). The process consists of five steps which we describe below. We omit the first step that entails verifying whether the project uses a VCS since this always holds in our case.

1. *Identifying the BFC* Analyzing issue reports labeled as bug reports to find the commits that resolve the issue.

2. *Ensure the perfect fixing* The model does not consider *BFC*s that are not a so-called perfect fix. When there exists a perfect fixing, the fix is not spread over several commits, is not incomplete, and the issue that led to the *BFC* is never re-opened.

3. *Describe whether a bug is present* The model assumes a *perfect test* that can decide for each commit whether the bug that is fixed by the *BFC* is present. The authors state that *"there are no practical means to implement and run the perfect test"*, hence making a literal implementation of the model impossible. For the results in their own research, Rodríguez-Pérez et al. used a mental implementation of the perfect test to overcome this issue.

4. *Identify the FFC* The model assumes a linear history of changes to identify the *FFC*. Starting at the *BFC*, the *perfect test* from step 3 is executed for all previous commits until the first commit that makes the test fail is found. There are several possible outcomes of executing the test. If there were no changes in the source code that could have made the test fail, the bug is considered an extrinsic bug.

This model would perfectly capture all intrinsic and extrinsic bugs, however, it cannot be implemented in practice since *"perfect tests"* cannot be constructed [51]. TESTAXIS, however, does not consider issues as a result of general bugs that may occur in a software system but only considers issues as a result of failing tests. Thus, given this restriction, we have a test (the failing test) that can be run on commits to decide whether the bug is present or not (step 3). This test is what Rodríguez-Pérez et al. refer to as the perfect test.

In fact, we do not even have to execute this test (step 4), since the history of the test executions for each revision is already known to our system. Since tests fail regardless of whether the issue causing the failure is intrinsic or extrinsic, we capture both types. Using our approach to find the relevant changes we can assume that the issue is extrinsic when there are no relevant changes. However, the approach may give false positives, and therefore we may not always be able to conclude that a bug is extrinsic.

Moreover, in our case, we do not have to identify the *BFC* or consider perfect fixes since the fix does not exist yet. Thus, our approach satisfies all requirements of the model and is, therefore, able to detect both intrinsic and extrinsic bugs. It is, however, not always able to discriminate intrinsic and extrinsic changes.

6.3.4 CI Build Notifications and Results in the IDE

Typical ways to find out that a build has failed are e-mail notifications or built-in notifications of platforms such as GitHub or GitLab. However, Kerzazi et al. state that only 28% of the

Table 6.1: IDE plugins that show CI build statuses and/or results.

Name	IDE	Users	CI Service	Build Status	Build Logs	Notifications	Test Results	Test Insights
TeamCity [59]	IntelliJ	778,6K	TeamCity	✓	✓	✓	✓	✓
Jenkins Control Plugin [10]	IntelliJ	204,7K	Jenkins	✓	✓	✓	✓	
IntelliJ GitLab Pipeline Viewer [61]	IntelliJ	7,2K	GitLab	✓		✓		
Github Tools [36]	IntelliJ	6,3K	Travis CI / CircleCI	✓				
GitHub Actions [4]	IntelliJ	3,1K	GitHub Actions	✓	✓			
Hudson/Jenkins Mylyn Builds Connector [68]	Eclipse	Not reported	Jenkins	✓	✓		✓	✓
TESTAXIS	IntelliJ	-	All	✓	✓	✓	✓	✓

developers in their study pay attention to build notifications [32]. The authors also found that, on average, it takes 171 minutes for developers to become aware of a failing build. The developers have different reasons for this, such as relying on team members to notice failing builds or avoiding distractions and checking build statuses only periodically. There are, however, also developers that do actively monitor build notifications. In fact, two of the 28 interviewed developers by Kerzazi et al. indicate they use an IDE plugin for these notifications [32].

Downs et al. have studied the effect of notifying developers in their physical work environment [20]. They found that developers became more aware of failing builds by introducing a light-indicator that reflects the build status. In fact, the ambient awareness of failing builds shortened the duration of broken builds. In this work, we propose to bring such notifications to the IDE, the virtual work environment of a developer. The solution of Downs et al. does, however, have the advantage of not having to compete with other information that is presented to the developer.

We created an IDE plugin that notifies the developer of build failures but also presents information on the test results. There exist several IDE extensions that show the status of CI builds, sometimes with additional information. Table 6.1 shows all IDE plugins displaying CI builds that we identified in the JetBrains and Eclipse Marketplace. When referred to IntelliJ, we consider all IDEs built on top of the IntelliJ platform, such as PyCharm, WebStorm, and Android Studio.

The plugins have different characteristics. Three of the plugins notify the developers of build status updates [59, 10, 61]. Half of the plugins only show raw information (the status or the logs) of the builds [61, 36, 4], while the others also interpret the builds and show the test results [59, 10, 68].

Moreover, the TeamCity [59] and Hudson/Jenkins Mylyn Builds Connector [68] plugins also provide additional insights. Table 6.2 shows a comparison between the test insights features of these two plugins and TESTAXIS. TeamCity displays which tests failed and highlights stack traces, an example is shown in Figure 6.3. It also offers the ability to easily

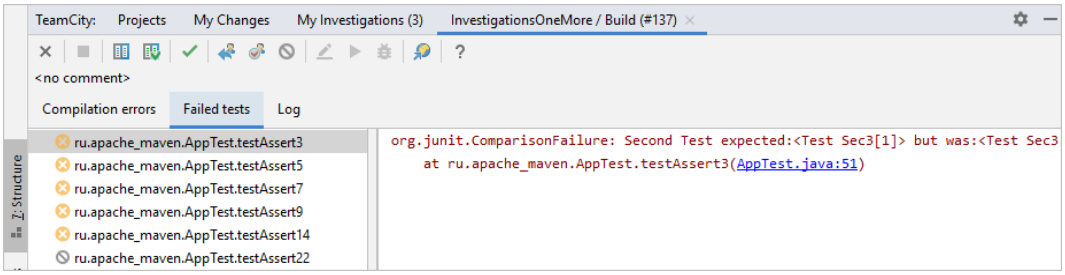


Figure 6.3: The test inspection interface of the TeamCity plugin [60].

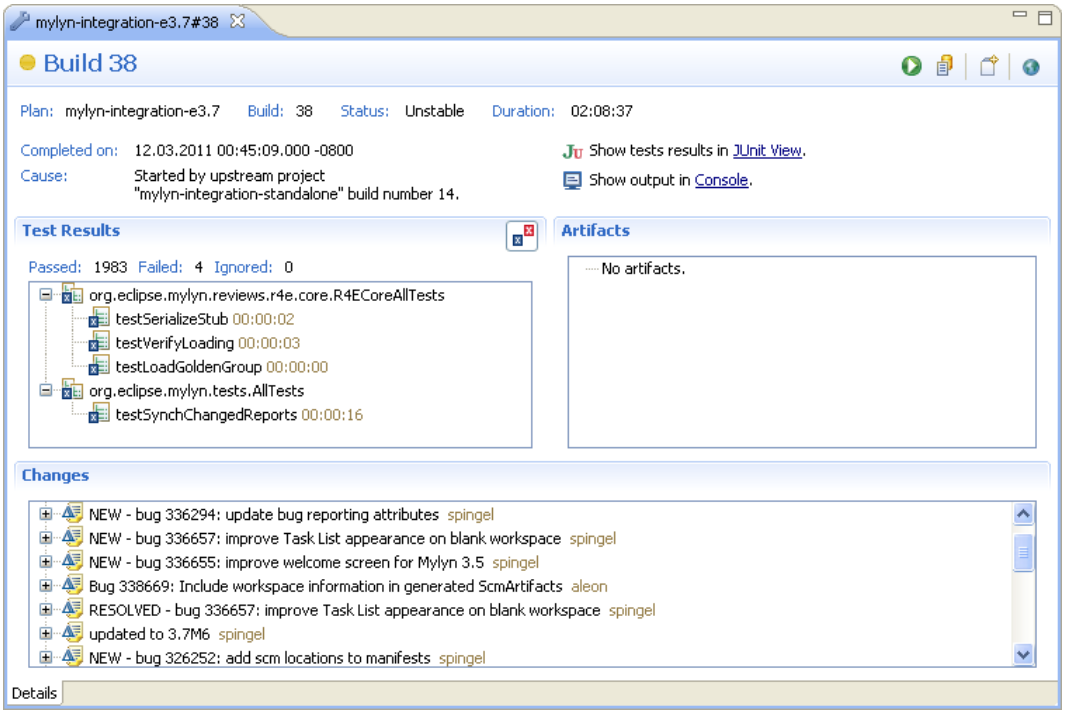


Figure 6.4: The interface of the Hudson/Jenkins Mylyn Builds Connector plugin [31].

Table 6.2: Comparison of IDE plugins with test insights.

Feature	TeamCity [59]	Hudson/Jenkins Mylyn Builds Connector [68]	TESTAXIS
Interactive Stacktraces	✓	Indirectly through the JUnit view	✓
Display of Test Code	Link only	Link only	✓
Code Under Test			✓
Code Changes	✓	✓	✓
Changed Code Under test			✓
Test Health Warnings			✓
Raw Build Log Inspection	✓	✓	
Rerun Test	✓	Indirectly through the JUnit view	Indirectly by opening the test code in the main window
Supported CI Providers	TeamCity	Jenkins	All

rerun a test locally. As shown in Figure 6.4, the Hudson/Jenkins Mylyn Builds Connector plugin shows the test results but also provides insights on execution times and code changes made for this build.

TESTAXIS also gives these insights. It shows an interactive stack trace together with execution details such as the run time. Also, while Hudson/Jenkins Mylyn Builds Connector only shows a list of changes, TESTAXIS incorporates these changes in the (changed) code under test feature that both shows which code fragments were changed and which ones were touched by the test. Furthermore, TESTAXIS also shows and provides easy access to the test code to understand the intent of the test or to spot mistakes in the test itself. Moreover, TESTAXIS is not limited to a specific CI service and can be included in the build process of any CI tool. Finally, TESTAXIS gives interpretation to the data it collects by showing the users test health warnings.

One might expect that showing CI test results in the IDE is not needed because developers can just execute the tests in their IDE that shows a good interface to review and inspect failing tests. However, it turns out that developers actually do not often execute tests in their IDE [6]. Therefore, showing a rich interface with information about the test failures during CI builds right in the IDE could be very helpful. Beller et al. also mention that *"Despite the tool overhead and a possibly slower reaction time, our low results on test executions in the IDE suggest that developers increasingly prefer such more complex setups [in which tests are run on CI servers] to manually executing their tests in the IDE."* and continue by recommending that IDE developers should improve CI integration [6].

Chapter 7

Conclusions and Future Work

Inspecting the results of a failing test in a CI build is a tedious process. It often requires developers to manually inspect and scroll through hundreds to thousands of lines of log output, while running tests inside an IDE offers specific, detailed, and interactive feedback on the test results. TESTAXIS attempts to bring CI test results to the IDE and offer a similar experience to running a test locally. Moreover, it exploits the additional information that is not available or is expensive to collect during local development (such as change or coverage information) to offer additional support while inspecting failures. In this work, we have explored whether that attempt is a successful one and justifies further research into the features offered by TESTAXIS. This chapter summarizes the conclusions based on the questions posed in our research to reflect on whether bringing CI build test results to the IDE with additional context is useful and performance increasing. We also list the main contributions of our work and reflect on opportunities for future research following from this thesis.

7.1 Conclusions

In Chapter 1, we presented six research questions that we posed at the start of this research project. We studied the current state-of-the-art, implemented TESTAXIS, and conducted a user study. Using the knowledge gathered during these research tasks, we now propose answers to the research questions below.

We conducted the user study with 16 participants with varying levels of software engineering experience. Overall, we found that the features of TESTAXIS in all but one assignment reduced the time a developer needed to fix a failing test. The code under test feature had the biggest influence on the failure-fixing time. The participants consider TESTAXIS to be a useful tool and would include it in their workflow.

Due to the design of our study, we can only conclude that certain effects happened *after* introducing TESTAXIS but not necessarily *because of* TESTAXIS. Based on the positive outcomes of the pre-experimental user study, we consider a more in-depth study justified and useful to capture the true effect of using TESTAXIS. In Section 7.3 we reflect on the possibilities for such future studies.

7.1.1 Failure-Fixing Time Performance

In the user study, the 16 participants conducted two assignments for each of the four categories: test failure metadata, test code, code under test, and advanced code under test. For each category, the participants attempted to solve one assignment *without* and one *with* the use of TestAxis. We measured the time it took to find and fix the issue causing the presented test case(s) to fail. Comparing this failure-fixing time between the *without* and *with* variant gives us the performance improvement of using TestAxis.

In all assignments with TestAxis, the participants rarely had to run a failing test locally in the IDE to get more information. Also, the time needed to find out *which tests* failed dropped significantly in the assignments *with* TestAxis compared to the ones *without*. These two general findings contributed in almost all assignments to an improvement of the failure-fixing time when using TestAxis.

Influence of Easy Identification of Failing Tests in CI Builds

RQ1

What is the influence of presenting a test failure in the IDE over a CI build log on the time a developer needs to fix a failing test?

We designed the assignments of the first category to be solvable with only the meta-information (the name and stack trace of the failing test) presented in TestAxis. The first category contains simple cases of test failures where the issue can be spotted in the stack trace alone. We found that the average time participants needed to solve the two assignments decreased by 13.4% when using TestAxis. Also, the participants indicated they spent considerably less time figuring out which tests failed with the help of TestAxis. Moreover, TestAxis reduced the need to run the failing tests locally to get more feedback. We thus conclude that presenting a test failure in the IDE over a CI build log has a positive influence on the time a developer needs to fix a failing test.

Influence of Showing Test Code

RQ2

What is the influence of showing the test code on the time a developer needs to fix a failing test?

The second assignment category featured test failures due to issues in the test code. A lot of participants hit the time limit for the assignments in this category, 10 participants for the assignments *without* TestAxis and 5 for the assignments *with*. This is likely caused by the participants focusing more on the code under test than on the test code. The results show an average improvement of the failure-fixing time of 13.8%. It is hard to say whether the test code display feature of TestAxis helps realize the location of the issue quicker in general. However, we can conclude that the usage of TestAxis in general helps to solve assignments where the issue is in the test code more quickly.

Influence of Showing Changed Code Under Test

RQ3

What is the influence of showing the code under test, where the changed code is highlighted, on the time a developer needs to fix a failing test?

We evaluate the effect of showing (changed) code under test (RQ3) in both the third and fourth category. In the third category, we observe an average improvement in the failure-fixing time of 49.6%. The test cases in this category are straightforward and the issues are in one of the few highlighted code fragments of the changed and covered code shown by TESTAXIS. The suggestions of potential locations of the issue cut down the number of lines of code to inspect drastically compared to inspecting the full code change diff, which is likely the explanation for the significant increase in failure-fixing performance. The participants also indicated that they spent the most time on the code under test while figuring out the cause of the failure.

The fourth category consists of two assignments where the same high-level end-to-end test fails and the participants must find out why. These assignments are more complex than the ones of the third category and require a deeper investigation by the developer. Because the performance results per assignment in this category show different trends, the results are inconclusive due to the experiment design. We found that the more experienced developers that perform the assignment *with* TESTAXIS need more time to solve the assignment than the less experienced developers, contrary to the other assignments. A possible explanation is that the less experienced participants find it easier to adopt new features, such as the changed code under test feature, into their workflows, whereas for more experienced developers it may be difficult to fit a new type of feature in their existing tool belt.

We conclude that the influence of showing the (changed) code under test has a positive influence on the failure-fixing time for simple cases and that our results are not conclusive about more complex cases.

7.1.2 Usefulness

We investigated the usefulness of two specific features (build notifications, RQ4, and test health warnings, RQ5) but also of the tool as a whole (RQ6). The participants of the user study reported their perceived usefulness of these different aspects during the experiment.

Usefulness of Build Notifications

RQ4

To what extent do developers prefer to be actively notified of CI build failures in the IDE over their current approach?

For each assignment the participants conducted with the use of TESTAXIS, they were shown a CI build notification in the IDE. Most participants preferred this notification style over their current approach of becoming aware of build failures, however, a minority disagreed

and thought that the IDE notifications may distract them. The most common approach the participants currently use is checking the build status manually. Others use e-mail notifications or integrations with chat services. The majority of the participants thought that CI build notifications in the IDE would make them aware of build failures quicker. We conclude that the majority of the developers in our study prefer CI build notifications in the IDE.

Usefulness of Test Health Warnings

RQ5

To what extent do developers find it useful to be warned about the health and history of a failing test?

TestAxis shows so-called test health warnings when a test is slower than average, fails often, or is potentially flaky. In the assignments, we showed several of these warnings to the participants. Most of the participants would act on these warnings, however, not everyone was convinced of its usefulness. The warnings on potential flaky and often-failing tests were considered most useful. The warnings on slow tests were considered less beneficial since the participants indicated they usually already know which tests are slow and that some tests are inherently slower than others. Some participants indicated they missed a feature to suppress or acknowledge the warnings which impacted their perceived usefulness. We conclude that the developers in our experiment found the warnings on often-failing and potential flaky tests useful and would act on them, while they consider the slow test warnings less useful.

Usefulness of TestAxis as a CI Build Test Result Inspection Tool

RQ6

To what extent do developers consider a CI build test result inspection IDE plugin useful?

During the assignments, the participants got to work with the TestAxis IDE plugin. TestAxis exceeded or matched their expectations. The participants completed assignments in four categories that align with the three main features of TestAxis: failure details, test code, and code under test. The participants consider all three features useful. The (changed) code under test feature is considered the most useful. The participants think that TestAxis solves a real problem and that it would save them time. Most participants would make TestAxis part of their workflow, one of the participants indicated that they would consider implementing it in their workflow *"as is"* and that they would *"not add much info further as I think its strength lays in the clean and concise overview"*. Overall, the participants find TestAxis useful in helping them understand test failures better and fix them more quickly. One of the participants described their experience as *"I thought it was super useful during the experiments. I much rather preferred using* TestAxis *over the traditional CI logs on GitHub. What* TestAxis *does, in my opinion, is recreate the steps I*

manually take on a GitHub pull request to identify a failing test, and it does so in the IDE so I don't have to switch tabs and interrupt my workflow."

7.2 Contributions

The contributions of this thesis are fourfold:

1. TESTAXIS: A modular software system that collects and analyzes CI build test results, and that visualizes these results with additional context (such as the test code or the code under test) in the IDE. We make the following open-source software artifacts available:
   a) The TESTAXIS backend that collects and analyzes test reports, and exposes them through an API[1].
   b) The TESTAXIS IDE plugin for IntelliJ Platform IDEs that presents build and tests results to the developer[2].
   c) A Gradle plugin that collects and generates coverage reports per executed test[3].
2. An evaluation of the effect of providing CI build test results with additional context (such as the test code or the code under test) in the IDE on the failure-fixing time performance.
3. An evaluation of the perceived usefulness of a CI build test result inspection tool for the IDE.
4. A publicly available dataset containing the data collected during the experiments [12]. Among others, this includes the timing results, anonymized questionnaire results, notes including observed time spent on actions, and analysis scripts.

7.3 Future Work

While our study has shown that TESTAXIS can have a positive effect on the type of cases we presented, more work is needed to confirm its usefulness and potential performance improvement in real-life projects. Based on our research, we make the following recommendations for future work:

Longitudinal Study over Long Periods of Time Due to the design of our study, we could only conclude that there is *an effect* after introducing TESTAXIS but not that this effect is *caused by* TESTAXIS. Our experiment could be repeated as a controlled experiment with a larger sample size to achieve stronger conclusions. However, these conclusions would still only hold for the type of cases presented in the assignments as part of the four categories.

[1] Available at `https://github.com/testaxis/testaxis-backend`
[2] Available at `https://github.com/testaxis/testaxis-intellij-plugin`
[3] Available at `https://github.com/testaxis/coverage-per-test-gradle-plugin`

Therefore, the effect of TESTAXIS may better be captured in a longitudinal study on a real software project over long periods of time. The CI build fixing behavior of one or more software development teams could be monitored in a period without TESTAXIS and a period afterward with TESTAXIS. This would provide more rich information on the actual usefulness and performance improvements in real-life software projects.

TESTAXIS Development A different direction of future work is to extend TESTAXIS. The participants of the experiment indicated which missing features they would like to see in TESTAXIS. Most requested features relate to different styles of navigation that could be implemented. The most requested feature is re-running a failed test locally to confirm that a fix in the code resolved the issue causing the test to fail. These features are relatively easy to implement within the current IDE plugin.

Another requested feature is integration with VCS platforms such as GitHub or GitLab. This integration could make it possible to display the PR name, message, and comments corresponding to a CI build. It would also enable the possibility to comment on lines of code from within TESTAXIS. On top of this integration, a connection to the CI build platform could offer the opportunity to rerun a CI build directly from within TESTAXIS. Both integration types would make TESTAXIS VCS and CI build platform-dependent whereas it is currently not tied to any specific platforms.

Furthermore, TESTAXIS could also exploit the implicit relationships between a test and the code under test more. Since this information is already available, a developer could possibly be helped in their daily work by providing links in the main IDE window between tests and production code.

Another way to extend TESTAXIS is by adding support for more languages and test frameworks. Currently, TESTAXIS supports Java and Kotlin and is only tested with JUnit as a test framework. Most parts of TESTAXIS are language agnostic. Only the logic to find test methods is language-specific and would require to be modified when adding support for other languages. The test framework is not important to the internals of TESTAXIS. However, the format of the incoming test reports is currently the JUnit XML report format. Since this output format is supported by most major test frameworks, it is likely that little to no changes are needed to support other test frameworks than JUnit.

Results of Non-Test Build Phases Currently, TESTAXIS only provides help in cases where a CI build breaks due to failing tests. While this is the case in the majority of the build failures [7, 65, 48], a build could also fail due to compilation errors or static analysis warnings. In such cases, TESTAXIS still shows that the build has failed but is not able to provide any further context. TESTAXIS could be extended to bring more build context from build logs to the IDE. This could help reduce the build-fixing time in more cases than only test failures.

It is likely already beneficial when the information of these other build steps is accessible directly from the IDE. However, the integration in the IDE could also provide other advantages such as direct links to the code base or auto-fixing suggestions.

Test Selection and Prioritization Using the collected test execution histories, recommendations can be made on which tests are most important to run after a certain code change based on which tests are likely to fail. Depending on the test suite size, an execution of the entire suite can take a long time. By suggesting which tests are relevant, a developer can decide to only run a part of the test suite locally. In practice, developers often already use a form of manual test selection by running only the tests directly related to the change during development. Beller et al. found that in 50% of the cases developers only run 1% of the tests in their IDE [6]. However, this might mean that developers miss tests that are not obviously related to the code change.

Test selection and test prioritization (inducing a particular order of tests that reveal issues early in the test suite run) are both well-researched topics. While the classical approaches focus on structural solutions [53, 54, 26], more modern approaches use historical test execution information [37, 17, 41]. The test case execution history is already available in TESTAXIS and can be exploited to implement test selection and prioritization using information collected during CI builds. In future work, it would be interesting to investigate whether this already available information can be used to provide good test selection and prioritization suggestions.

Fix and Failure History of a Test Case The availability of test case execution histories also enables the possibility to show an additional context element for a failing test that shows previous failures and fixes. Because the issue that is causing the test to fail may have already occurred in the past, this may be useful information to show to the developer. TESTAXIS could show a history of test failures with their error messages together with the commit where the failure was introduced and the commit that introduced the fix. The developer could quickly look at this list and check if a similar failure has happened earlier and re-use the earlier fix.

Before this feature would be added to TESTAXIS, the assumption that test failures reoccur in a software project should be verified. TESTAXIS can be used to collect test execution histories in software projects that allow analyzing the presence of such behavior.

Bibliography

[1] J. G. Adair. The Hawthorne Effect: A Reconsideration of the Methodological Artifact. *Journal of Applied Psychology*, page 69(2):334, 1984. URL https://doi.org/10.1037/0021-9010.69.2.334.

[2] Anunay Amar and Peter C. Rigby. Mining historical test logs to predict bugs and localize faults in the test logs. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 140–151, Montreal, Quebec, Canada, May 2019. IEEE Press. URL https://doi.org/10.1109/ICSE.2019.00031.

[3] V. Ambriola and V. Gervasi. Representing structural requirements in software architecture. In R. N. Horspool, editor, *Systems Implementation 2000: IFIP TC2 WG2.4 Working Conference on Systems Implementation 2000: Languages, methods and tools 23–26 February 1998, Berlin, Germany*, IFIP Advances in Information and Communication Technology, pages 114–127. Springer US, Boston, MA, 1998. ISBN 978-0-387-35350-0. URL https://doi.org/10.1007/978-0-387-35350-0_9.

[4] Andrey Artyukhov. GitHub Actions, February 2020. URL https://plugins.jetbrains.com/plugin/13793-github-actions. [Accessed on: 2020-09-22].

[5] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, October 1999. ISSN 1558-0814. URL http://doi.org/10.1109/2.796139.

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, August 2015. Association for Computing Machinery. ISBN 978-1-4503-3675-8. URL http://doi.org/10.1145/2786805.2786843.

[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367, May 2017. URL https://doi.org/10.1109/MSR.2017.62.

[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450, Buenos Aires, Argentina, May 2017. IEEE. ISBN 978-1-5386-1544-7. URL https://doi.org/10.1109/MSR.2017.24.

[9] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, March 2019. ISSN 1939-3520. URL http://doi.org/10.1109/TSE.2017.2776152.

[10] David Boissier, Yuri Novitsky, and Michael Suhr. Jenkins Control Plugin, May 2011. URL https://plugins.jetbrains.com/plugin/6110-jenkins-control-plugin. [Accessed on: 2020-09-22].

[11] Grady Booch. *Object-Oriented Analysis and Design With Applications, 2nd edition*. Addison-Wesley Professional, 2 edition, 1994. ISBN 978-0-8053-5340-2.

[12] Casper Boone. TestAxis Replication Package. *4TU.ResearchData*, June 2021. URL https://doi.org/10.4121/14748894.

[13] Philipp Bouillon, Jens Krinke, Nils Meyer, and Friedrich Steimann. EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors. In Giulio Concas, Ernesto Damiani, Marco Scotto, and Giancarlo Succi, editors, *Agile Processes in Software Engineering and Extreme Programming*, Lecture Notes in Computer Science, pages 101–104, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-73101-6. URL https://doi.org/10.1007/978-3-540-73101-6_14.

[14] Carolin E. Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. LogChunks: A Data Set for Build Log Analysis. In *17th International Conference on Mining Software Repositories (MSR '20)*, page 5, Seoul, Republic of Korea, October 2020. ACM, New York, NY, USA. URL https://doi.org/10.1145/3379597.3387485.

[15] Martin Brandtner, Emanuel Giger, and Harald Gall. SQA-Mashup: A mashup framework for continuous integration. *Information and Software Technology*, 65:97–113, September 2015. ISSN 0950-5849. URL https://doi.org/10.1016/j.infsof.2014.10.004.

[16] Donald Thomas Campbell and Julian Cecil Stanley. *Experimental and quasi-experimental designs for research*. Houghton Mifflin Company, Boston, 2nd edition edition, 1963. ISBN 978-0-395-30787-8. OCLC: 247359300.

[17] Younghwan Cho, Jeongho Kim, and Eunseok Lee. History-Based Test Case Prioritization for Failure Information. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 385–388, December 2016. URL https://doi.org/10.1109/APSEC.2016.066. ISSN: 1530-1362.

[18] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, USA, 1995. ISBN 978-0-02-874048-5.

[19] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. ReAssert: Suggesting Repairs for Broken Unit Tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 433–444, November 2009. URL https://doi.org/10.1109/ASE.2009.17. ISSN: 1938-4300.

[20] John Downs, Beryl Plimmer, and John G. Hosking. Ambient awareness of build status in collocated software teams. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 507–517, June 2012. URL https://doi.org/10.1109/ICSE.2012.6227165. ISSN: 1558-1225.

[21] Florian Dreier. Obtaining Coverage per Test Case. Master's Thesis, November 2017. URL https://www.cqse.eu/fileadmin/content/news/publications/2017-obtaining-coverage-per-test-case.pdf.

[22] Lamyaa Eloussi. *Determining flaky tests from test failures*. PhD thesis, University of Illinois at Urbana-Champaign, 2015. URL http://hdl.handle.net/2142/78543.

[23] Jesse Glick. JUnit Plugin for Jenkins, September 2010. URL https://plugins.jenkins.io/junit. [Accessed on: 2020-09-15].

[24] D. Goodman and M. Elbaz. "It's Not the Pants, it's the People in the Pants" Learnings from the Gap Agile Transformation What Worked, How We Did it, and What Still Puzzles Us. In *Agile 2008 Conference*, pages 112–115, August 2008. URL https://doi.org/10.1109/Agile.2008.87.

[25] Michael Gumowski. Sonar/Jacoco: Per Test coverage - need help, January 2021. URL https://community.sonarsource.com/t/sonar-jacoco-per-test-coverage-need-help/37515/3. [Accessed on: 2021-05-21].

[26] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for Java software. *ACM SIGPLAN Notices*, 36(11):312–326, October 2001. ISSN 0362-1340. URL http://doi.org/10.1145/504311.504305.

[27] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 426–437, New York, NY, USA, August 2016. Association for Computing Machinery. ISBN 978-1-4503-3845-5. URL http://doi.org/10.1145/2970276.2970358.

[28] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings*

of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pages 197–207, New York, NY, USA, August 2017. Association for Computing Machinery. ISBN 978-1-4503-5105-8. URL http://doi.org/10.1145/3106237.3106270.

[29] Victor Hurdugaci and Andy Zaidman. Aiding Software Developers to Maintain Developer Tests. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 11–20, March 2012. URL https://doi.org/10.1109/CSMR.2012.12. ISSN: 1534-5351.

[30] IBM. IBM Knowledge Center: JUnit XML format, October 2014. URL www.ibm.com/support/knowledgecenter/ssq2r2_9.1.1/com.ibm.rsar.analysis.codereview.cobol.doc/topics/cac_useresults_junit.html. [Accessed on: 2020-09-15].

[31] Mik Kersten. Mylyn 3.5: New & Noteworthy, March 2011. URL https://www.eclipse.org/mylyn/new/showVersion.php?version=new-3.5.html. [Accessed on: 2020-09-22].

[32] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why Do Automated Builds Break? An Empirical Study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50, September 2014. URL https://doi.org/10.1109/ICSME.2014.26. ISSN: 1063-6773.

[33] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22 140:55–55, 1932.

[34] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, November 2014. Association for Computing Machinery. ISBN 978-1-4503-3056-5. URL http://doi.org/10.1145/2635868.2635920.

[35] Henry B. Mann and Donald R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947. ISBN: 0003-4851 Publisher: JSTOR.

[36] Diego Marcher. Github Tools, October 2019. URL https://plugins.jetbrains.com/plugin/13366-github-tools. [Accessed on: 2020-09-22].

[37] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *2013 IEEE International Conference on Software Maintenance*, pages 540–543, September 2013. URL https://doi.org/10.1109/ICSM.2013.91. ISSN: 1063-6773.

[38] Rebecca L. Mauldin. *Foundations of Social Work Research*. Mavs Open Press, 2020. ISBN 978-0-9898878-8-5. URL https://rc.library.uta.edu/uta-ir/handle/10106/29081.

[39] Ade Miller. A Hundred Days of Continuous Integration. In *Agile 2008 Conference*, pages 289–293, August 2008. URL https://doi.org/10.1109/Agile.2008.8.

[40] Charity J. Morgan. Use of proper statistical techniques for research studies with small samples. *American Journal of Physiology. Lung Cellular and Molecular Physiology*, 313(5):L873–L877, November 2017. ISSN 1522-1504. URL https://doi.org/10.1152/ajplung.00238.2017.

[41] Tadahiro Noguchi, Hironori Washizaki, Yoshiaki Fukazawa, Atsutoshi Sato, and Kenichiro Ota. History-Based Test Case Prioritization for Black Box Testing Using Ant Colony Optimization. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–2, April 2015. URL https://doi.org/10.1109/ICST.2015.7102622. ISSN: 2159-4848.

[42] OpenClover. Unit Test Results and Per-Test Coverage. URL https://openclover.org/doc/manual/4.4.0/ant--test-results-and-per-test-coverage.html. [Accessed on: 2021-05-21].

[43] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 345–355, New York, NY, USA, May 2000. Association for Computing Machinery. ISBN 978-1-58113-253-3. URL http://doi.org/10.1145/336512.336586.

[44] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. Creating a shared understanding of testing culture on a social coding site. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 112–121, May 2013. URL https://doi.org/10.1109/ICSE.2013.6606557. ISSN: 1558-1225.

[45] Achilleas Pipinellis. GitLab: Unit test reports, August 2018. URL https://docs.gitlab.com/ee/ci/unit_test_reports.html. [Accessed on: 2020-09-15].

[46] Paul C. Price, Rajiv S. Jhangiani, I-Chant A. Chiang, Carrie Cuttler, Dana C. Leighton, and Carrie Cuttler. *Research Methods in Psychology*. 3rd edition, 2017. URL https://opentext.wsu.edu/carriecuttler/.

[47] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168, February 2014. ISSN 0164-1212. URL https://doi.org/10.1016/j.jss.2013.10.019.

[48] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 345–355, May 2017. URL https://doi.org/10.1109/MSR.2017.54.

[49] Naomi B. Robbins and Richard Heiberger. Plotting Likert and Other Rating Scales. *Proceedings of the 2011 Joint Statistical Meeting*, 2011.

[50] Suzanne Robertson and James Robertson. *Mastering the Requirements Process: Getting Requirements Right*. Addison-Wesley Professional, 3rd edition, 2012. ISBN 978-0-321-81574-3.

[51] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M. Germán, and Jesus M. Gonzalez-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25(2):1294–1340, March 2020. ISSN 1573-7616. URL https://doi.org/10.1007/s10664-019-09781-y.

[52] Bart Van Rompaey and Serge Demeyer. Establishing Traceability Links between Unit Test Cases and Units under Test. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 209–218, March 2009. URL https://doi.org/10.1109/CSMR.2009.39. ISSN: 1534-5351.

[53] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 169–184, New York, NY, USA, August 1994. Association for Computing Machinery. ISBN 978-0-89791-683-7. URL http://doi.org/10.1145/186258.187171.

[54] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, 2000. ISSN 1099-1689. URL https://doi.org/10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E.

[55] Fabio Sani and John Todman. Appendix 1: Statistical Tables. In *Experimental Design and Statistics for Psychology*, pages 183–196. John Wiley & Sons, Ltd, 2006. ISBN 978-0-470-77612-4. URL https://doi.org/10.1002/9780470776124.

[56] M. K. Slack and J. R. Draugalis. Establishing the internal and external validity of experimental studies. *American journal of health-system pharmacy: AJHP: official journal of the American Society of Health-System Pharmacists*, 58(22):2173–2181; quiz 2182–2183, November 2001. ISSN 1079-2082.

[57] JetBrains s.r.o. Run with coverage. URL https://www.jetbrains.com/help/idea/2021.1/running-test-with-coverage.html. [Accessed on: 2021-05-21].

[58] JetBrains s.r.o. TeamCity: the Hassle-Free CI and CD Server, October 2006. URL https://www.jetbrains.com/teamcity/. [Accessed on: 2020-10-05].

[59] JetBrains s.r.o. TeamCity IntelliJ Plugin, December 2007. URL https://plugins.jetbrains.com/plugin/1820-teamcity. [Accessed on: 2020-09-22].

[60] JetBrains s.r.o. Re-Running Failed Tests, 2020. URL https://www.jetbrains.com/help/teamcity/ij-addin/tc-rerunningfailedtests.html. [Accessed on: 2020-09-22].

[61] Simon Stratmann. IntelliJ GitLab Pipeline Viewer, February 2020. URL https://plugins.jetbrains.com/plugin/13799-intellij-gitlab-pipeline-viewer. [Accessed on: 2020-09-22].

[62] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, January 2014. ISSN 0164-1212. URL https://doi.org/10.1016/j.jss.2013.08.032.

[63] G. Tassey. *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, 2002.

[64] Fabian Trautsch, Steffen Herbold, and Jens Grabowski. Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects. *Journal of Systems and Software*, 159:110421, January 2020. ISSN 0164-1212. URL https://doi.org/10.1016/j.jss.2019.110421.

[65] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193, September 2017. URL https://doi.org/10.1109/ICSME.2017.67.

[66] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. Un-break My Build: Assisting Developers with Build Repair Hints. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 41–4110, May 2018. ISSN: 2643-7171.

[67] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering*, 25(3):2218–2257, May 2020. ISSN 1573-7616. URL https://doi.org/10.1007/s10664-019-09765-y.

[68] Paul Verest. Hudson/Jenkins Mylyn Builds Connector, October 2013. URL https://marketplace.eclipse.org/content/hudsonjenkins-mylyn-builds-connector. [Accessed on: 2020-09-22].

[69] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining Software Repositories to Study Co-Evolution of Production Test Code. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229, April 2008. URL https://doi.org/10.1109/ICST.2008.47. ISSN: 2159-4848.

[70] Sai Zhang, Cheng Zhang, and Michael D. Ernst. Automated documentation inference to explain failed tests. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 63–72, November 2011. URL https://doi.org/10.1109/ASE.2011.6100145. ISSN: 1938-4300.

[71] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, May 2005. Association for Computing Machinery. ISBN 978-1-59593-123-8. URL http://doi.org/10.1145/1083142.1083147.

Appendix A

Glossary

In this appendix, we give an overview of frequently used terms and abbreviations.

Build Log: A build log contains the console output generated during the build process of a software project. Depending on the build configuration, this may, for example, include the output of a compiler, static analysis tools, and a test runner.

CI Build: A typical CI build comprises building the application to ensure the code compiles, executing the tests to check whether the application works as expected, and running static analysis tools to safeguard the quality of the codebase and detect potential issues.

Code Under Test: The lines of production code that a test case targets.

Continuous Integration (CI): A software engineering practice where functional and quality attributes of a software application are verified after a change to the source code of the application. The goal is to detect potential issues with the integration of the new change in the main codebase as soon as possible by providing early feedback before a change makes it to production.

Failure-Fixing Time: The time it takes a developer to fix a failing test case.

Integrated Development Environment (IDE): An IDE comprises a set of tools to assist in software development. The main part of an IDE is the code editor. An IDE has deep knowledge of the programming language and provides features such as syntax highlighting, code completion, and application debugging.

IDE Plugin: An extension of the IDE that provides additional functionality over the built-in tools to the developer.

Test Case: A set of actions that should bring a software application in a certain state followed by a set of assertions to verify that the software application is indeed in the correct state and thus working as expected.

Appendix B

Questionnaires

In this appendix, we list the questions of the three types of questionnaires used in the experiment. Section B.1 shows the pretest questionnaire that participants fill out at the start of the experiment. In Section B.2, we show the post-assignment questionnaire that we conduct after each assignment. At the end of the experiment, the participants fill out the posttest questionnaire which can be found in Section B.3.

B.1 Pretest Questionnaire

B.1.1 Informed Consent

You are being invited to participate in a research study titled "TestAxis: Better Insights in Failing Tests During CI Builds". This study is being done Casper Boone from the TU Delft.

The purpose of this research study is to discover ways a developer can fix failing builds faster without having to dive into build logs, and will take you approximately 90 minutes to complete. The data will be used to evaluate the effectiveness of the TestAxis IDE plugin that visualizes CI builds.

Your participation in this study is entirely voluntary and you can withdraw at any time. You are free to omit any question.

We believe there are no known risks associated with this research study; however, as with any online related activity the risk of a breach is always possible. To the best of our ability your answers in this study will remain confidential. We will minimize any risks by storing all data on secure storage within the EU. The access to the data is limited to the researcher and their supervisors. Your personal information will be removed from the data after the research has been completed (mid 2021). The results of the experiments will be published in a thesis report and possible publications and will be anonymized.

Contact Details

The contact details below may be use to file a complaint or to contact any of the involved parties.

Researcher: Casper Boone c.c.boone@student.tudelft.nl
Research Supervisor: Prof.dr. A.E. Zaidman a.e.zaidman@tudelft.nl
Data Protection Officer: privacy-tud@tudelft.nl
Institution: Delft University of Technology info@tudelft.nl

Please tick the appropriate boxes below.

Taking part in the study

1. I have read and understood the study information above, or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction.
   [multiple choice]: yes/no

2. I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.
   [multiple choice]: yes/no

3. I understand that taking part in the study involves a survey questionnaire and written notes during the experiment.
   [multiple choice]: yes/no

Use of the information in the study

1. I understand that information I provide will be used for a thesis report and possible publications
   [multiple choice]: yes/no

2. I understand that personal information collected about me that can identify me, such as my name, will not be shared beyond the study team.
   [multiple choice]: yes/no

3. I agree that my information provided during the experiment can be quoted anonymously in research outputs
   [multiple choice]: yes/no

Future use and reuse of the information by others

1. I give permission for the anonymized survey results that I provide and the anonymous notes taken during the experiment to be archived in the research output and the 4TU Research Data repository so it can be used for future research and learning.
   [multiple choice]: yes/no

Signing by the participant

1. I acknowledge that I have completely read and fully understand this consent form. I hereby sign this consent agreement.
   [multiple choice]: yes/no

2. I sign this consent agreement with the following name:
   [open]

3. I sign this consent agreement on the following day:
   [open]

Signing by the researcher

1. I have accurately read out the information sheet to the potential participant and, to the best of my ability, ensured that the participant understands to what they are freely consenting.
   [multiple choice]: yes/no

2. The researcher signs this consent agreement with the following name:
   [open]

3. The researcher signs this consent agreement on the following day:
   [open]

B.1.2 Personal Background

1. What is your current or highest level of education?
   [multiple choice]: Primary School, Secondary School, Associate (MBO), Bachelor (HBO/college), Bachelor (WO/university), Master (HBO/college), Master (WO/university), PhD

2. What is your (main) professional occupation?
   Please use the ”other” option if the given options do not accurately describe your professional occupation.
   [multiple choice]: Student, PhD Student, Software Engineer, Consultant, Researcher, Other

3. How many years of experience in programming do you have?
   [open]

B.1.3 Experience

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

Development Experience

1. I consider myself to be experienced in developing software applications
   [Likert 1-5]

2. I develop software applications professionally
   [Likert 1-5]

3. I consider myself to be experienced in developing Java applications
   [Likert 1-5]

4. I consider myself to be experienced using IntelliJ (or another JetBrains IDE)
   [Likert 1-5]

5. I have used JPacman before and am still aware of the project structure
   [Likert 1-5]

6. I consider myself to be experienced in using Gradle
   [Likert 1-5]

Testing Experience

1. I consider myself to be experienced in software testing
   [Likert 1-5]

2. I consider myself to be experienced using mocks in my tests
   [Likert 1-5]

3. I consider myself to be experienced in using Mockito
   [Likert 1-5]

4. I consider myself to be experienced in running tests with coverage collection enabled and inspecting the results
   [Likert 1-5]

CI Experience

1. I consider myself to be experienced in using GitHub Pull Requests
   [Likert 1-5]

2. I consider myself to be experienced in using continuous integration build tools (like Travis CI, GitHub Actions or Jenkins) and inspecting the output logs when a build fails
   [Likert 1-5]

3. I consider myself to be experienced in using GitHub Actions
   [Likert 1-5]

B.1.4 Attitude towards software testing and continuous integration

Below we use the terms unit and integration/system tests. We consider unit tests to be testing a single component in isolation, and integration/system tests to be testing the integration of several components or a system feature in its entirety.

1. What are the steps you take when a build fails because of a failing tests between encountering the build failure and committing the fix?
   [open]

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. While developing, I run my tests inside the IDE
   [Likert 1-5]
2. I primarily use e-mail notifications to become aware of a build failure
   [Likert 1-5]
3. I primarily check the PR status checks or CI tool to become aware of a build failure
   [Likert 1-5]
4. My current notification approach makes me aware of build failures quickly
   [Likert 1-5]
5. Are there other ways you are notified of failing builds? (optional)
   [open]
6. When a build fails, I often (re)run the entire test suite locally to find out which test is failing
   [Likert 1-5]
7. When a build fails due to a failing test, I inspect the build log the find the failing test and run this specific test locally
   [Likert 1-5]
8. When employing software testing, I think writing more integration or system/end-to-end tests is more effective than writing unit tests
   [Likert 1-5]
9. CI build logs by themselves give me enough information to point to the issue causing a failing test
   [Likert 1-5]
10. I like the way the cause of failing builds is presented in CI build logs
    [Likert 1-5]
11. Test failures in a CI build are as easy to solve as ”local” failures.
    [Likert 1-5]

12. I care about improving the speed of my test suite.
    [Likert 1-5]

13. Flaky tests are difficult to recognize.
    [Likert 1-5]

14. Tests that fail often are as meaningful as tests that fail almost never.
    [Likert 1-5]

B.1.5 Expectations of a CI build test execution visualization tool

Please read the following high-level introduction of a tool for software developers:

"A visualization tool for the results of tests executed during a CI build should improve the build fixing experience of the developer. The tool should show test failures in a more approachable manner than build logs.

Furthermore, the tool should show information relevant to the test failure such as an interactive stack trace, the test code and the modified code under test, in addition to information usually available in a build log.

This should help fix failing tests quicker and make the fixing experience closer to a locally failing test. Moreover, the tool should actively notify developers of failing builds."

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. Such a tool would save me time
   [Likert 1-5]

2. Such a tool by itself would be able to provide me with enough information to fix a failing build
   [Likert 1-5]

3. Such a tool would become a part of my workflow
   [Likert 1-5]

4. Such a tool would provide benefits over inspecting CI build logs
   [Likert 1-5]

5. Such a tool would make sense to be integrated in an IDE
   [Likert 1-5]

6. Such a tool would solve a real problem
   [Likert 1-5]

B.2 Post-Assignment Questionnaire

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. This assignment was too difficult to solve in the given amount of time
   [Likert 1-5]
2. I spent most time understanding the code/codebase
   [Likert 1-5]
3. I spent most time finding out which test(s) failed
   [Likert 1-5]
4. I spent most time figuring out the cause of the test failure
   [Likert 1-5]
5. While figuring out the cause of the test failure, I spent most time on looking at the metadata of the failure (test name, stack trace, etc)
   [Likert 1-5]
6. While figuring out the cause of the test failure, I spent most time on looking at the test code
   [Likert 1-5]
7. While figuring out the cause of the test failure, I spent most time on looking at the code under test
   [Likert 1-5]
8. To get more information on the cause of the test, I had to run the failing test(s) locally
   [Likert 1-5]
9. Do you have any remarks? (optional)
   [open]

B.3 Posttest Questionnaire

Verification

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. The assignments were challenging and/or interesting to complete
   [Likert 1-5]
2. The time limit made me feel pressured
   [Likert 1-5]

3. I would have solved the assignments easier/quicker outside this (experiment) environment
   [Likert 1-5]

4. During the assignments without TestAxis, I used the same tactics to solve the assignments as I would have used outside of this experiment
   [Likert 1-5]

5. My abilities to solve failures were influenced by my lack of pre-existent knowledge about the codebase
   [Likert 1-5]

6. The assignments represented test failures similar to the ones I encounter when working on software projects
   [Likert 1-5]

7. Enough guidance was provided to solve the assignments
   [Likert 1-5]

Usefulness of Informational Elements

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. I feel like the information given by TestAxis helps me understand a test failure better and fix it quicker
   [Likert 1-5]

2. I consider the details tab of a test case execution in TestAxis to be useful and to give relevant information to find the issue of failing test
   [Likert 1-5]

3. I consider the test code tab of a test case execution in TestAxis to be useful and to give relevant information to find the issue of failing test
   [Likert 1-5]

4. I consider the code under test tab of a test case execution in TestAxis to be useful and to give relevant information to find the issue of failing test
   [Likert 1-5]

5. The code under test feature would be less useful if changed files were not indicated and changes were not highlighted
   [Likert 1-5]

Build Notifications

During the experiment, new builds were presented to you through notifications in the IDE. We call these notifications ”build notifications”.

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. The build notifications were too intrusive
   [Likert 1-5]
2. I think build notifications in the IDE would make me aware of a build failure earlier
   [Likert 1-5]
3. The build notifications provided enough information to recognize the triggering code change
   [Likert 1-5]
4. I prefer my current approach of becoming aware of build failures over build notifications in the IDE
   [Likert 1-5]

Test Health Warnings

During the experiment, test health warnings were presented to you for some of the test cases.

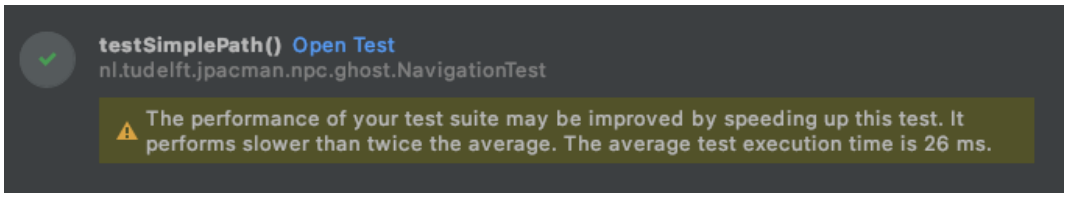
There are three possible warnings that can be shown by TestAxis:

1) This test did not fail due to any code changes. The test failure may be caused by flakiness or an extrinsic issue such as a configuration change.

2) The performance of your test suite may be improved by speeding up this test. It performs slower than twice the average. The average test execution time is X.

3) This test is failing often (X times in the last 50 builds). This may be an indication that your test is too tightly coupled to your production code or that the test may be flaky.

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).



1. I noticed the test health warnings (indicating potential flakiness, long run times or often failing tests) while performing the tasks
   [Likert 1-5]

2. If I was working on my own software project, I would act on the health warnings
   [Likert 1-5]

3. I consider the test health warning on a potential flaky test (1) to be useful
   [Likert 1-5]

4. I consider the test health warning on a slow test (2) to be useful
   [Likert 1-5]

5. I consider the test health warning on a often failing test (3) to be useful
   [Likert 1-5]

TestAxis IDE Plugin User Experience

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. I thought TestAxis was easy to use
   [Likert 1-5]

2. I would imagine that most people would learn to use TestAxis very quickly
   [Likert 1-5]

3. The functionality of TestAxis would be better as a standalone (web) tool which is not integrated in the IDE
   [Likert 1-5]

4. TestAxis provides a similar experience to inspecting IntelliJ test run results
   [Likert 1-5]

5. TestAxis integrates well with IntelliJ
   [Likert 1-5]

6. Bugs in TestAxis influenced my user experience
   [Likert 1-5]

7. I would have preferred to see only the code of the relevant test method and not the surrounding test class in the ”Test Code” tab
   [Likert 1-5]

8. The ordering of covered files, where modified files are highlighted and shown first, was useful to me
   [Likert 1-5]

9. It was clear to me what the different code highlighting colors in the ”Code under Test” tab meant
   [Likert 1-5]

Expectations of a CI build test execution visualization tool

In the introductory questions, we asked you a few questions about CI build test execution visualization tools. We now ask you the same questions for TestAxis specifically.

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. TestAxis provides benefits over inspecting CI build logs
   [Likert 1-5]

2. TestAxis makes sense to be integrated in an IDE
   [Likert 1-5]

3. Using TestAxis will save me time
   [Likert 1-5]

4. TestAxis would become a part of my workflow
   [Likert 1-5]

5. TestAxis solves a real problem
   [Likert 1-5]

6. TestAxis by itself would be able to provide me with enough information to fix a failing build
   [Likert 1-5]

Comments

1. Is there any information or feature missing in TestAxis that would have improved your ability to fix a failing test?
   [open]

2. Do you have any comments or remarks on TestAxis in general?
   [open]

3. Do you have any comments or remarks on the experiment?
   [open]

4. Do you have any other comments or remarks?
   [open]

The Experiment

Below, a number of statements is shown. Please rate each statement on a scale from 1 (totally disagree) to 5 (totally agree).

1. The quality of the assignments was high
   [Likert 1-5]
2. The quality of the questionnaire was high
   [Likert 1-5]
3. The explanations of TestAxis and JPacman were clear
   [Likert 1-5]
4. The project choice of JPacman allowed for interesting cases that were suitable to answer the questions
   [Likert 1-5]
5. There was enough opportunity to give feedback
   [Likert 1-5]
6. I enjoyed performing the experiment
   [Likert 1-5]

Appendix C

Assignments

In this appendix, we list the order in which the participants conducted the assignments (Section C.1) and show the assignment descriptions of all eight assignments.

C.1 Assignment Ordering

The table below shows the order in which the participants conducted the assignments for each participant.

	Participant					
Position	1	2	3	4	5	6
1	4b-without	2a-with	2a-without	1b-without	3b-without	2a-with
2	2a-with	1a-without	3a-with	4a-with	1b-with	2b-without
3	2b-without	1b-with	3b-without	3a-with	2b-with	4a-without
4	4a-with	2b-without	2b-with	2b-without	4a-without	1a-with
5	3b-with	3b-with	4a-with	4b-without	4b-with	3a-without
6	1a-without	3a-without	1a-without	1a-with	2a-without	4b-with
7	1b-with	4b-without	1b-with	2a-with	1a-without	1b-without
8	3a-without	4a-with	4b-without	3b-without	3a-with	3b-with
Position	7	8	9	10	11	12
1	4a-without	2a-without	3a-without	2a-without	2b-without	2a-with
2	1b-with	1a-with	3b-with	3a-with	2a-with	3b-without
3	3a-without	3b-with	4b-with	4b-with	1a-with	3a-with
4	4b-with	3a-without	1b-without	4a-without	4a-without	2b-without
5	2b-with	1b-without	1a-with	1b-without	4b-with	4a-without
6	2a-without	4b-with	2a-without	2b-with	3a-without	1b-with
7	1a-without	4a-without	2b-with	3b-without	3b-with	1a-without
8	3b-with	2b-with	4a-without	1a-with	1b-without	4b-with
Position	13	14	15	16		
1	2a-without	2b-without	3b-without	1a-with		
2	4a-with	2a-with	3a-with	4b-without		
3	3a-with	1b-with	1a-with	3a-without		
4	3b-without	4b-without	2a-without	2a-with		
5	4b-without	3a-with	2b-with	4a-with		
6	1b-with	1a-without	4b-without	1b-without		
7	1a-without	4a-with	4a-with	2b-without		
8	2b-with	3b-without	1b-without	3b-with		

C.2 Assignment Descriptions

The following pages show the assignment descriptions that the participants of the user study read before they performed the assignments.

TestAxis EXPERIMENT

Task 1a

The (intended) change: A resource file is given a new name.

The developer description: Rename sprite test resource for consistency

Previously, the resource was called `64x64white.png`. This is inconsistent with other project resources. Therefore, I propose to rename the resource to `white_64x64.png`.

When you perform this task without TestAxis, you can find the Pull Request with the failing build here: https://github.com/testaxis/jpacman/pull/3

TestAxis EXPERIMENT

Task 1b

The (intended) change: A new map element is added and put on the map

The developer description: Add brick wall map element

This PR adds a new brick wall element to the map. I have updated the map to include a few brick walls (with ! as the character).

When you perform this task without TestAxis, you can find the Pull Request with the failing build here: https://github.com/testaxis/jpacman/pull/4

TestAxis EXPERIMENT

Task 2a

The (intended) change: Adding a new method with tests

The developer description: Add feature to retrieve a square by a relative position
Add `getSquareAtByDelta` *method to retrieve a square (that is present on the board) relative to another square by providing a delta x and y steps. This is a more general version of the* `getSquareAt` *method that is already present and allows finding indirectly connected squares.*

When you perform this task without TestAxis, you can find the Pull Request with the failing build here: https://github.com/testaxis/jpacman/pull/5/

TestAxis EXPERIMENT

Task 2b

The (intended) change: Adding a new method with tests

The developer description: Add feature to determine the winner of the game
When the new method `getWinningPlayer` *is called, at the end of the game, the player with the highest score that is still alive will be returned.*

When you perform this task without TestAxis, you can find the Pull Request with the failing build here: https://github.com/testaxis/jpacman/pull/6/

TestAxis EXPERIMENT

Task 3a

The (intended) change: Adding a new method with tests

The developer description: Add new pellet type with random score
This PR allows for adding pellets with a random score between 20 and 30 (instead of the default fixed score of 10). The map character is `$`*.*

When you perform this task without TestAxis, you can find the Pull Request with the failing build here: https://github.com/testaxis/jpacman/pull/7/

TestAxis EXPERIMENT

Task 3b

The (intended) change: Adding a new method as a refactor

The developer description: Add `opposite` method to `Direction`
When Clyde runs away from the player, it looks up the opposite of the direction it would otherwise have gone in. To find the opposite, Clyde uses a hash map that maps each direction to its opposite. This PR replaces that hash map with a dedicated function in the `Direction` *class.*

When you perform this task without TestAxis, you can find the Pull Request with the failing build here: https://github.com/testaxis/jpacman/pull/8/

TestAxis EXPERIMENT

Task 4a

The (intended) change: Add new power-ups that are triggered after the number of times a certain event happened crosses a lower threshold. Sometimes there is also an upper threshold.

Your task: The task is to find out which power-up is causing the test to fail (in other words: which power-up is activated during the test execution) and change the threshold for the power-up to make sure it does not influence the failing test. The new threshold does not have to be realistic, as long as it effectively disables the power-up for the failing test, it is fine.

The developer description: Add power-ups to the game
This PR adds several power-ups to the game:

- *A safety "shield" when the game has started so ghosts cannot kill the player*
- *A bonus for taking more steps*
- *A power-up in which earned points get doubled*

When you perform this task without TestAxis, you can find the Pull Request with the failing build here: https://github.com/testaxis/jpacman/pull/9/

TestAxis EXPERIMENT

Task 4b

The (intended) change: Add new power-ups that are triggered after the number of times a certain event happened crosses a lower threshold. Sometimes there is also an upper threshold.

Your task: The task is to find out which power-up is causing the test to fail (in other words: which power-up is activated during the test execution) and change the threshold for the power-up to make sure it does not influence the failing test. The new threshold does not have to be realistic, as long as it effectively disables the power-up for the failing test, it is fine.

The developer description: Add power-ups to the game
This PR adds several power-ups to the game:

- *A power-up in which ghosts freeze*
- *A power-up in which walls become accessible*
- *A power-up that doubles the moving speed*

When you perform this task without TestAxis, you can find the Pull Request with the failing build here: https://github.com/testaxis/jpacman/pull/10/

Appendix D

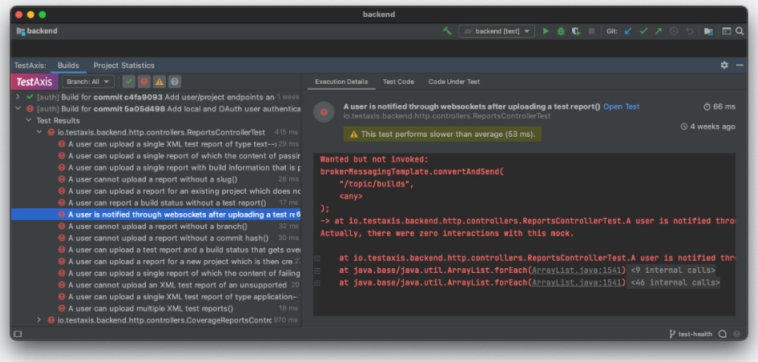
Experiment Participants Information Website

The follow pages shows the website with information for potential epxeriment participants.

TestAxis EXPERIMENT

A CI build fails, what do you do? — To identify the issue causing the failure, you'd probably open the build log only to find **hundreds to thousands of lines of logs** you need to scroll through to get to the cause. The issue is likely a failing test since that is the most important reason why builds fail. However, even when you finally found which test is failing, the build log **only shows you the name and error message of the test**. This is not always helpful: a message like "Failed asserting that 24 is equal to 15" will not bring you any closer to fixing your failing system test where the issue can come from anywhere in the codebase.

TestAxis improves the experience of fixing failing CI builds: it gives you an **overview of the failing tests** without having to look at logs and it puts tests **in the context of the code change**. The TestAxis IDE plugin **alerts** you of a build failure and gives you the ability to inspect the results, so you never have to leave your IDE when a build fails. For each failing test, it shows you **the stack trace** (similar to locally running your tests), the **test code**, and the code under test. The **code under test** feature shows you the covered production code which was modified in the commits leading up to the build failure: so, a likely source of the issue! Once you have found the issue, you can **apply the fix** directly from within the IDE plugin. Are you curious to check out TestAxis and help us make it better? Join our experiments!



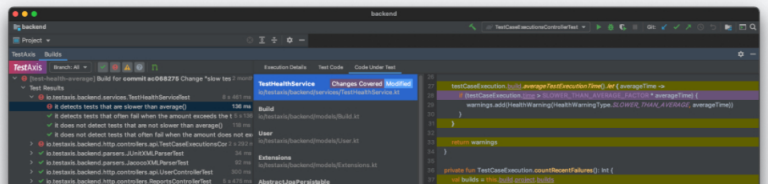
The Experiment

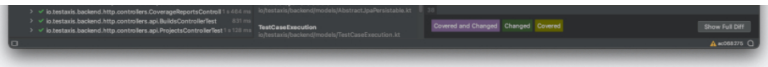
We need your help!

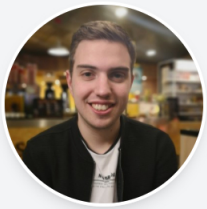
To evaluate the working and effectiveness of TestAxis, we'd kindly like to ask you to participate in our experiment. During the experiment we ask you to solve a number of tasks with and without TestAxis. The tasks involve fixing failing tests in a simple Java project. We also ask you a couple of questions about your personal views on software testing and CI, and of course about your experience with TestAxis.

An experiment session takes about 90 minutes and is fully remote through Zoom. To thank you for your participation, we raffle four 15 euro gift cards among the participants.

I WANT TO HELP!







The Project

Hi! I'm Casper Boone and I'm in my final stage of my master Computer Science at Delft University of Technology in the Netherlands. The research project above is the topic of my master's thesis which I've been working on since September 2020. I work on TestAxis under the supervision of prof. dr. Andy Zaidman and Carolin Brandt.

The goal of TestAxis is to give developers insights into failing tests after CI builds and to provide them with information that helps to resolve the issues causing the failing tests faster. The information presented to the user is based on (historical) test executions with the goal to shift the axis from builds to tests.

TU Delft

Appendix E

Results

In this appendix, we list the full results of the experiment. Section E.1 and Section E.4 show the results of the one-group pretest-posttest study. In Section E.2, we list the results of the post-assignment questionnaire of the within-subjects study for each assignment. Finally, in Section E.3, we present the assignment failure-fixing time results per participant.

E.1 Pretest Questionnaire Results

E.1.1 Personal Background

See Figure 5.1.

E.1.2 Experience

Development Experience

See Figure 5.2.

Testing Experience

See Figure 5.3.

CI Experience

See Figure 5.4.

E.1.3 Attitude towards software testing and continuous integration

See Figure 5.5, 5.6, 5.7, and 5.14 for the Likert-scale questions.

What are the steps you take when a build fails because of a failing tests between encountering the build failure and committing the fix?
1. Looking at error logs and identifying the failing line(s)
2. Trying to replicate the failing line in a (simple) test

3. Solving the issue in the code
4. Verifying if the issue is resolved by some (manual) testing
5. Committing the fix

Going to Travis, identifying the failing build, inspecting (the bottom of) its logs, seeing which test failed, rerunning that test locally to see if I can reproduce, fixing the test, and adding/committing/pushing it

---

1. Inspect the log
2. Look up the source code
3. Think about how it could have gone wrong
4. Apply patch and test if it worked locally
5. Commit and Push fix to Git

I look at the build log to see the exit code or failure message and hope that it tells me what I need to know. If that does not give me enough info I try to run the build locally to see if I get the same result. Once I think I have fixed it, I run the build locally again to verify it, and then push it and hope for the best.

I go to the git repository in my browser, I click the link to the failed build, I scroll down until I see red text or something that looks like a failure or stack trace, I look up what part of the code caused it, and then I try to fix it, if possible add a test that also fails on my local machine until the code is fixed.

Read the failure message and try to identify the location of the fault. If needed, I will write or update a test to check for the expected behaviour. Otherwise, I will use debugging tools to diagnose the issue. However, when the issue is not with the code but of improper handling of build tools, I will first search around the internet for a solution.

I first observe the stacktrace, then identify any components I own. Thereafter if the issue is obvious I fix it immediately, else use logging or breakpoints(not that often) to understand the logic flow.

Search for which test fails in the log.
Inspect the failure details (which assertion fails) and find the corresponding location in the code.
Figure out which part of the test or code to change (depending on how complicated the issue is, also debug the application)
Write the fix and commit it.

Static analysis of the test report, then verifying that the test code is written in a correct order followed with static analysis of the tested code if static analysis is inconclusive I usually continue towards debugging using break points and validating that the logic is in order. Another thing I would consider doing is to check the change log within git to see if this code was working before hand and what changes have been made (this only applies to regression testing of course).

Check the logs
Retry running the tests locally
Identify failing tests
Identify failure reason
Fix

Scan/read logs on CI/CD pipeline
Run build and tests locally, see if they also fail
Further investigate and solve problem
Run build and tests locally, see if they now pass
Push to repository and see if the pipeline passes

I look at the aggregated TeamCity logs to see what tests fail (if tests failed) and amend a fix to the commit and push it directly. TeamCity also offers the ability to see if the test is ”flaky” - in this case I simply re-start the build.

If Gitlab is used I would manually inspect which specific pipeline failed (for testing purposes we have separated our CICD tests into different ”domains” to quickly identify which ”domain” is incorrect) and scroll (read: look)

at the bottom of the logs to find out which tests failed - try to reproduce locally (if possible) and git commit / amend a fix.

I would take a quick look at the test that is failing on CI and then reproduce it on my development machine, and go from there.

---

* Investigate the build failure
* Consider how to fix the issue
* Implement the chosen solution & add a test or multiple tests for the given problem
* Consider whether the test(s) verify the problem correctly

---

1) Look at the stacktrace
2) Check if it passes locally
3) If it passes, then check the log of the CI
4) Use the log(exceptions) to find the lines/lines of code that may cause the test to fail
5) Find the bug, if it exists, and commit
6) If there is no bug, revisit the test and check if it is what is expected

---

1. Read which unit tests fails
2. Read what the expected value and actual values are
3. Read the stacktrace if present
4. Deduce the problem
5. If I cannot deduce why it fails, then use the debugger to step through the unit tests and the logic and perform intermedian tests to inspect the state.
6. Try to solve the problem

Are there other ways you are notified of failing builds? (optional question)

Just checking the GitHub action/build myself after I committed to a PR.

A Slack notification

I have used Slack bots in the past but they cause a lot of spam

---

Students or other developers that I am working with this notify me that my build failed.

---

push notifications for mobile apps for GitHub, Netlify, etc.

Usually I would run tests before opening my PR to validate their state. Co-workers that reviewed code would also indicate to others that tests were failing

---

Slack

We use Gerrit for some of our codebase which will automatically give a ”-1” score for the change - blocking any merge - this is quite intrusive and therefore it becomes quite obvious to us that the build failed.

For larger builds (read: nightly) we use mattermost.

If Gitlab is used, typically we use mattermost.

Microsoft Teams bot

Discord web hook

By colleagues through the chat.

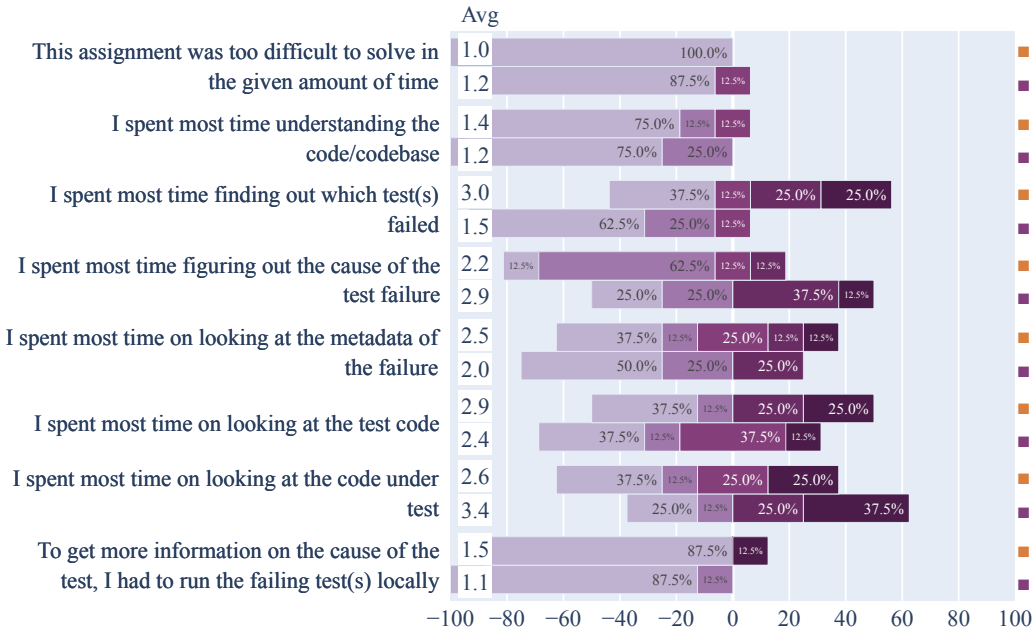
E.1.4 Expectations of a CI build test execution visualization tool

See Figure 5.11.

E.2 Post-Assignment Questionnaires Results

In the results below, the orange squares indicate the *without* variant and the purple squares the *with* variant of an assignment.

E.2.1 Assignment 1a

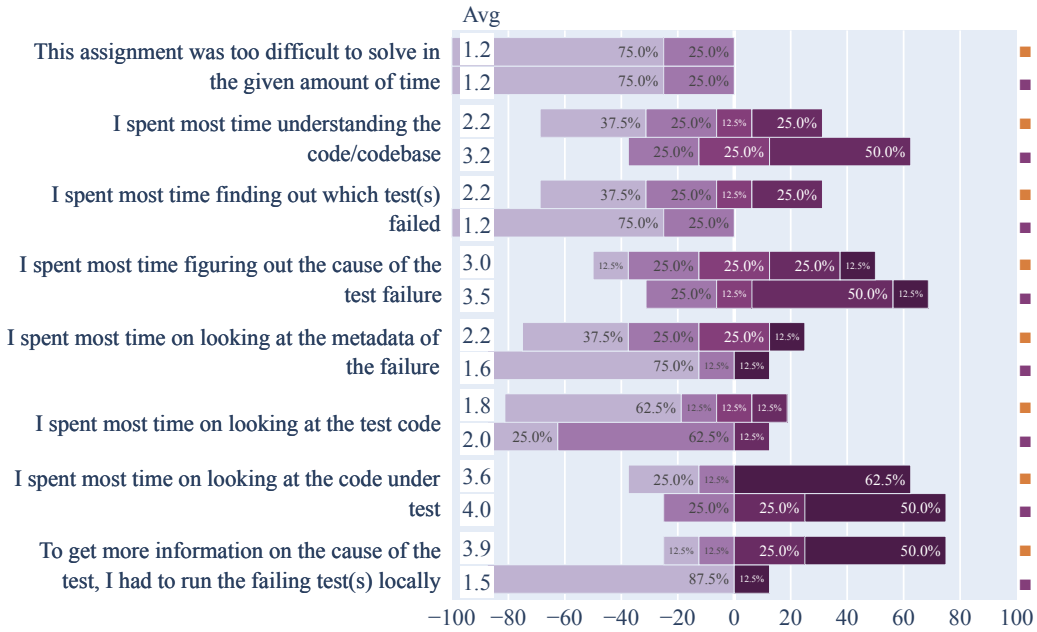


[without] Do you have any remarks? (optional)
Only one file being changed made it quite obvious what the cause was.

Since I knew the only change was a rename of a resource file, I was pretty sure the problem had to do with resource paths being out of date, and I didn't really have to understand the test cases/cut for this problem.

[with] Do you have any remarks? (optional)
Mostly figured out the issue by experience.

E.2.2 Assignment 1b



[without] Do you have any remarks? (optional)

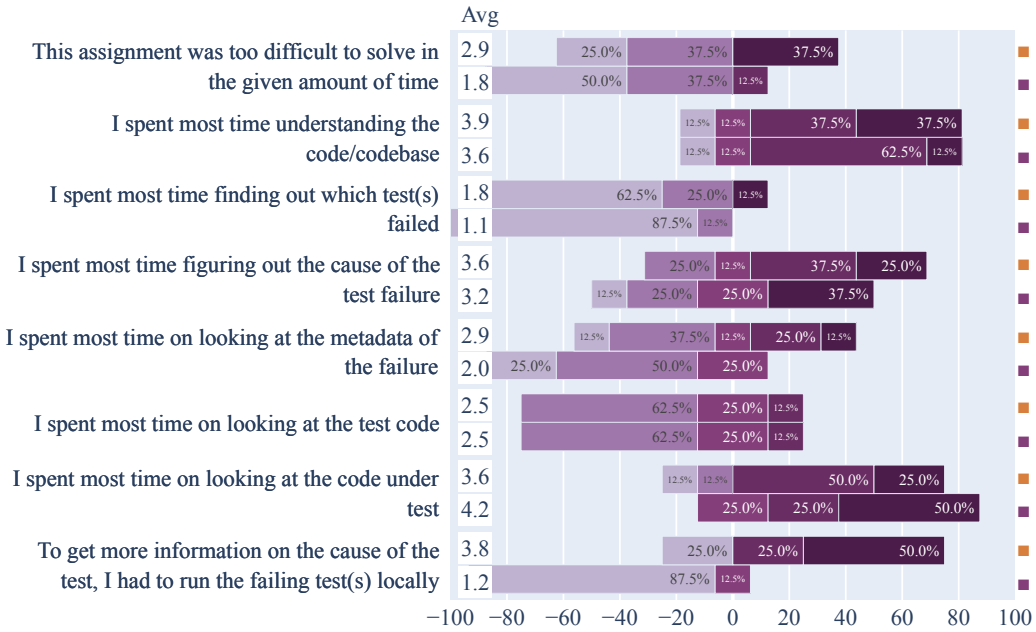
ez pz

[with] Do you have any remarks? (optional)

Very cool!

The Smoke test failing (along with others) made it obvious to me that the code changes was the first thing to look at

E.2.3 Assignment 2a



[without] Do you have any remarks? (optional)

-

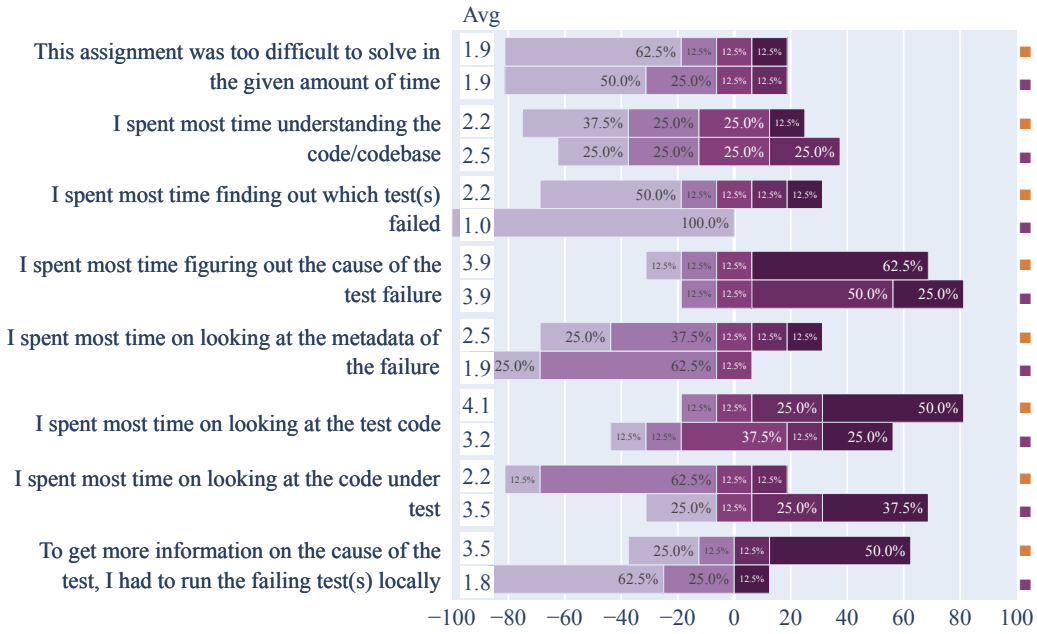
[with] Do you have any remarks? (optional)

UI remark: The code highlighting in the test code field and the code under test field is a bit confusing at first glance. In the CUT field, there are 3 different highlighting colors with different labels, but in the test code field, there's only one color without label. After a bit of thinking it's clear that this is simply the "changed" color, but this still caused some initial confusion for me.

I had to orient myself around the codebase a bit before I could really get into solving the failing test.

It would be convenient if TestAxis would load the same code fragment when you click a link in the stacktrace.

E.2.4 Assignment 2b



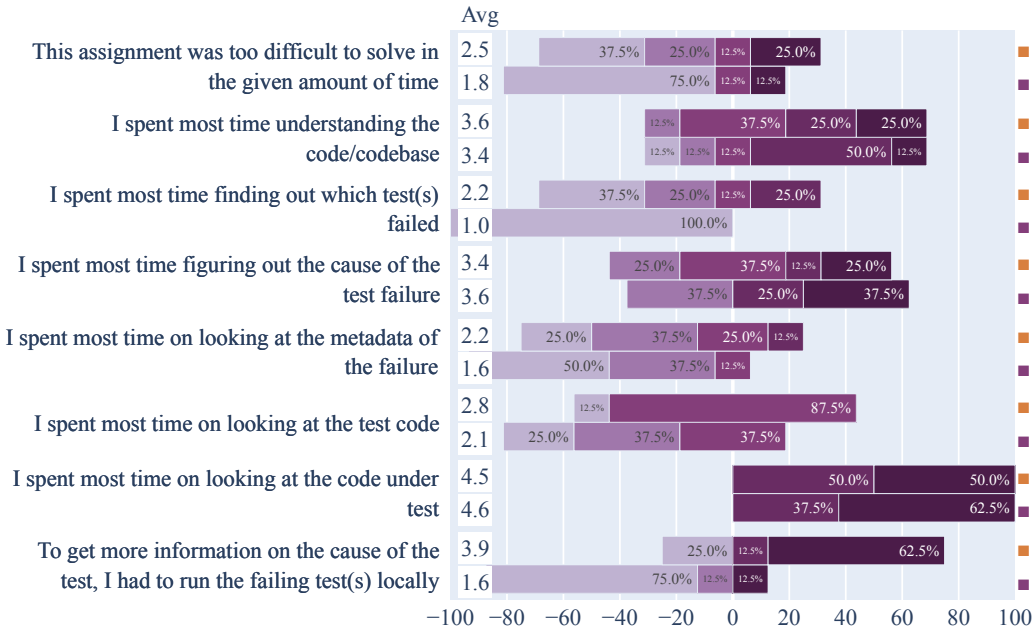
[without] Do you have any remarks? (optional)

-

[with] Do you have any remarks? (optional)

I ran the test code locally but that didn't really yield any useful information.

E.2.5 Assignment 3a



[without] Do you have any remarks? (optional)

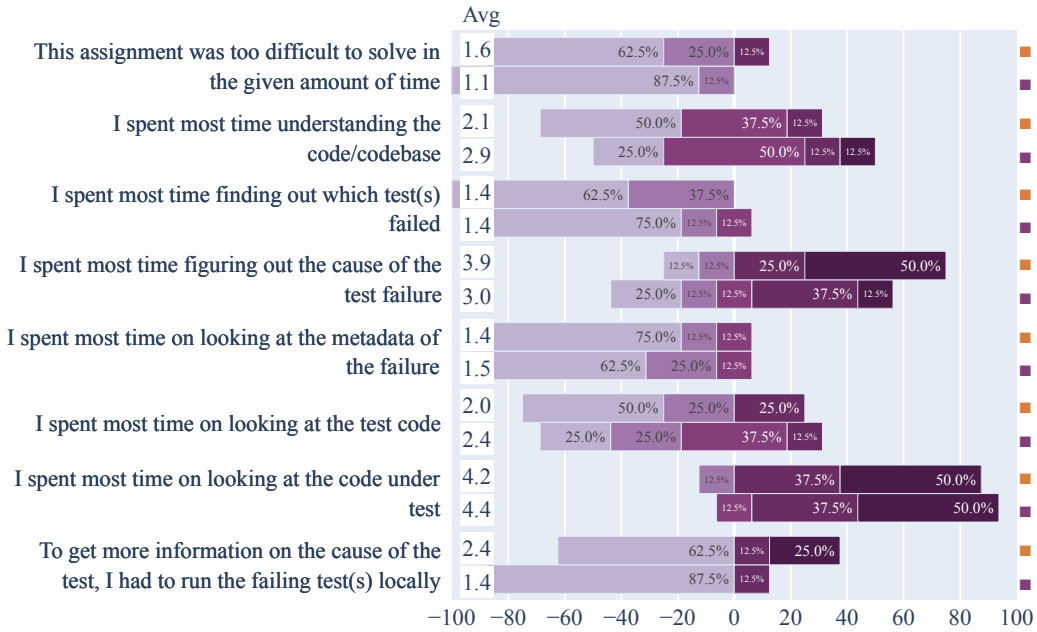
Use Sonar too :P

[with] Do you have any remarks? (optional)

Felt a lot easier than the first task.

I spent most time verifying that the change would actually cause the test to fail - and some time on figuring out my keyboard.

E.2.6 Assignment 3b



[without] Do you have any remarks? (optional)

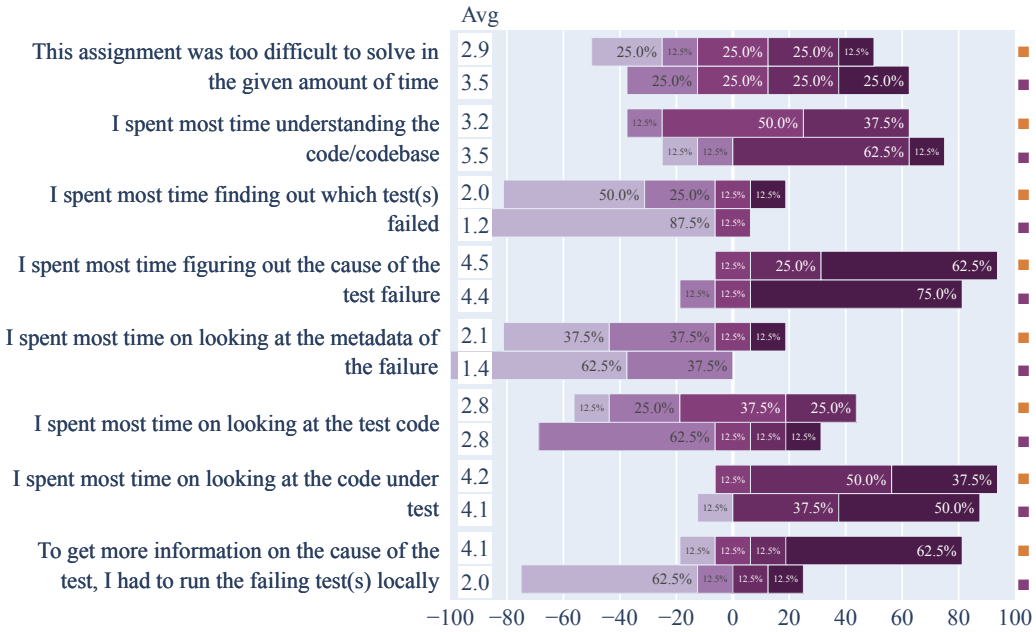
It was a bit more difficult figure out what the failing test was testing - it was a bit surprising to me to not see obvious unit test failing that would test the behaviour of ”Direction.opposite()”

I kind of knew that the problem would be in the newly introduced method (’opposite’) due to the constraints of the experiment and once I looked at the method, the discrepancy was obvious, and I didn’t really need to understand the test code.

[with] Do you have any remarks? (optional)

-

E.2.7 Assignment 4a



[without] Do you have any remarks? (optional)

I forgot git is a thing.

Running locally was quite important as in the build logs the values did not show up

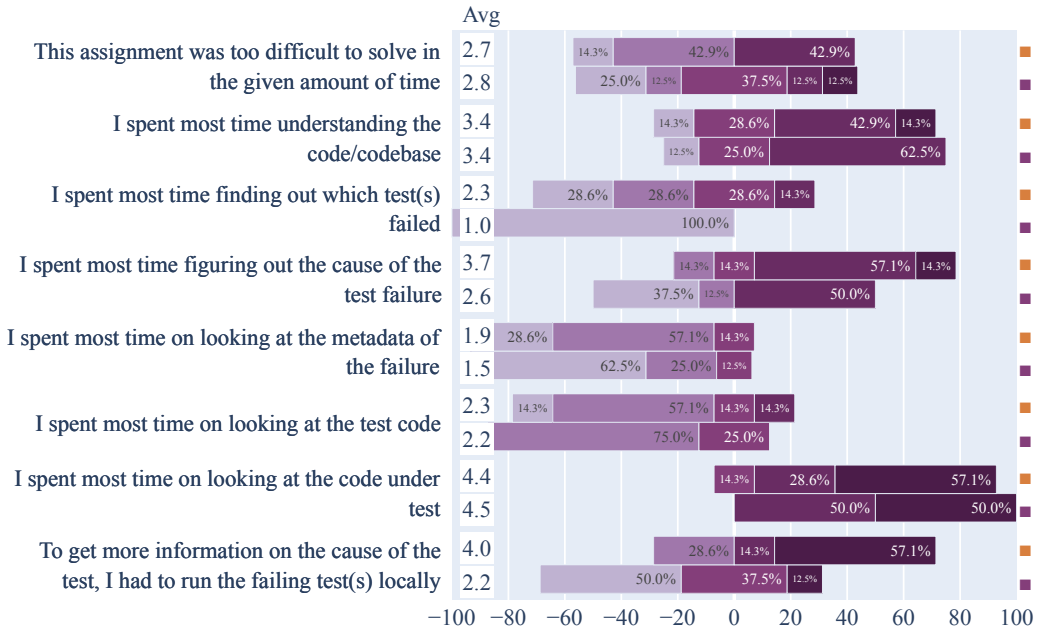
[with] Do you have any remarks? (optional)

When I click away from a tab within TestAxis and come back to that tab later, I don't end up where I left off

I would also like to see only the changed code because TestAxis shows too much information, would be nice if I could toggle the different information.

I cannot dismiss the warnings.

E.2.8 Assignment 4b



[without] Do you have any remarks? (optional)

-

[with] Do you have any remarks? (optional)

You should add a right mouse option ”Run locally” (please)

Used some gut intuition to figure out the issue
Purple highlighted code is useful to limit scope of what to look at.
Re-running the test was not needed as TestAxis gave the assert failure locally

E.3 Assignment Timing Results

Participant	1a-without	1a-with	1b-without	1b-with	2a-without	2a-with	2b-without	2b-with
1	01:20			01:30		05:00	03:20	
2	01:35			00:55		02:35	01:48	
3	00:58			01:44	02:17			04:42
4		00:19	02:42			03:48	02:22	
5	00:57			02:41	05:00			02:08
6		00:52	04:27			04:18	05:00	
7	01:48			01:02	05:00			*
8		01:38	01:52		03:51			01:49
9		00:41	02:24		05:00			03:38
10		00:34	01:59		05:00			01:10
11		00:39	01:50			01:26	05:00	
12	00:37			00:41		05:00	02:19	
13	02:51			02:01	05:00			05:00
14	01:24			04:43		05:00	05:00	
15		03:22	02:12		05:00			05:00
16		02:37	01:38			02:40	05:00	
Participant	**3a-without**	**3a-with**	**3b-without**	**3b-with**	**4a-without**	**4a-with**	**4b-without**	**4b-with**
1	04:10			00:46		06:49	10:00	
2	02:24			00:57		04:10	01:45	
3		02:28	05:00			10:00	10:00	
4		02:31	01:22			08:41	08:24	
5		03:46	05:00		08:40			03:02
6	05:00			00:39	10:00			05:06
7	05:00			00:16	10:00			10:00
8	04:42			02:03	05:53			05:52
9	05:00			02:12	04:12			10:00
10		03:09	01:51		07:01			06:39
11	04:43			00:29	07:08			01:42
12		03:02	03:17		06:54			02:38
13		02:12	03:26			10:00	10:00	
14		03:17	03:31			10:00	10:00	
15		05:00	05:00			04:56	06:53	
16	04:11			01:18		09:14	08:12	

* *Omitted result as explained in Section 5.1.2.*

E.4 Posttest Questionnaire Results

Verification

See Figure 5.16.

Usefulness of Informational Elements

See Figure 5.10.

Build Notifications

See Figure 5.13.

Test Health Warnings

See Figure 5.15.

TestAxis IDE Plugin User Experience

See Figure 5.12.

Expectations of a CI build test execution visualization tool

See Figure 5.11.

Comments

Is there any information or feature missing in TestAxis that would have improved your ability to fix a failing test?

Not at this moment.

Being able to rerun tests locally right from inside the test code tab (if that's not already possible).

Would also be great to be able to go right back from a line in TestAxis to the corresponding line in the GitHub PR, to be able to place a comment about it to communicate some thought/idea to colleagues. In that same lane, I could also think of a bunch of other GitHub-related integrations that could make the local experience even richer and more entwined with the online/collaborative experience.

Minimap

---

1. Add coverage highlighting in the diff of the CUT. In fact, show diff by default inside the tab.
2. If it's from a PR, add PR info somewhere maybe?

An option to run the test locally (maybe even automatically to detect tests that only fail on CI). Also it would have been nice to open the code under test in the editor.

Navigation shortcuts for the different labels in order to quickly jump to affected source code without having to scroll.

No.

Right clicking the test so I can run it via test axis

Quickly running the test again by right clicking it.

I think just a couple simple things like opening a file in TestAxis in the main I DE editor would be great. Another feature I'd like to see is to be able to run a failing test locally, so that a user can verify their changes intended to fix the failing test.

See remarks

Do you have any comments or remarks on TestAxis in general?

I think it's very nice not to crawl through large build logs myself.

One small quirk: When I return to the TestAxis tool tab from the main window, the current file is marked gray, and I need to switch between files to reactivate it. It would be great if that could be done for me, automatically.

Other than that: Great tool, and really useful! Especially the differential highlighting feature, but also simply the integration of such a core part of an everyday development workflow into the local IDE is really useful. Also works quite stably in the scenarios I worked with in this experiment, no bugs or the like.

I would consider using it in my workflow as is. I would not add much info further as I think it strength lays in the clean and concise overview

cool

I think it saves a lot of time and it is nicely integrated with Jetbrains IDEs.

The integration offers a great experience. However, the need of additional source code windows besides the main editor one takes up too much space in my opinion. But maybe I were to use multiple screens, this would not be much of an issue.

Well structured information with well thought oout UX design

Very useful.

Nope

I liked it.

The covered code/changed code features is awesome.

I thought it was super useful during the experiments. I much rather preferred using TestAxis over the traditional CI logs on github. What TestAxis does in my opinion is recreate the steps I manually take on a github pull request to identify a failing test, and it does so in the IDE so I don't have to switch tabs and interrupt my workflow.

- For the best experience, requires a user to learn the intuitions of the tool

It is a great time saver, especially the hints on flakiness and the covered part of the code under test

None

Do you have any comments or remarks on the experiment?

None, clear exercises and good explanation upfront! One comment on the last experiment was noted already (about the point thresholds), other than that very good but also quite representative while still doable test tasks.

Nice setup, enjoyed working along

The tasks were sometimes really easy and did not always highlight the benefits of TestAxis. Some tasks could be fixed with a simple code review, and TestAxis doesn't really seem to integrate with the code review process. Right now TestAxis is only used by a reviewer if he cannot find the bug by reviewing. But maybe the issue is that in this experiment I was the reviewer while in reality you'd use it as the author of the changed code.

I think it was consistent and I did not feel stressed.

Good explanations of the tasks at hand!

Well conducted and structured. Followed good practices.

Some of the questions in the questionnaire might not give the insight that you are after.

No it was fun

Very well set up and smooth experiment !

The experiment was very streamlined. The only thing I'd say is remote control over Zoom results in a noticeable input lag that can make things a little slower (like navigating files etc). Maybe this can be compensated by increasing the time duration for the experiment?

- JPacman is a relatively clean code base; in day to day life, code bases are often much more messy and less accessible; by having a less approachable code base, finding bugs becomes harder

This was a fun experiment to partake in. Most of the bugs were simple yet subtle just like in the real world.

Maybe another 5-10 minutes to browse through the codebase and play the game.

Do you have any other comments or remarks? None, other than: when is this readily and publicly available and a default plugin in IntelliJ?

good luck
It would be cool to test this with other programming languages, but I imagine this takes a lot of time.
Will you made your tool available to the public?
No :)
Nope
I think TextAxis is a very nice tool
Super useful tool, excited to try it out in my projects!
Good luck!

The Experiment

See Figure 5.17.