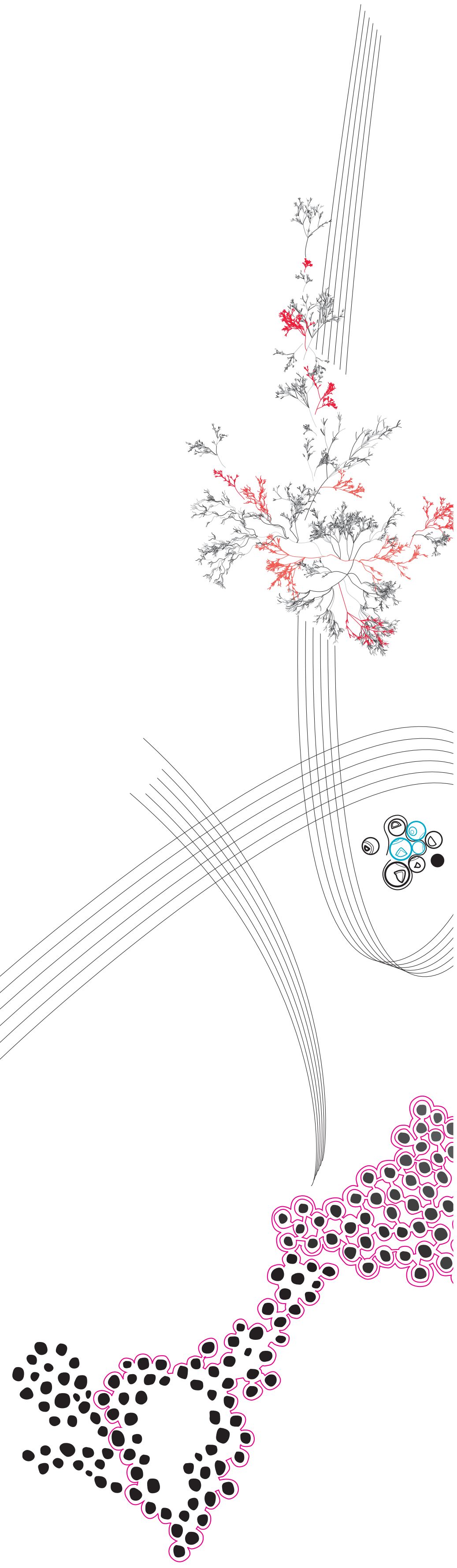
MSc Computer Science
Thesis



Introducing Automated Testing into an Existing Codebase with Limited Effort

T.G.J. Bolding

Supervisor: dr. P. van den Bos

July, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

When introducing automated tests in an existing codebase with limited effort, one needs to prioritize the creation of the possible tests. To do so, fault-proneness metrics and test implementation effort estimation can be used such that the most faults are likely to be detected with the least effort required. However, in an environment where not all functionality is equally valued, user value can be taken into account as well.

In this research UPSS, a strategy for suggesting and prioritizing tests with as little manual effort required as possible, is presented and evaluated. The strategy is accompanied by an implementation of the strategy in an automated tool, TIES. The strategy shows promising results, even though some refinement on one of the factors used in its test prioritization is required. The tool presents a novel approach to linking code and functionality in a codebase, giving it the ability to provide meaningful insights.

Keywords: computer, science, testing, automated testing, test prioritization, testing effort estimation, user value, fault-proneness

Preface

This research was performed in cooperation with Ordina[1], ProRail[2], and RiskChallenger[3]. One software project used for the case studies in this research is an application developed by Ordina for ProRail, the other is developed by RiskChallenger.

[1]https://www.ordina.com/
[2]https://www.prorail.nl/
[3]https://www.riskchallenger.nl/

Contents

Chapter 1

Introduction

Automated testing helps ensure the quality of a software project. However, implementing and maintaining a suite of automated tests requires significant effort that is then not available for the development of new features.

It is therefore the case that in some situations, developers and/or their managers choose to develop a software system without immediately co-developing a test suite. If then, after some time, the desire for automated tests arises it can be hard to know which tests to implement; i.e. which components of the system to test and how thorough. In a perfect world an "all-covering" test suite would be developed that tests every component sufficiently, ensuring that no undesired behavior remains. In the real world however, limited time is (made) available for designing, implementing, and maintaining such a test suite, also since there might still be the desire to focus on developing new features over maintaining and testing old ones. In such a case, one has to choose which tests-to-be to prioritize. When creating such a prioritization, a few questions arise. The first of these questions is "What tests *could* we write?". Then one might ask "Which of these tests are most important?". And finally one could wonder "How much effort would implementing these tests cost?", since without such an estimate it is impossible to know which tests can be implemented in the available time.

The process of answering these questions is traditionally labor intensive; requiring someone to analyze the codebase, look at (functional) requirements, and estimate the effort required to implement the tests manually. For e.g. a small team or a team that wants to focus on innovation, going through this process could well not be feasible. An incorporated strategy that uses automation could assist the team, taking for example the tasks of test suggestion, prioritization, and effort estimation out of the team's hands.

This research presents a strategy that is able to perform the tasks mentioned in a way that is as lightweight and automated as possible. The ability of this strategy to be automated is demonstrated by implementing a tool implementing the strategy. The tool and the strategy are then evaluated through user studies on two projects.

The strategy is designed by examining existing literature and incorporating it along with novel ideas.

The research is incentivized by a case study from a team that is developing prototype software, and has been doing so for quite some years. Until recently there has been (close to) no attention paid to implementing automated tests, as the focus has been (and still is) on innovating in the application. However, the team did find that the lack of automated quality assurance hurts them by having demonstrations fail because of undiscovered failures in the software. Also, due to the nature of the project no formal requirement specifications exist for the entire system.

1.1 Research questions

We have transformed the problem stated before into research questions that we aim to answer with this research.

RQ1 How to introduce automated testing into an existing codebase with limited effort?

RQ1.1 How to suggest possible tests in an automated manner?

RQ1.2 How to prioritize these suggested tests?

RQ1.3 How to estimate the effort of implementing these suggested tests in an automated manner?

RQ1.4 How to rank the suggested tests?

Chapter 2

Background

2.1 Types of tests

There are multiple dimensions to testing[33]. Tests can have different goals, different amounts of knowledge about the system under test (SUT), and different ways of interacting with the SUT, among others. In this section some background will be given on these different dimensions.

2.1.1 Testing goal

Not all tests have the same *goal*. Some could focus on ensuring that a system implements desired functionality, while others could focus on ensuring that the system implements this functionality well (for some definition of well), for example. ISO 25010[1] specifies attributes in which the quality of a software system can be measured. Examples of these attributes are functional suitability, performance efficiency, and maintainability.

Different tests can focus on assessing the quality of a software system on different attributes. For example, a performance test could measure the response time of system to assess its performance, while a different test could test the contents of said response to assess the system's functional correctness. Another test could use a *fuzzer* to check for security vulnerabilities[58].

2.1.2 Knowledge of SUT

Also, not all tests have the same knowledge of the SUT. The most commonly used terms to describe having knowledge of the SUT or not, are white-box and black-box testing respectively.

White-box testing

White-box tests have knowledge of the internal workings of the system under test, and often test a specific part of the system. A common example of a white-box test is the unit test. In a unit test a unit is tested. This unit is often a piece of the source code that cannot be divided further (i.e. the smallest testable component). Most often this is a specific function. With these unit tests it can be confirmed that the implementation of a unit conforms to a specification. Even though a unit test is not able to test all combined

[1]`https://www.iso.org/standard/35733.html`

functions of a system, major advantages of unit tests are their ease of implementation and maintenance[39]. Because of their specificity to a unit, they do not need to be changed when the source code of a different unit is changed or when functionality is added.

Black-box testing

Black-box tests do not have knowledge of the internal workings of the SUT; they only interact with the system's outward facing interface(s). An example of a black-box test is a system test. In a system test an interaction with the system is tested, for example through having an automated program use the system as a user would. System tests can be helpful in checking whether the system's behavior complies with the desired behavior/requirements. A drawback of system tests is that the effort required to implement and maintain them can be large, due to their dependence on multiple units of source code, among other reasons.

2.2 Case studies

In this research, two software projects and their maintainers were available for case studies. Both software projects are in active development by software development companies. Since the case studies are referenced to in other chapters, and their details might be of importance, they are described here.

The first of these projects is the project mentioned in the introduction of this reportr that incentivized this research. This project is a simulation tool wherein new technologies and developments in the train controlling sector can be experimented with. For this, the actual system that a train controller would use is emulated using a more modern software stack. This emulated system can then be demo'ed with train controllers and drivers as a means of testing whether the workflow with the new technologies works well for the envisioned users. Since from the start of this project the choice was made to focus on innovation and the rapid development of new features, the amount of attention given to the implementation of tests has been close to zero. Quality assurance is therefore performed by manually going through the system and checking that behavior matches expectations. This way of working obviously is highly failure-prone and has resulted in cases where surprises with non-conforming functionality have occurred when demo'ing with users. For this reason, a desire to implement automated tests has arisen. The project's codebase lends itself well for a case study of the proposed research due to its size and age. The part of the system that the case study will focus on is a backend written in Java[2]; for this reason we will refer to this project as "the Java project".

The other project is a web-application for risk management. It is currently used in production and already has a large, but non-exhaustive (meaning that not all functions and units are covered), test suite. This test suite contains both unit and functional tests. Again the part of the system that the case study will focus on is a backend, but in this system it is written in TypeScript[3]. Therefore we will refer to this project as "the TypeScript project".

These two projects were chosen because of available access to their codebases, as well as the possibility to have extensive contact with their maintainers. This contact was essential to being able to perform extensive case studies.

[2]https://www.java.com/
[3]https://www.typescriptlang.org/

Chapter 3

Related work

In this chapter existing literature related to the proposed research and some background information is described.

3.1 Test suggestion

3.1.1 Automated test case/suite generation

Because writing test cases for a software system is labour-intensive, automated test case generation is an active research topic. In [3], Anand et al. present an orchestrated survey of literature on the topic of Automation of Software Testing. Per subtopic, an allocated author has written a section that gives a brief description of the technique's underlying ideas, a survey of the current state of the art in research, practical use of the technique, and a discussion of further development of the technique. From this orchestrated survey we identified some topics relevant to our research. In the next sections we summarize relevant findings from Anand et al.'s survey and enhance them with contemporary literature.

Search Based Software Testing

Search-Based Software Testing (SBST) is a branch of Search-Based Software Engineering that uses search algorithms to find test inputs that maximize some objective or objectives (such as branch coverage). Due to the ability of handling multi-objective targets SBST techniques are able to balance cost and benefit by, e.g., minimizing the number of tests and maximizing the number of branches covered.

EvoSuite[1] is an open source SBST system for Java first introduced by Fraser & Arcuri in [20]. It is able to automatically generate JUnit[2] (a Java unit testing framework) test suites for Java classes based on the current behavior of the class. In [22] Fraser et al. use EvoSuite to investigate the issues concerning the observed limited use of automated test case generation in practice. They found out that even though, a.o., a large increase in code coverage (up to 300%) was observed when subjects started using automated test, there was no measurable improvement in the number of bugs actually found. In fact, slight decreases in number of bugs found were observed when subjects used EvoSuite compared to when they wrote tests manually, however, this decrease was not statistically significant. They

[1]https://www.evosuite.org/
[2]https://junit.org/

observe that it seems that testers write stronger assertions manually than those that are generated by EvoSuite and show that significant effort is required by testers to adapt and correct automatically generated tests.

Artificial Intelligence

Artificial Intelligence (AI) has been on the rise lately. In a short time, services such as ChatGPT[3], GitHub Copilot[4], and Tabnine [5] have become trusted upon by developers to augment their coding experience. These services can generate complete source files, partial implementations, and/or tests based on a given source codebase. Under the hood these services use some kind of Large Language Model (LLM).

Wang et al. in [54] survey contemporary research and findings in the field of LLMs for software testing. They find that LLMs are commonly used for tasks such as test case generation, but not for tasks that happen more early on in the testing process, such as test plan generation.

3.2 Test prioritization

Often it is not feasible to implement all possible test cases in a test suite, therefore a selection has to be made. Fenton & Ohlsson describe in [18] that the Pareto principle also applies to the relation of software modules and faults; a small number of modules contribute to a large number of faults. While the exact numbers differ between studies (from 20% of the modules giving 60% of the faults, to even a study where 10% of the modules gave rise to 80% of the faults), this demonstrated effect shows that prioritization of tests can be highly effective in optimizing the number of faults found per amount of effort spend on implementing test cases, by focusing testing efforts on the most fault-prone parts of a software system.

To make a selection on which test cases to implement in a test suite, the possible tests need to be prioritized in some way. This section presents several ways in which existing work perform this prioritization.

Several approaches exist for prioritizing the *execution* (as opposed to implementation) of test cases[38, 53, 56, 25]. Do et al. in [14] propose a method for doing so for a Java project using JUnit. They experiment with different techniques for prioritizing test cases based on the amount of code coverage the test. They found out that prioritizing test cases by code coverage leads to faults in the software being found earlier, thereby minimizing the amount of resources spent on executing tests. These approaches for test execution prioritization are not directly relevant for our research, as we cover test implementation, but their prioritization methods were of inspiration to us.

3.2.1 Fault prediction

The field of fault prediction in a codebase is thoroughly researched. Literature describes a lot of different possible ways to build predictors. An example of a large literature review in this field is [37] by Rathore & Kumar. They list traditional software metrics (e.g., lines of code (LOC)), object-oriented specific metrics (e.g., class coupling), process metrics (e.g.,

[3]https://chat.openai.com/
[4]https://github.com/features/copilot
[5]https://www.tabnine.com/

code change frequency or number of developers involved) among the metrics used to build predictors found in literature. They also list methods to combine these metrics into a predictor; examples of these methods are using machine learning (such as in [8]) and logistic regression analysis. In this section some methods are showcased, focusing on what Rathore & Kumar call "traditional" (e.g., code complexity) and "process" (e.g., Version Control System (VCS)-based) metrics. These metrics can be used for the static analysis of most codebases, whereas object-oriented and dynamic metrics can only be used for software that is built in an object-oriented language or through non-static analysis, respectively.

In [59] Zimmermann et al. suggest that the more complex code is, the more defects it has. They come to this conclusion by using regression to retroactively correlate defects/bugs (found through VCS history) with the complexity of the code where the bug originated. They found out that total lines of code remains the best predictor of defect-prone files and packages. They also found out that on a method level, LOC correlates better with failures than McCabe complexity (as is also suggested by [19] and [42], wherein both criticize the use of code complexity metrics over "simple" line counting for fault prediction), as well as that predicting faults on a package level is more successful than predicting faults on a file level.

A similar study is the one of Nagappan et al.[34]. They show a method of using historical data, mined from version and bug databases, to build fault-proneness predictors. Another contribution is showing that different fault-proneness predictors could have varying levels of relative performance over different software projects. A predictor that performs well for one project, could perform (relatively) poorly on another and vice versa.

Holschuh et al. show empirical results of experimenting with different fault-proneness predictors on several hundred Java projects in [26]. The metrics that they used for the predictors stem from a wide assortment of sources; they use code complexity metrics, object-oriented metrics (such as coupling), change history metrics, and code smell metrics (e.g. conformance of code to a certain code style or having unused pieces of code in a file). To evaluate the predictors they mine defect and change histories to find the number of defects fixed in a component. Then, they perform a correlation analysis to see which predictor would have best predicted which components would contain defects. Also, their results confirm the findings of Nagappan et al.[34] that the performance of predictors varies between software projects. Also, they confirm the aforementioned Pareto principle and its applicability to the domain of software engineering; in the experiments of Holschuh et al. it is found that 20% of all packages contain more than 70% of the defects.

A different source for heuristics predicting fault-proneness of a software component is historical data of the specific software project. With this heuristic source, data from for example the project's VCS is used to determine what components of the software are most likely to fail.

In [43] Shihab et al. propose an approach for applying Test-Driven Development[7] practices to already-implemented code; an approach that they call Test-Driven Maintenance. They compare several history-based heuristics, such as Most Frequently Modified, Most Recently Fixed, and Largest Modified. They show that heuristics based on the function size, modification frequency, and bug fixing frequency perform the best for prioritizing unit tests. Counterintuitively, they find that combining several of the best performing heuristics did not give an improvement over the best performing single heuristic.x

Arisholm and Briand assess the quality of fault predictions on class level for a legacy Java system in [5]. Their fault predictions are based on a logistic regression analysis on both code metrics and change history data. They show that such fault-proneness models are promising and state that "using history change and fault data about previous releases is paramount to developing a useful prediction model on a given release".

Nagappan et al. in [35] propose looking at "change bursts", consecutive changes to the same piece of code, for predicting fault-prone code. This predictor only uses VCS history, making it simple to use. However, they do note that it requires code changes only to be committed when they are "ready", i.e., expected to keep the system stable. The argument for this is that even though the code was deemed ready, it still had to be changed again, giving rise to the suggestion that *maybe* the functionality is more difficult to implement correctly than expected.

3.2.2 Risk-based testing

Risk-based testing is a method of test prioritization where the estimated "risk exposure" of a function is used to determine on which parts of a codebase to focus the testing process. This risk exposure is calculated by multiplying the likelihood of failure (determined by examining factors concerning, a.o., the (perceived) complexity of the function and the quality of its specification) with the impact of failure (determined, for example, by the Product Owner) [17].

3.3 Testing effort estimation

There is a significant amount of existing research into the field of software development effort estimation that can be used during the design phase of a project to estimate the amount of effort required to bring the project to completion. Examples of often cited techniques are Function Point Analysis[2], Use Case Points[28], and COCOMO[12]. A subtly, but significantly, different field is "Testing Effort Estimation" (TEE), which aims to estimate the amount of work needed to implement tests for a given software project. Bluemke & Malanowska found out that research into this field is surprisingly scarce[11]. Techniques in the field of TEE can be subdivided into two categories:

- techniques that should be used during/just after the design phase of a project and mostly use the requirements that are gathered,
- techniques that need to be used later in the project development cycle, when (some) code has already been written.

Since the proposed research aims to provide a strategy for an existing codebase for which requirement specifications are not necessarily available, techniques of the latter category are the most relevant. However, research into these latter category techniques is relatively scarce, and since techniques of the former category can be inspirational, research into techniques in both categories needs to be considered.

Bluemke & Malanowska provide a review of existing literature in the field of TEE in [11]. This literature review was used extensively in the process of compiling the list of literature to read for this research.

It should also be noted that there is extensive research into the field of estimating manual testing execution effort, such as [4], [16], and [15]. While interesting, this is not relevant for the proposed research since we only considered automated testing.

3.3.1 Requirements-based techniques

Sharma & Kushwaha in [41] propose an approach that estimates manual testing effort (not test *implementation* effort) by using requirements specified using IEEE-Software Requirements Specifications[1] in a calculation with specified weights.

Test Point Analysis and Function Point Analysis

Veenendaal & Dekkers introduce in [51] a method for estimating the effort required for developing system tests, called Test Point Analysis (TPA). TPA is based on the Function Point Analysis (FPA) method, first introduced by Albrecht in [2]. FPA uses the Function Point (FP) unit of measurement to describe the "function value delivered to the customer", which can then be used to determine the productivity of a project by measuring the amount of work hours needed per FP. Albrecht uses FPA to compare the productivity of projects using different programming languages, however, FPA has historically been used as a general approach to estimate the complexity of a software project. FPA uses metrics like the number of inputs, outputs, and data used (among others), combined with suggested reference values to calculate the number of Function Points. By looking at past comparable projects (e.g., done with the same programming language(s)), one can deduce a productivity measure that relates the number of FPs with the number of work-hours required to implement them. Also, FPA has been further developed to incorporate contemporary programming techniques and to simplify the approach[32].

TPA extends on FPA by incorporating further metrics and reference values to estimate the amount of work-hours required to implement tests for a given function. More than FPA, TPA uses user-supplied subjective values for the relative importance and the amount of usage of a function, among others. The user has to determine the relative importance etc. and then lookup the values to be used in the TPA calculation in tables such as Table 3.1.

	User-importance	**Usage-intensity**
Low	3 (relatively low importance)	2 (used a few times per day/week)
Normal	6 (relatively normal importance)	4 (used a "great many" times per day)
High	12 (relatively high importance)	12 (used continously)

Table 3.1: TPA's user-related factors

3.3.2 Code-based techniques

Bhattacharyya & Malgazhdarov in [9] present an approach that estimates the time required by a symbolic execution tool to cover a given amount of code coverage, or vice versa. Even though no symbolic execution tool is used in this research, their research approach or code metrics used can be interesting.

Bluemke & Malanowska in [10] present a tool that estimates the testing effort based on the system's UML model. Under the hood this method uses Test Point Analysis (further described in Section 3.3.1). They also state that TPA is the only method for Testing Effort Estimation known to them that can be automated. For this last statement it should be noted that the same authors are the authors of [11], the aforementioned large literature review in the field of TEE.

3.4 Validation

Next to related work giving insights into the topic at hand, there is also related work giving inspiration for how validation of the research can be performed.
Shibab et al. in [43] look into a way of prioritizing the creation of unit tests in an existing codebase, as mentioned before. To demonstrate the effectiveness and validity of their approach they run a simulation over the history of a software project. They let their approach retroactively propose tests that should be implemented, and then see if this test would have prevented a bug. They do this by looking at commits that come after moment of the proposed test, and see if they have a "bugfix" label and touch the tested piece of code.

Chapter 4

Methodology

In this chapter we describe the methodology used to answer the research questions described in Section 1.1. Section 4.1 explains the methodology behind the creation of the test suggestion and prioritization strategy. Section 4.2 describes the methodology for the evaluation of the strategy. The resulting strategy as designed in this research is described in Chapter 5, while the evaluation results are given in Chapter 8.

4.1 Strategy

This section covers the methodology used for developing the test suggestion and prioritization strategy.

4.1.1 Test suggestion

To answer research question RQ1.1, we looked into ways that tests can be suggested for an existing codebase. To do so, we first looked at existing literature and held interviews with developers.

Input was gathered from interviews with developers to confirm/reject our ideas about strategies for suggesting tests. These discussions were held in semi-structured interviews, where a clear program for the interview was created, but open discussion was allowed. The prepared questions for these interviews are listed in the Appendix (Section D.1.1).

After literature was processed and interviews were held, a strategy for test suggestion was devised. This strategy and the details on how it was devised will be elaborated on in Section 5.1.

The devised test suggestion strategy was then implemented in an automated tool. This tool was used to provisionally suggest tests for the Java case study project. The output of this tool was checked to assert whether the suggested tests and the way of working made sense and whether the tests were exhaustive. This was done through interviews with the developers of the project, a.o. The questions and summaries of these interviews can be found in Appendix Section D.2.

4.1.2 Test prioritization

For the test prioritization and answering RQ1.2, a similar approach as for the test suggestion was taken. First, research into existing literature was performed, and afterwards a test prioritization strategy was designed. Results from the previously mentioned interviews and informal discussions with persons from the Java case study project were factored into this design.

Effort estimation

We also want the strategy to be able to estimate the (relative) amount of effort that would be required to implement and maintain suggested tests (RQ1.3). This is useful such that project maintainers can make choices based on the trade-off between the effort required to implement tests, and the priority of the respective tests.

For this topic of estimating the effort required for implementing the suggested tests, we also first performed a literature research. When this literature research was completely finished, we designed the strategy to be used; this was done by combining existing work with novel ideas that suit the specific goal of this research.

One of the restrictions of this research was that no dependency should exist on available detailed (functional) requirement specifications. Most related work (see Section 3.3), including the most used TPA method[51], does use detailed requirement specifications for the estimation of test implementation effort. The restriction to have no dependency on existing requirement specifications stems from wanting the strategy to be applicable in real-world situations where an existing codebase might not (/no longer) have requirement specifications for the entire system. This lack of a complete suite of requirement specifications is also applicable to the Java case study project (the project that incentivized this research).

4.2 Evaluation methodology

This section describes how the research was evaluated, and why this approach was chosen.

As the strategy resulting from this research uses, a.o., subjective metrics for test prioritization, the strategy cannot be evaluated completely quantitatively. If there was no subjective aspect, we could apply the strategy to a codebase, and, e.g., evaluate how many defects in the code would have been found if the suggested tests were implemented. Shibab et al. in [43] use such a method to evaluate their prioritization strategy. They evaluate their strategy by simulating over the history of a software project, and seeing whether if their strategy would have been followed, future bugs would have been prevented.

Due to the subjective aspect of our research there is a difference to Shibab et al. in how tests are to be prioritized. In an environment where the only goal is to find as many bugs as possible with the least amount of effort required (such as in Shibab et al.), tests that discover the highest number of bugs should have the highest priority. However, in our research we do not want to find as many bugs as possible; we instead want to find bugs in functionality that is deemed as the most important to be working correctly.

Subjective input leads to subjective output, which in turn leads to subjective evaluation. In this research, the subjective evaluation was performed through user studies. On two software projects, the strategy was applied by their maintainers (developers and Product Owners), and the way of working and the results were qualitatively evaluated through interviews. With this way of working, both the usability of the strategy as well as the actual results could be evaluated. We deemed evaluating both the strategy's usability and results to be necessary, since a strategy that gives perfect results, but is too cumbersome to use in practice, would not be suitable for real-world applications.

In the evaluation, the strategy and the tool implementing it (described in Chapter 7) are tightly coupled. We think evaluating the strategy without an accompanying tool would be difficult to do, because of several reasons. For one, the size of a software system desired to produce meaningful evaluation would make applying the strategy by hand infeasible. Since the strategy's goal is to suggest tests for every method that is depended on by a function (as is explained further in Chapter 5), in a real-world software system the number of suggested tests would quickly become quite large. Especially when also considering dependent methods, the number of methods per functionality becomes infeasibly high for manual processing.

Because of the described subjective nature of the evaluation, we did not want to create and use a new (small) software system purely for the evaluation. We think that in a purely experimental environment, it would be hard to impossible for users to form an opinion on the way of working with the strategy and its results, for they would have no connection with the codebase. Because of this lack of connection with the codebase, we think that it would be hard for them to judge whether a suggested test would be more important than another, for example.

Also, because of the way the strategy is designed, the connection between functionality and the code that implements the respective functionality is essential, as this information is required for both the test suggestion and prioritization. While theoretically different ways of connecting functionality with code could be used, and we do not want to claim that our approach is "the best", we feel that the code annotations approach that we designed (as described in Chapter 6) is a highly suitable and relatively accessible way of connecting code with functionality. Other approaches to the issue of connecting code with functionality could be evaluated in future work.

We wanted to evaluate the combination of the strategy and the implemented tool on several aspects. A list of these aspects with the research question that they aim to (help) answer is given in Table 4.1.

Aspect number	**Description**	**Related RQ**
1	the perceived usefulness of its test suggestions	RQ1.1
2	the perceived value of its test prioritization calculations	RQ1.2
3	the perceived value of its effort estimations	RQ1.3
4	the perceived quality of its test sorting	RQ1.4
4.1	the perceived value of using user-assigned user value and stability values for test prioritization	
4.2	the perceived value of using implementation effort estimation for test prioritization	
4.3	whether the sorting fulfils the objective of balancing test priority and implementation effort	
5	the ease of use of assigning user value and stability values to functionality	
6	the ease of use of the code annotations	

Table 4.1: Evaluation aspects

To evaluate the strategy and the tool on these aspects user studies on the case study projects were held. As described in Section 2.2, two case study projects were available for this research; one Java project, and one TypeScript project. Chapter 8 goes into detail on how these case studies were performed. An option that we considered was to use the strategy for some time, actually following it to implement tests, and seeing whether faults were uncovered. However, we decided against doing so because of time constraints. Especially since we wanted to study the approach on at least two different software projects (to study the effects of the goal of the project on the use of the strategy, a.o.), the amount of effort and time required to implement and analyze a significant number of tests would have been too large.

Chapter 5

User-value Prioritized Suggestion Strategy (UPSS)

In this chapter we will describe the strategy that we developed. The strategy's goal is to suggest and prioritize automated tests that could be implemented in an existing software project, considering both the usefulness of the test as well as the estimated amount of effort that would be required to implement it. Because of the user-assigned input that the strategy uses, we have decided tot call the strategy User-value Prioritized Suggestion Strategy (UPSS). This chapter covers purely the theoretical aspects of UPSS. In Chapters 6 and 7 we describe how we defined theories and designed tools to make UPSS effectively applicable to existing codebases.

In Section 5.1 we describe how we designed the approach to test suggestion for UPSS. Then, in Section 5.2 we describe how UPSS prioritizes these suggested tests, taking implementation effort in account, a.o.

5.1 Test suggestion

5.1.1 Exisiting work

We first looked into existing literature to check whether there were existing solutions that we could use/adapt. No relevant literature on the act of purely suggesting tests could be found. A field that we did explore was that of automated test case generation. In Section 3.1.1 we discussed existing techniques that are able to generate test cases or even full test suites for a given codebase. This might seem ideal, since it spares developers from having to spend any effort on the development of tests. Therefore, in a perfect world no developer would have to write tests anymore, and all tests would be generated by supporting tools. However, this obviously is not the world we live in (at least not at the moment), and we think this has multiple reasons.

One of these reasons is the (limited) availability of test generation tools for specific software stacks. Not for all programming languages exists a test generation tool. One of our goals with the devised strategy of this research is having a high ease of applicability to different software stacks. We want it to be as easy as possible to create implementations of the strategy for a specific programming language. In Chapter 7 we describe how we achieve this goal.

Another related issue is that even if a test generation tool exists for a programming lanuage, the tool might not be able to handle all ways of using that programming language. Fraser & Arcuri in [21] note that test case generation tools are often not capable of handling all real-world codebases well, due to, e.g., environmental dependencies (such as file input/output). They say that in literature approaches often perform well in studies due to the (biased) selection of case study codebases, and that the limitations of the approaches are only listed as threat to validity/future work.

To create a sensible test that actually fails when a fault is introduced into the source code, the test needs to contain assertions. For these assertions it needs to somehow be decided what behavior is correct, and what is not. This problem is also called the "test oracle problem" and is discussed by, a.o., Barr et al. in [6]. Test case generation tools often have no other way of generating assertions than to regard the current behavior of the code. This means that if the current behavior of the code does not actually match the desired behavior, this fault could propagate into the generated test(s). Therefore, a tester needs to check the assertions generated, costing the tester effort. Fraser et al. in [22] note it in the folling way: "Investigating how test suites evolve over the course of a testing session revealed that there is a need to rethink test generation tools: developers seem to spend most of their time analyzing what the tool produces. If the tool produces a poor initial test suite, this is clearly detrimental for testing.". This lack of understanding of the tests could persist during the future lifecycle of the codebase, leading to increased maintenance efforts required. This last observation is also made by Shamshiri et al. in [40], who empirically evaluated the influence of automated test generation on the effort required for software maintenance, and found that indeed software maintenance takes longer with generated tests.

5.1.2 Our test suggestion strategy

As discussed in Section 4.1.1, we needed to decide what types of tests to suggest (Section 2.1 discusses different types of tests and their uses). It was decided that the main focus for this research would be tests that have the goal of testing the SUT on compliance to desired behavior (functional suitability). This means that other testing goals, such as performance or security testing, were not considered for this research. A question that remained was what kind of functional suitability tests to suggest. In several interviews, developers were asked about their opinions on what tests are of use for a software project. In Appendix Sections D.1 and D.2 the pre-determined questions for the interviews, as well as summaries of the interviews, are given. Due to the semi-structured nature of the interviews, discussion was not limited to just the pure suggestion of tests. Naturally, the interviewees also talked about the further prioritization of tests, among others. The results from the interviews are used where applicable, and therefore results from interviews on test suggestion could also be found in sections on, e.g., test prioritization, and vice versa.

We hypothesized that automated testing is deemed of use and that the most important types of tests are the unit test and functional test. The interviews confirmed these hypotheses. However, interviewees do not agree on the perceived relative use of unit tests and functional tests. One interviewee stated that, if possible, he would write unit tests for every method. Contrastingly, another interviewee deemed unit tests of use, but felt like often the accompanying overhead was not worth the added value. These contrasting views exist to a lesser extent for functional tests; the interviewees all shared that tests that test functionality in some automated manner are useful, but that there is concern over the amount of effort required to develop and maintain them.

An aspect that we deemed important was the exhaustiveness of the test suggestions. With this we mean that no test should be missed, as a missing test could be the one test that was needed to discover a certain (important) bug.

It was decided to focus on both unit tests and functional tests and comply with the opinions of the interviewed developers when designing the test prioritization strategy.

5.1.3 Unit tests

From the interviews we concluded that unit tests should be suggested on a per-method basis, as It is industry-practice to write unit tests on a per-method basis[39], in contrast to, e.g., writing a single unit test for an entire class. In several interviews (Appendix Sections D.1.3, D.2.2 and D.2.3) we confirmed with developers of the Java project that they also write unit tests on a per-method basis. In other words, for every method, a separate unit test can be suggested. This also makes unit tests easy to suggest automatically. Namely, an automated tool only needs a list of all methods existing in a codebase to be able to suggest unit tests.

In Listing 5.1 we show an example `Util` Java class. With the strategy of suggesting unit tests for every method, the list of tests in Listing 5.2 is created.

This strategy is exhaustive by definition. As we suggest tests for every method that we can find.

We have thought about suggesting unit tests on a branch level, meaning that a test would be suggested for every branch that exists in a method. However, such an approach would make assumptions on how a user would implement tests. Also, incorporating branch information would increase the effort required to create a tool that gathers the required information for UPSS from a codebase. An interface displaying the suggested tests could possibly show the branches that exist in the method under test.

Note that we make no distinction between public and private methods (in a object-oriented programming context). While it is generally regarded as an anti-pattern to test private methods, there are exceptions to this advice[30]; for example, when a method's complexity is such that testing is warranted. We cannot infer when users would or would not want to test a private method, and therefore should not try to do so. Filtering out private method is therefore left as a task for a user of UPSS when desired. A second reason that private methods are treated the same as public methods, is that the insight of also showing private tests can be valuable to the user. Hidden complex implementation details can be a sign of a missing abstraction[30]. In the case of a private method being complex and therefore being ranked highly by UPSS, the user could consider refactoring the code such that it becomes public and better testable or that the complexity is reduced.

5.1.4 Functional tests

In Section 4.1.2 we described the desire to have no dependence on existing detailed (functional) requirement specifications. However, we found that automatic inference of functionality in a system is hard to impossible to do based on pure source code alone. In existing literature we found no existing suitable work that is able to perform this inference, and we were not able to come up with a fitting solution ourselves. Automatic infering of functionality based on package/module, class, and/or method names could be possible, but would rely on assumptions about the way developers name their code. Use of AI to infer functionality based on the source code might be a possibility, but is not desired for

```java
package nl.tiesb.test_suggestion;

class Util {

    public int parseBinary(String input) {
        return Integer.parseInt(input, 2);
    }

    public String[] splitOnCommas(String input) {
        return input.split(",");
    }

}
```

LISTING 5.1: Example Java code with multiple methods.

```markdown
# Unit tests:

- Unit test for "nl.tiesb.test_suggestion.Util_parseBinary"
- Unit test for "nl.tiesb.test_suggestion.Util_splitOnCommas"
```

LISTING 5.2: Suggested unit tests for the code in Listing 5.1, in Markdown.

```markdown
# Functional tests:

- Functional test for "Let the user login"
- Functional test for "Show the current list of books to the user
    ↪ "
```

LISTING 5.3: Example suggested functional tests, in Markdown.

this research because of issues with, a.o., exhaustiveness. In Section 10.2.1 we discuss the decision not to use AI further.

Therefore, some way of explicitly defining functionality is required to be able to suggest functional tests. We require users of UPSS to formulate a (short) description of every function in the system. Then, UPSS can suggest a test for every function that is described in the system. These descriptions can be short, but need to be long enough such that a developer is able to implement a functional test for the specific function. So, imagine a system that has the described functions of "Let the user login" and "Show the current list of books to the user"; the resulting list of suggested functional test would be as shown in Listing 5.3.

The exact approach for how the function descriptions should be created and put in the codebase is not a core part of UPSS's strategy, since this does not matter for the application of the test suggestion strategy. For UPSS it is only required that some short description exists. Therefore, how to extract functionality information is not covered in this current chapter. A possible approach, and the one that we have used in the design of our implementation of UPSS, is using comments in the code to describe the functions. In Chapters 6 and 7 we describe this approach in detail.

5.2 Test prioritization

As is discussed in Chapter 3, work that prioritizes tests already exists. However, discussions with the maintainers of the Java case study project (see Section 2.2), lead to an insight in a difference between existing work and a desired solution for this project. While existing work attempts to prioritize tests such that the most bugs are discovered, regardless of their location in the codebase, this is not what is always desired by maintainers. This research was incentivized by the Java case study project where there were significant differences in how important to the users certain functionality was deemed. Therefore, the maintainers want to find a balance in writing tests that discover the highest number of bugs, and tests that have a higher likelihood of finding bugs in functionality that is deemed more important to be working correctly.

As discussed in Section 4.1.2, we want UPSS's test prioritization to also factor in the effort required to implement and maintain the suggested tests. We have split the discussion of UPSS's test prioritization approach into a section on test usefulness and test effort. The usefulness value can be seen as the risk from risk-based testing (as discussed in Section 3.2.2).

In the following subsections we first give a preface on linking methods and functions. Secondly, we describe what factors *could* contribute to the usefulness of a test and what choices we have made. Thirdly, we discuss UPSS's approach to test implementation and maintenance effort estimation. Finally, we discuss we discuss how the usefulness and effort estimates are combined into one test priority value.

An important part is the heaving quantified measures for all factors of the test prioritization. This is essential in combining prioritization and effort estimation later on. For example, it does not suffice to only rate a component as "fault-prone" or "not fault-prone", since then there is no quantified value to use in the ranking calculation afterwards. In several of the following subsections equations are shown that use concepts related to code or functionality definitions. An overview of these concepts is given in Table 5.1.

5.2.1 Linking methods to functions

As we will discuss, in UPSS both information that is gathered on a method-level as well as information that is gathered on a function-level is used for test prioritization. Method-level information could, for example, be method complexity metrics. Function-level information could, for example, be ratings that a user has given to the function (e.g., how important the function is). To be able to use related method-level information for prioritizing functional tests (that are suggested based on function information) and vice versa, methods and functions need to be linked.

We say that a method is linked to a function when this method supports the function. In other words, for the function to be fulfilled by the program the method is called.

Symbol	Description
M	The set of all methods in the system
m	A specific method from M $(m \in M)$
c_m	The estimated complexity of method m
F	The set of all described functions in the system, as discussed in Section 5.1.4
f	A specific function from F $(f \in F)$
uv_f	The user-assigned user value value of function f
s_f	The user-assigned stability value of function f
$isLinked(m, f)$	A predicate that returns $true$ when method m is linked to f or $false$ otherwise
F_m	The set of functions that are linked to method m, or, $F_m = \{f \in F \mid isLinked(m, f)\}$
M_f	The set of methods that function f is linked to, or, $M_f = \{m \in M \mid f \in F_m\}$

TABLE 5.1: Overview of symbols used in UPSS equations.

5.2.2 Contributors to the usefulness of a test

Examples of factors that could contribute to the usefulness of a test are:

- fault-proneness of the code under test,
- importance of the functionality that the code under test supports,
- code coverage achieved by the test.

In interviews (Appendix Section D.1) we also asked developers what they thought would be important factors to prioritize tests on. From these interviews we deduced that these developers thought that the importance of the code (and whether it was used by a lot of other code), the fault-proneness of the code, and the value of the functionality to the user are the most important factors for prioritizing tests. By linking methods and the function(s) they belong to, we aim to cover the factor of how often code is used by other code; if a method supports multiple functions, all these functions are considered for the priority of the test testing this method, as is further discussed in Section 5.2.4.

`a` is called by multiple other methods that are linked to different functions, method `a` inherits the links to all these functions from the other methods.

We will discuss the possible contributors to usefulness and justify our choices on if and how we incorporated them into UPSS below.

Fault-proneness

An often-used indicator for test priority is the fault-proneness of the code that is under test. Code that is deemed to be more fault-prone has a higher chance of containing a defect that would disrupt the usage of the program by a user. As discussed in Section 3.2, there are several methods for estimating the fault-proneness for a method, class, or module. In this research we do not have to re-invent the wheel, and instead we can rely on the findings of other researchers. An often-used approach is looking at the complexity of the code, derived from one or more code metrics. These code metrics could be measures such as the LOC, cyclomatic complexity, or Halstead complexity measures. Another approach is looking at historical data on the code. For example, one could suggest that code that had more bugs in the past, is more likely to contain bugs in the present or future. However, we chose to base our fault-proneness metric purely on the LOC of the code. Studies have shown that the size of a method is highly correlated with its complexity[23], and thereby its fault-proneness. A small improvement could possibly be achieved if historical data is included, however doing so has a major drawback in that incorporating such a metric into a strategy (and thereby tool) that is meant to be as easily applicable as possible could be hard. Not all projects use the same VCS, not all projects use the same issue tracking software (that is likely needed to be able to gather information about bugs and their related commits), and not all developers label their commits and issues the same, etc. Also, we think that introducing a metric that is based on historical data could skew the prioritization of suggested tests to giving higher priority for tests for code that has been in the codebase for a longer time. Especially in a project that is still in active development (such as the case study projects of this research) this could lead to improper prioritization. While this effect should be investigated and could possibly be minified by using a kind of normalization technique, this is not the focus of this study, and is therefore not included. Future work could possibly improve on the fault-proneness estimation in UPSS.

For unit tests, the LOC (and thereby the estimated complexity and fault-proneness value) of the code under test is the LOC of the specific method that the unit test should test. The exact definition of LOC to use, and thereby whether to include, for example, lines blank lines or lines with only comments on them, is left up to the user of UPSS. It is obviously important that a user is consistent in their choice of metric, as otherwise comparisons between methods and functions would not be fair.
We suggest counting the total number of lines from the start of the body of the method (e.g., the starting bracket `{`) to the last line of the method body (e.g. the closing bracket `}`). We suggest using this metric due to its ease of implementation in a automated tool (no need for further parsing of the lines to discover its contents, and the start and end lines of a method body are often given by a code parser, if one is used) and it not including the documentation that precedes a method (e.g., javadoc), since its length can vary significantly without adding/substracting from the method's complexity/fault-proneness.

For functional tests, the estimated complexity of a function is calculated by summing the complexity of all the methods linked the function, as in:

$$C_{sum}(f) = \sum_{m \in M_f} c_m$$

The complexity of these methods is calculated in the same way as for unit tests. If done manually it may seem cumbersome to manually keep track of all methods involved with the implementation of a function. In Chapter 6 we explain how we approach this issue for automated implementations of UPSS.

Importance of functionality

As discussed, we wanted to factor in the importance of functionality into the test prioritization. We looked into existing literature and found some test prioritization approaches that incorporated some kind of subjective priority/value/importance value[46, 47, 36, 29]. These approaches all assign a numerical value (that is on some scale) to functions that indicates a measure of how important the function is to the customer/end-user. The scales for the numerical values that these approaches use differ. Srikanth et al. and Kavitha et al. use a scale of 1 to 10, while Panda & Mohapatra use a scale of 1 to 3. Neither of the papers gives a justification for the scale chosen. For UPSS we do not enforce a specific scale that the numerical value should be on. The only requirement is that the user values of all functions are on the same scale.

We looked into ways that user value could automatically be extracted from source code without additional user input, but found no suitable approach. A possible approach could be basing user value on how often the function was actually used by end-users, and gathering this use count by, e.g., instrumenting the code such that method calls are logged. However, this approach has at least a few drawbacks. For one, implementing such instrumentation could cost a lot of effort. Secondly, the application would have to be actively used before meaningful data can be gathered. The Java project, for example, is only ran infrequently and with possibly different goals each time. This could mean that the possibility of gathering data is far apart and that data from one run of the application is not representative for other runs. Thirdly, use count does not necessarily represent importance to the user. For example, a web application could have a small feature (such as displaying a different welcome message everyday) that is used often, but the feature not working would not result in significant disturbance to the end-user. In one of the interviews (Appendix Section D.1.3) the interviewee also mentioned that he thinks that the importance of functions or code cannot be infered from the codebase.

We decided on an approach where a numerical value is assigned to a function to indicate its importance to the user. We call this the user value of the function. This value should be assigned to the function by someone who has knowledge of how the application is used by users. In Scrum settings this would most likely be the Product Owner.

Code coverage achieved

A metric that could be used to prioritize tests is the code coverage that the tests achieves. However, it is hard to impossible to estimate beforehand how much code coverage a test is actually going to achieve, especially for functional tests. Also, since code coverage is a function of the amount of code, the metric would give similar results to LOC, the fault-proneness metric that we chose to use. For these reasons it was decided not to include code coverage in the test prioritization.

5.2.3 Effort estimation

We want to consider both the effort required to implement a suggested test, as well as the effort that is then required to maintain this test. We discuss these items seperately.

Test implementation effort

Studying existing literature we found that test implementation effort estimation methods either were designed to only estimate the effort of implementing unit tests, or, if they were able to estimate the effort required for implementing functional tests, they required elaborate requirement specifications to do so. Since with this research we wanted to both be able to suggest unit & functional tests, as well as have no dependency on elaborate requirement specifications (as explained in Section 4.1.2), there was no existing work that was directly suitable for us. In seperate subsections per type of test, we describe the approach we designed for UPSS to estimate test implementation effort.

The goal is not to produce a definite effort estimate in, e.g., number of workhours, because estimating actual workhours is too dependent on the specific software project. Factors that could influence the number of workhours needed are, a.o., the software stack used, the existence of already developed tooling, and the experience/productivity of a developer tasked with implementing the test. An approach used by, a.o., Test Point Analysis[51] is to introduce productivity and environmental factors. For the productivity factor, they require the TPA user to keep track of historical data to guesstimate a productivity factor for a new project. For the environmental factor, they list 7 contributing factors (such as "Test tools" and "Development environment"), each with classes with corresponding ratings. A TPA user should pick one of the classes per factor, and use the corresponding rating in further calculations. Our critique on this approach has multiple aspects. Firstly, historical productivity data might be unavailable or inaccurate. In such cases, incorporating this productivity factor produces no accurate real-world estimations, and solely serves as a scaling factor on the effort estimate that is applied to all suggested tests. Secondly, the system with classes with specified ratings used with the environmental factor limits the level of granularity that a user can provide. Thirdly, the ratings specified in TPA's original paper seem arbitrary and are not well-explained. This leads to a feeling that while the ratings could be well-suited for some projects/organizations, they could well be unsuitable for others. This observation is also shared in [10] and [31]. Finally, incorporating factors such as TPA's environmental and productivity factors would increase the effort necessary to apply UPSS. The aim with UPSS was to create a strategy that is as lightweight as possible, such that the barrier to usage is as low as possible. Therefore, we deemed it undesirable to include factors that we deemed not fully necessary. Users of UPSS are obviously free to nevertheless use a method similar to the one used by TPA to convert UPSS's effort estimates into estimated workhours.

Rather than estimating workhours, we want to be able to give estimates per suggested test such that they can be compared to each other. We will first describe how UPSS estimates the effort required to implement unit tests, and will then discuss the estimation of effort required for functional tests.

Unit tests For unit tests, we chose to base the effort estimate on the complexity of the code under test (the "internal" code). As described in Section 5.2, we decided to use LOC as estimate for the complexity of code, and thereby for the implementation effort required. This is the same approach as used by Shihab et al. in [43]. Toure et al. in [50], Bruntink & van Deursen in [13], and Terragni et al. in [49] observe that for unit tests the LOC of the test implementation highly correlates with the LOC of the internal implementation, and is a good indicator for the amount of effort required for implementing the associated unit test. It is also shown that this observation is independent of the software stack or testing tools used[49, 50], meaning that the observation holds for most (if not all) software projects. It should be noted that these observations were made on class-level, and we are suggesting tests on method-level. However, it is the best (simple) indication that we could find, and therefore the one that we have chosen/

Functional tests We looked into ways to estimate the amount of effort required to implement a functional test. However, we could not find one suitable for UPSS. It is not easy (and maybe even not possible) to estimate effort required for implementing functional tests based on source code alone. A function could be extremely complex under the hood, but validating its result could be as easy as asserting whether one value matches an expected value. As discussed in Section 3.3.1, approaches such as TPA[51] exist, which attempt to estimate the amount of effort required for black-box testing, but require a significant amount of effort to be applied, which is not desired for UPSS.

For these reasons we have chosen not to estimate implementation effort for functional tests. Future work could further examine the possibilities of estimating the effort required to implement functional tests with limited effort. A possible idea would be to introduce an extra user-assigned value at function definitions that would indicate implementation effort in some way. Due to time constraints this idea was not further developed for UPSS.

Test maintenance effort

We were not able to find any relevant literature on specifically the estimation of effort required to maintain automated software tests. An idea that came up in discussions with the Product Owner of the Java project, was to include a stability factor in the test prioritization as a measure on how much the function and/or its code is going to change in the future. A high stability value would mean that the function and code does not significantly need changing for a long time, while a low stabilty value would mean the opposite. An example of such low stability would be when a function is only temporarily built into the application for demonstration purposes, or when code is planned to be refactored for technical reasons. The idea behind this factor is that if a function and/or its code will change soon, it makes less sense to create tests for it *now*. Effort would be spent now to implement a first edition of the test(s), and effort would have to be spent again to refactor the test(s) when the code is changed, for example. That effort could be better spent on functions/code that stay the same for a long time.

We decided to have users of UPSS assign a stability value to functions, in a similar way as user value. This does have the drawback of combining the stability of the function and the stability of the code into one value. This means that to come to a stability value, input is required from both the developers of a project, as well as those that decide on functionality (e.g., Product Owners). Also, it could be that a core underlying part of the codebase is up for refactoring, while the functions stay stable. In this case a user of UPSS would possibly have to decrease the stability value of all functions that use this part of the codebase. We have thought about having a seperate stability value that can be assigned to pieces of code (e.g., methods or classes), but decided against doing so since we think that assigning these values would require too much effort.

Just as with user value, we do not enforce a specific scale that the numerical value should be on for UPSS. Again, the only requirement is that the user values of all functions are on the same scale. We recommend that the user value and stability values are placed on the same scale, such that they have an equal influence on the final test priority value when weighted the same. We will cover the topic of weighing factors in Section 5.2.5.

5.2.4 Combining functions

It is possible that a method is linked to multiple functions. This means that there are multiple functions that rely on the method for the function to be fulfilled. When a method is linked to multiple functions, there is no longer a one-to-one relation between a function (and its user value and stability values) and the method. However, for such a method, we still want to somehow determine a value for its user value and stability, derived from the functions it contributes to, such that these values can be used for the prioritization of its unit test. To do this, we need to combine the aforementioned functions' values in some way. There are several ways this could be done, and we will now discuss them and what we have chosen and why.

User value

In this section we discuss how user value of functions can be combined for a method.

Mean Probably one of the most intuitive approaches for combining user value would be to calculate their mean, as in:

$$UV_{mean}(m) = \frac{1}{|F_m|} \sum\nolimits_{f \in F_m} uv_f$$

However, we think that this approach is not suitable because we think that its results would not be representative of what one would expect. An example to show why is: imagine that a method currently serves one function that has a user value value of 5. The method would than also have a UV_{mean} of 5. Now a new function is introduced to the application, and this function also uses the same method. The new function is however a small feature that only benefits one specific customer, and is therefore given a user value value of only 2. This would lead the method to have a new UV_{mean} of 3.5. This means that the indicated user value of the method has gone down, even though it has only been used *more*, and also still serves its original function. For this reason, we chose not to use the mean of a method's functions' user value. The same reasoning also applies to approaches using the mode or median, instead of the mean.

Maximum To address the issue discussed above, we could assign a method only the maximum value of its linked functions' user value:

$$UV_{max}(m) = max(\{uv_f \mid f \in F_m\})$$

This way, a method's indicated user value would not go down when a new function is linked to it. However, we again think that this approach is not suitable and show an example why. Imagine a situation with two methods with linked functions such as in Figure 5.1. With UV_{max} these two methods would have the same indicated user value. However, if `Method 1` is not working correctly, undesired behavior could be apparent in three different functions of the application (for example), while if `Method 2` is not working correctly, only one function could show undesired behavior. We think that therefore it is more important for `Method 1` to be working correctly than it is for `Method 2`. For this reason, we chose not to use this approach.

Direct summation An approach that mitigates the issue described with UV_{max} is to sum the user value values of all functions the method contributes to, as in:

$$UV_{sum}(m) = \sum\nolimits_{f \in F_m} uv_f$$

Method 1	
Function name	**User value**
Function a	3
Function b	2
Function c	3
UV_{max}(`Method 1`)	**3**

Method 2	
Function name	**User value**
Function x	3
UV_{max}(`Method 2`)	**3**

FIGURE 5.1: A comparison of two methods, when the maximum of its functions' user value is taken

While this could be a suitable method, we feel that it has a major drawback. A method that contributes to a lot of "low-user-value" functions, could quickly outscore a method that contributes to (few) "high-user-value" functions. An example of this is given in Figure 5.2. In this example, `Method 2` gets a lower combined user value score than `Method 1`, even though `Method 2` serves two functions that are both highly important to be working correctly, and of which one has the highest available user value. We think that this is not right, since (depending on how the Product Owner assigned user value values) `Method 2` not working correctly could lead to the entire application not functioning, while `Method 1` not working could only lead to some inconveniences. Therefore we decided no to use this method.

Method 1	
Function name	**User value**
Function a	2
Function b	1
Function c	2
Function d	1
Function e	2
Function f	2
UV_{sum}(`Method 1`)	**10**

Method 2	
Function name	**User value**
Function x	5
Function y	4
UV_{sum}(`Method 2`)	**9**

FIGURE 5.2: A comparison of two methods, when its functions' user value are summed directly

Fibonacci-based summation An approach to mitigate the issue described with UV_{sum}, while keeping its benefits, is to sum the user value values, but using a method of scaling the values such that the higher user value values "weigh" disproportionately more than the lower ones. There are a plethora of ways to do so (e.g., raising the values to some power), but we chose an approach using the Fibonacci sequence[44] inspired by the Scrum's planning poker[57, 24]. This leads to the following equation:

$$UV_{fib}(m) = \sum_{f \in F_m} Fib(uv_f + 1)$$

with

$$Fib(n) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \\ Fib(n-1) + Fib(n-2) & \text{otherwise} \end{cases}$$

This gives the following relation between user-assigned user value and the value used in the summation:

User-assigned value	1	2	3	4	5
Value used in summation	1	2	3	5	8

The idea of using the Fibonacci sequence came from a discussion with the Product Owner of the Java project. The Fibonacci sequence is often used in planning poker with the reasoning that the higher the estimate, the larger the uncertainty. For example, it is easier to estimate whether a task is going to take 2 hours or 3 hours than it is to estimate whether a task is going to take 13 or 14 hours. As estimating the user value of a function is also a non-exact task, it felt appropriate to use this approach. Future work could evaluate the usage of different scaling approaches and different approaches to combining function values.

When we apply UV_{fib} to the example from Figure 5.2, we get the results in Figure 5.3. We can see that with UV_{fib} `Method 2` now has a higher calculated user value than `Method 1`. Obviously, there is a point where the sum of many "low user value" functions is still larger than a few "high user value" functions, but we think that using the Fibonacci sequence strikes a proper balance.

`Method 1`

Function name	**User value**
Function a	2
Function b	1
Function c	2
Function d	1
Function e	2
Function f	2
UV_{fib}(`Method 1`)	**10**

`Method 2`

Function name	**User value**
Function x	5
Function y	4
UV_{fib}(`Method 2`)	**13**

FIGURE 5.3: A comparison of two methods, when its functions' user value are summed after being converted to their Fibonacci counterpart

Stability

For the stability values of a method's linked functions the same considerations as for their user value values needs to be made.

Summation Summing the functions' stability values would give

$$S_{sum}(m) = \sum\nolimits_{f \in F_m} s_f$$

We think that summing the stability values is not a suitable approach. Doing so would increase the stability of a method when it is used by more functions, and we feel that this would not be the case in practice.

Minimum or maximum Taking the minimum or maximum value of the functions' stability values, as in

$$S_{min}(m) = min(\{s_f \mid f \in F_m\}) \text{ or } S_{max}(m) = max(\{s_f \mid f \in F_m\})$$

also does not feel appropriate, as we think that this also does not correctly reflect what happens in practice. Imagine a method that is currently only linked to a function with a stability value of 5, and there are no thoughts of refactoring/removing the method any time soon. Then, a new function is added that is purely for a short trial and is due to be removed again quickly. This new function would likely get a stability value of only 1. If this function is then linked to the aforementioned method, and the minimum value of a method's functions' stability values is used as the method's stability indication, this would suddenly indicate that the method is also highly unstable, even though it has not changed and will probably still stay the same for a significant amount of time.
A similar reasoning discounts using the functions' maximum stability value as the stability value for a method. Imagine a method is in a part of the codebase that is planned to be entirely refactored soon, and therefore the functions of this part of the codebase all have a stability value of 1. If now from a different part of the codebase the aforementioned method happens to be used by a function with a high stability value, this does probably not mean that the method will no longer be a part of the planned refactor. We think that most likely the developers will either implement a new method to serve the high stability function, or adapt the high stability functions' methods to use the refactored version of the method.

Mean All in all, we came to the conclusion that we lack context of the reasoning behind a function's stability value, and that therefore no perfect approach for combining functions' stability values exist. We think that the best approach that we can take is to use the mean of the functions' stability values, as in:

$$S_{mean}(m) = \frac{1}{|F_m|} \sum_{f \in F_m} s_f$$

We realise that this approach is not perfect, but we think that it is the best that can be done without having more context of the reasoning behind a function's stability value. .

5.2.5 Combining test usefulness & estimated implementation effort

In our strategy, we combine the usefulness of a test and its effort estimate to get a final priority value for a test. This value can then be used to rank the suggested tests and decide on what tests should be implemented first.

We want to be able to weigh the different factors of the priority calculation differently, because we expect that different projects attribute different weights to the factors of the priority calculation. A project that is more focused on innovation and is used to demonstrate new functionality to a group of test users, could accept having some bugs, as long as these bugs are not in the *most important* functions. A project that is already running in a production environment and has a large group of paying customers might rather focus on having as few bugs as possible. The former project could then assign a higher weight to the tests' user value, while the latter project could assign a higher weight to the tests' complexity. Therefore, UPSS uses four weighing factors:

- W_{uv}, to influence the weight of the tests' user value values,
- W_c, to influence the weight of the tests' complexity values,
- W_s, to influence the weight of the tests' stability values,
- W_e, to influence the weight of the tests' effort estimations.

Combining the choices explained in the previous sections and the weighing factors leads to Formula 5.1 (with some parentheses for legibility).

$$P(t) = \begin{cases} \dfrac{(UV_{fib}(m))^{W_{uv}} * (c_m)^{W_c} * (S_{mean}(m))^{W_s}}{(c_m)^{W_e}} & \text{if } t \text{ is a unit test for method } m \\ (uv_f)^{W_{uv}} * (C_{sum}(f))^{W_c} * (s_f)^{W_s} & \text{if } t \text{ is a functional test for function } f \end{cases} \tag{5.1}$$

It should be noted that the weighing factors can also be given negative values. This works as one might expect, inverting the effect that the respective factor has on the calculated priority value.

Because of the inability to estimate functional test implementation effort and the choice that therefore was made not to include effort in Formula 5.1 for functional tests, it is not possible to compare test priorities between unit tests and functional tests. We had hoped to do so, but were not able to because of the reasons discussed in Section 5.2.3.

An instance of UPSS with specific weighing factors is represented as UPSS(W_{uv},W_c,W_s,W_e). For example, an instance that gives a higher weight to user value and disregards implementation effort could be UPSS(2,1,1,0).

We chose to multiply/divide the factors, rather than adding/substracting them (as is done in, for example, [56]). Adding/substracting would only work if all factors are on the same scale, e.g. all factors give a value between 1 and 5. However, with UPSS this is not the case, as for example an absolute value is used for a test's complexity value. Since multiplication is used, the weighing factors cannot be simple scaling factors that multiply a value with the scaling factor (e.g. $c_m * W_c$). If this was done the combination of scaling factors could be factored out of the equation so that:

$$P(t) = \frac{UV_{fib}(t) * W_{uv} * c_m * W_c * S_{mean}(t) * W_s}{c_m * W_e}$$

would be the same as

$$P(t) = \frac{W_{uv} * W_c * W_s}{W_e} * \frac{UV_{fib}(t) * c_m * S_{mean}(t)}{c_m}$$

and thus the combination factors would simply scale all tests' priority value, instead of acting upon the individual factors as desired. Therefore, the factors are raised to the power of their respective weighing factors, leading to Formula 5.1.

Chapter 6

Code annotations

We described how we need some way of explicitly defining functionality (with user value and stability values) in a codebase, such that this information can be used for suggesting tests with UPSS. Also, we described how in UPSS we need to know which method(s) serve to fulfill which function(s), such that the test prioritization formulas can use the appropriate values. We have designed an approach that uses code annotations to fulfill the tasks of defining functionality and linking functions with methods.

6.1 Overview

Code annotations are code decorations that can be used by tooling, but have no influence on the compilation or execution of the program. They are commonly used to signal specific information in a specific manner. A common use case of such annotations is to document a method with a description of the method and a description of the parameters it takes and the value it returns. Examples of standards for such code annotations are javadoc[1] and JSDoc[2]. Javadoc and JSDoc are popular standards for annotating Java and JavaScript/TypeScript code, respectively[48, 55]. In these formats there is the possibility of using tags, often with accompanying values, to describe certain aspects of the documented code. Listing 6.1 shows an example where the javadoc `param` and `return` tags are used to describe the parameters the method takes and the value it returns, respectively.

For our tool we chose to extend existing code annotation standards with new tags. Using this approach has benefits for both us as well as for users of our tool. Because of the common usage of standards such as javadoc, we can use existing tools to extract the information from codebases, and users can easily integrate the new tags in documentation they might already have. This latter benefit means that source code can remain tidy, and the functionality information does not clutter the codebase more than necessary. We also had no reason to develop an entirely new annotation standard, since all of our needs could be fulfilled through extending existing tooling.

Because the case study projects concern software programs written in Java and TypeScript, focus on javadoc and JSDoc was enough for this research. However, similar solutions exist for other programming languages, such as docstrings[3] for Python, documentation

[1] `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`
[2] `https://jsdoc.app/`
[3] `https://peps.python.org/pep-0257/`

```java
public class HelloService {

    /**
     * Greets the user.
     *
     * @param name the name of the user to greet
     * @return a greeting to the user
     */
    public static String greetUser(String name) {
        return "Hello " + name "!";
    }

}
```

Listing 6.1: Example of javadoc usage

comments[4] for C#, and Doxygen[5] for C and C++, a.o.

6.2 Newly introduced code annotations

We want users to be able to define functions in two ways:

1. define a function that is linked to only one method and can be defined in the documentation of this method itself,
2. define a function centrally and link to it from one or more methods.

We introduce the `function` and `functionDef` tags. The `function` tag is meant to be placed directly with a method to define functionality that is implemented only by this one method. It should be used in the following way:

```
@function <description> <user value> <stability>
```

Listing 6.2 shows the previous javadoc example from Listing 6.1, now extended with the definition of the function provided by the method. In this example, the function is given a user value of 3 and a stability value of 5.

It might seem that the already existing description of the method (`"Greets the user."`) could be used as description of the function, and therefore no further description definition would be necessary. However, we cannot trust these method descriptions to exist or to be a description of the functionality provided (they might instead contain a more formal definition of what the method does). Also, this inference of functionality through existing documentation would not work for situations where one function is fulfilled by multiple methods.

`@functionDef` is meant for such situations where multiple methods support one function. `functionDef` would be used in a central place to define the function, and afterwards `@function` can be placed at a method, refering to the previously defined function and

[4]https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/documentation-comments

[5]https://www.doxygen.nl/

```
public class HelloService {

    /**
     * Greets the user.
     *
     * @param name the name of the user to greet
     * @return a greeting to the user
     * @function "Greet a user" 3 5
     */
    public static String greetUser(String name) {
        return "Hello " + name "!";
    }

}
```

Listing 6.2: Example of javadoc usage with the `function` tag

linking the method and function. To be able to be referred to, an identifier for the function is required besides its description and user-assigned values, therefore its syntax becomes:

```
@functionDef <identifier> <description> <user value> <stability>
```

An example of using `functionDef` is:

```
@functionDef userLogin "Login the user" 5 3
```

This function could then be referred to by placing a `function` tag with only the function's identifier as value at amethod, such as:

```
@function userLogin
```

Obviously, users are free to use the approach of defining a function centrally and then referencing to it also for a function that only depends on one method.

6.3 Entry points

Annotating every method in a codebase would be cumbersome. Therefore we introduce an approach using "entry points". We define entry points as the methods at the highest level in an application, where requests/commands from a user enter the application. A common example of such an entry point is a method that receives calls to endpoints in web servers. These entry points are thus the places where functions "start". This means that by analyzing which methods are called by an entry point, and then which methods are called by those methods etc., we can derive all methods that are used to fulfill a function. Methods inherit the function links of the method(s) that call them. This analysis of the function call stack can be done by, for example, exploring the program's Abstract Syntax Tree (AST).

A simple example of such an entry point with a linked function and underlying methods is shown in code in Listing 6.3 and graphically in Figure 6.1. In this example the `HelloController.hello` method is the entry point, and it is directly linked to the function through the `@function` annotation. The `HelloController.hello` calls the

`Util.parseName` method, and this latter method is therefore indirectly linked to the "Greet the user" function, as this function depends on both methods to be fulfilled. An example of a method inheriting different functions from methods that call it is given in Listing 6.4 and Figure 6.2.

```
class HelloController {

    /**
     * Greets the user.
     *
     * @param name The user's name
     * @return The user's name prepended with "Hello"
     * @function "Greet the user" 2 4
     */
    public String hello(String name) {
        return "Hello␣" + Util.parseName(name);
    }

}
```

Listing 6.3: Example of a method that is directly linked to a function and calls an underlying method

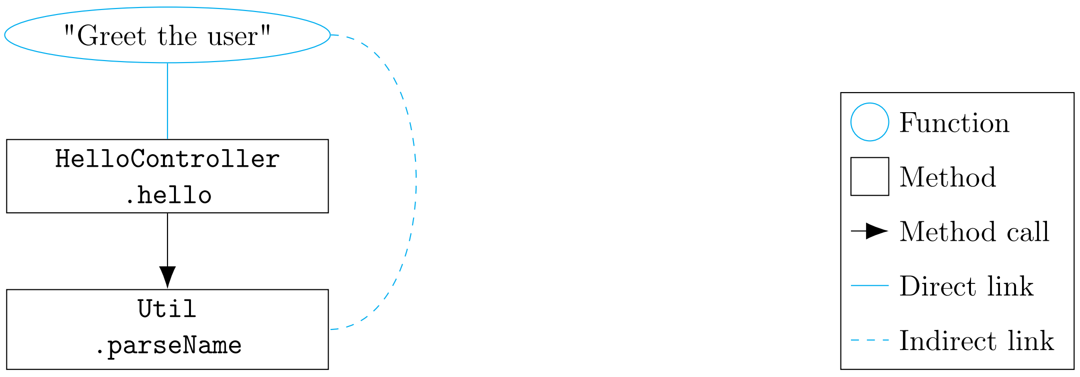


Figure 6.1: Method/function graph of the method shown in Listing 6.3

Entry points can serve multiple functions, such as in the example in Appendix Section A.2. In addition, one function can depend on multiple entry points, such as in the example in Appendix Section A.4. A practical example of this latter case is a web application wherein to update a seting, first the current value of the setting needs to be retrieved, and thus two calls to the back end need to be made (one to retrieve the value, one to update the setting to the new value).

```
class AuthorsController {

    /**
     * Get information of author.
     *
     * @param authorId The author's ID
     * @return The author
     * @function "Show author information" 4 5
     */
    @GetMapping("/author/{authorId}")
    public Author getAuthor(@PathVariable int authorId) {
        return authorsService.getAuthor(authorId);
    }

}

class BooksController {

    /**
     * Get books written by author.
     *
     * @param authorId The author's ID
     * @return A list of all books written by the author
     * @function "Show author books" 3 3
     */
    @GetMapping("/author/{authorId}/books")
    public List<Book> getBooksForAuthor(@PathVariable int
       ↪ authorId) {
        Author author = authorsService.getAuthor(authorId);

        return author.loadBooks();
    }

}
```

LISTING 6.4: Example of a method that inherits multiple functions from methods that call it

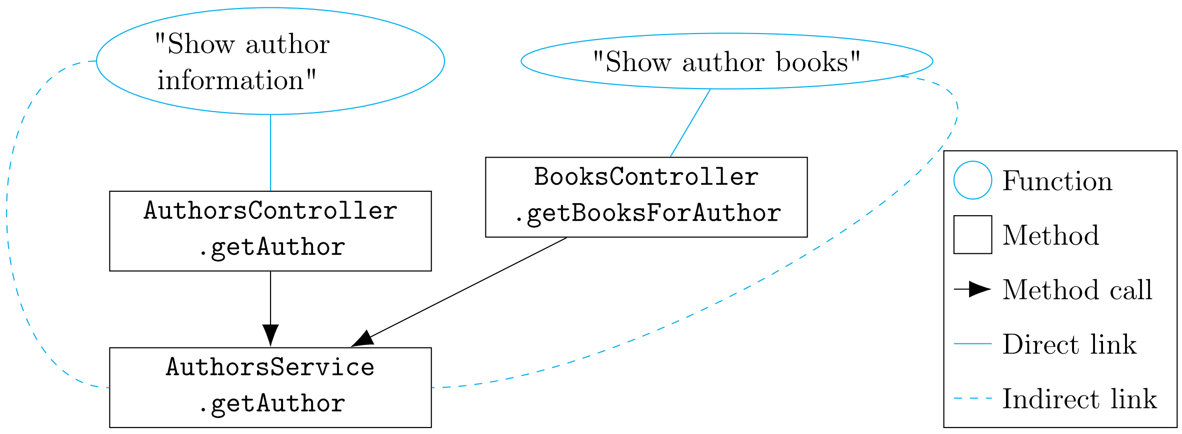

"Show author information"
"Show author books"
AuthorsController .getAuthor
BooksController .getBooksForAuthor
AuthorsService .getAuthor
Function
Method
Method call
Direct link
Indirect link

FIGURE 6.2: Method/function graph of the method shown in Listing 6.4

Chapter 7

Strategy implementation

This chapter introduces our implementation of UPSS. The implementation consists of three different programs, that together make applying UPSS possible. We called the suite of programs together Test Indicator for Existing Software (TIES), as it gives indications for which tests maintainers of a software program should implement.

We first give an overview of the programs, and then go into detail on each one of them.

7.1 Overview

TIES has to fulfill the following objectives:

- suggest tests based on a given codebase,
- prioritize the suggested tests in a configurable manner,
- display the prioritized suggested tests to a user.

We find it important that our approach is applicable to as many software projects as possible, and therefore it has to be able to be adapted to specific software stacks easily. To make the tool usable for different types of codebases (e.g., different programming languages) and different future use cases, it was set up in a modular fashion. The different independent software programs making up the tool are:

- a program to extract method and function information from a codebase (Code Analysis Tool (CAT)),
- a program that uses the method and function information to suggest tests and calculate the test prioritization factors for the tests (Test Suggestion and Scoring Tool (TSST)),
- a program performs the final prioritization calculation and displays the prioritized tests in a human readable manner (Tests Graphical User Interface (TGUI)).

This way, only CAT would need to be adapted for specific programming languages. An overview of the programs is shown in Figure 7.1.

Source code of our implementations of these programs is available on `https://github.com/TiesB/TIES`. A user would use the programs of TIES in the same order that they are presented here. After incorporating the code annotations into their codebase, the user would first run an implementation of CAT to get the JSON file containing method and function information. In our implementations the user has to adapt CAT's configuration file to let CAT point at the desired codebase. After adapting TSST's configuration file to point the tool at CAT's output file, TSST can be run. Finally, after serving TGUI (e.g., with a web server or with `ng serve`[1]), TSST's output JSON file can be uploaded to TGUI through its graphical interface. Then, the results will show and the user can adapt weighing factors if desired.

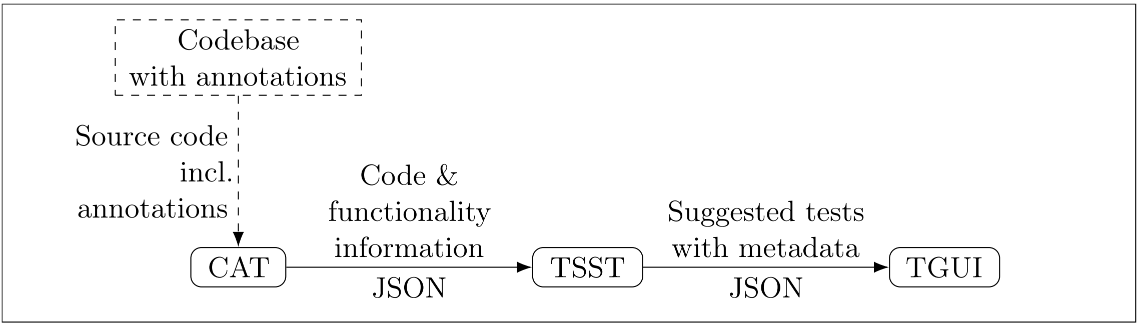

Codebase with annotations
Source code incl. annotations
CAT
Code & functionality information
JSON
TSST
Suggested tests with metadata
JSON
TGUI

FIGURE 7.1: An overview of the programs forming TIES.

7.2 Code Analysis Tool

The first program in the chain (CAT) extracts method and function information from a given (part of a) codebase. For every method, this information consists of:

- a unique name,
- the complexity value of the method,
- other methods the method calls, and thus depends on,
- the functions the method contributes to according to its annotation(s).

To make a method's name unique, it can, e.g., be prefixed with the name of the file, class and/or module/package it is located in. If no unique name can be infered for a method, the program should fail. If the program is not able to differentiate between methods with the information it has, TGUI would also not be able to display different titles for the unit tests that are suggested for the methods. This would lead to the user not definitively knowing for which method to implement the test.

For every defined function the information consists of:

- a unique name,
- the user-given description,
- the user-assigned user value,
- the user-assigned stability.

The unique name has two purposes: firstly, when a function uses multiple entry points, the annotations on these entry points need to refer to the same function. This function can then be defined somewhere else (e.g., at the class-level). When a function is defined for

[1]https://angular.dev/tools/cli/serve

only one entry point, the unique name does not need to be referenced by the user, and therefore does not have to be shown or be user-provided. It can instead be derived from the user-given description. For example, an identifier for the function with the description "List all books by specific author" can be "`list_all_books_by_specific_author`".

Since the validation of this research is performed with Java and TypeScript codebases, two seperate implementations are created: one in Java that uses JavaParser[2] for code parsing, data extraction and method call resolution, and one in TypeScript that uses the TypeScript compiler API[3] for code parsing, data extraction, and method call resolution. The output of the program is a file containing JSON[4] that is formatted according to the JSON schema[5] specified for the required format of the input of TSST. This JSON schema is given in the listing in Appendix Section C.1. Example output is given in Listing 7.1.

```
{
    "methods": [
      {
        "name": "example.ts_GreetingsController_hello",
        "length": 4,
        "functionalities": ["Greet_a_user"],
        "dependencies": ["example.ts_UsersService_findById
           ↪ "],
        "dependents": []
      },
      {
        "name": "example.ts_UsersService_findById",
        "length": 3,
        "functionalities": [],
        "dependencies": [],
        "dependents": [
          "example.ts_GreetingsController_hello"
        ]
      }
    ],
    "functionalities": [
      {
        "name": "Greet_a_user",
        "description": "Greet a user",
        "userValue": 4,
        "stability": 5
      }
    ]
  }
```

Listing 7.1: Example output of CAT

[2]https://github.com/javaparser/javaparser
[3]https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API
[4]https://www.ietf.org/rfc/rfc4627.txt
[5]https://json-schema.org/

7.3 Test Suggestion and Scoring Tool

The second program (TSST) in the chain uses the extracted method and function information to suggest tests. Each test nis represented with the following information:

- the type of test (unit or functional),
- a title,
- its complexity value,
- its user value value,
- its stability value.

For functional tests, it uses the function information from the JSON input to suggest a test for each function, as described in Section 5.1.4. The description of the function becomes the test's title, while the user value and stability values are directly taken from the function definition. To calculate the complexity value for a functional test, it sums the complexity values of the methods that are linked to the function, as described in Section 5.2.2. It discovers which methods are linked to a function by recursively looking into methods that a method depends on, starting from the entry point (the method the function was explicitly linked to). An example suggested functional test is given in Listing 7.2.

```json
{
    "testType": "FUNCTIONAL",
    "title": "Greet a user",
    "userValue": 4.0,
    "stability": 5.0,
    "complexity": 7.0
}
```

Listing 7.2: Example of a functional test in JSON format

For unit tests, it uses the method information from the JSON to suggest a test for each method, as described in Section 5.1.3. The name of the method becomes the test's title, and the complexity value is taken directly from the input data. To apply the calculations given in Section 5.2, the tool needs to find out what functions the method contributes to. To do so, it again uses the dependency information gathered by CAT. When it has the complete list of functions, it applies the calculations to derive the unit test's user value and stability values. An example suggested unit test is given in Listing 7.3.

```json
{
    "testType": "UNIT",
    "title": "example.ts_UsersService_findById",
    "userValue": 7.0,
    "stability": 4.5,
    "complexity": 3.0
}
```

Listing 7.3: Example of a unit test in JSON format

The output of the program is a file containing JSON that is formatted according to the

JSON schema specified for the required format of the input of TGUI. This JSON schema is given in the listing in Appendix Section C.2.

TSST does not perform the final prioritization calculation of the tests. This functionality was intentionally left out, such that the tool that displays the tests can alter the weighing factors used. It was important for us to be able to change weighing factors in TGUI and immediately see the results. This way case study participants could easily use the tool themselves and see the effects of changing the weighing factors.
An approach where TSST takes input (e.g., from a configuration file or through a web socket connection with a user interface) for the weighing factors and is then able to perform the final prioritization calculation and return a sorted list of tests is also possible. This approach would have the benefit that the output TSST would be complete, without a dependence on a final program to perform the prioritization calculation. This approach was not chosen for the implementations created during this research due to time constraints and ease of implementation.

7.4 Tests GUI

Finally, a program was implemented for displaying the suggested tests to the user, called Tests Graphical User Interface (TGUI). We implemented a GUI for several reasons:

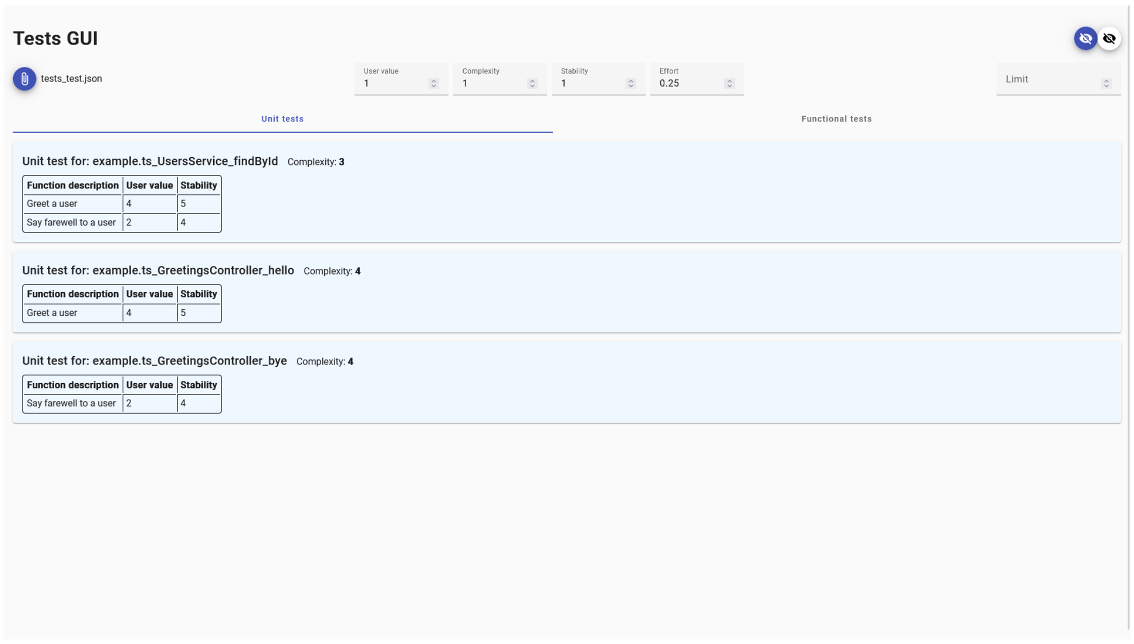
- to show suggested tests to participants of the user study and let them change weighing factors, without them having to look into more technical data,
- to be able to generate graphical representations of suggested tests that were used during different tasks in the evaluation interviews (Section 8.4).

Besides just showing the tests, TGUI also performs the final prioritization calculation for the tests (using Formula 5.1) and sorts them.

TGUI was implemented using Angular[6], a framework for web frontends using TypeScript. The JSON file generated by TSST can be uploaded directly to TGUI. TGUI will then parse it, perform the required calculations, and display the results.

To enable the generation of graphical representations of suggested tests with different amounts of information shown, two toggles were added to the interface. This way the application can be set to either show both the weighing factor settings and the tests' metadata (complexity, user value, and stability values, as well as what functions a test covers). Figures 7.2 and 7.3 show the application with both the weighing factor settings and the tests' metadata and an example list of unit tests and functional tests respectively. The aforementioned toggles can be seen in the top-right. Figure 7.4 shows the application with an example list of unit tests and the weighing factor settings hidden. This would be the view that a user would mostly use when looking at the results of UPSS. Finally, Figure 7.5 shows the application with an example list of unit tests and both the weighing factor settings and test metadata hidden. It is expected that this view would not be used by actual users, but we used it to generate items for the tasks of the evaluation interviews.

[6] `https://angular.dev/`



Unit test for: example.ts_UsersService_findById Complexity: **3**

Function description	User value	Stability
Greet a user	4	5
Say farewell to a user	2	4

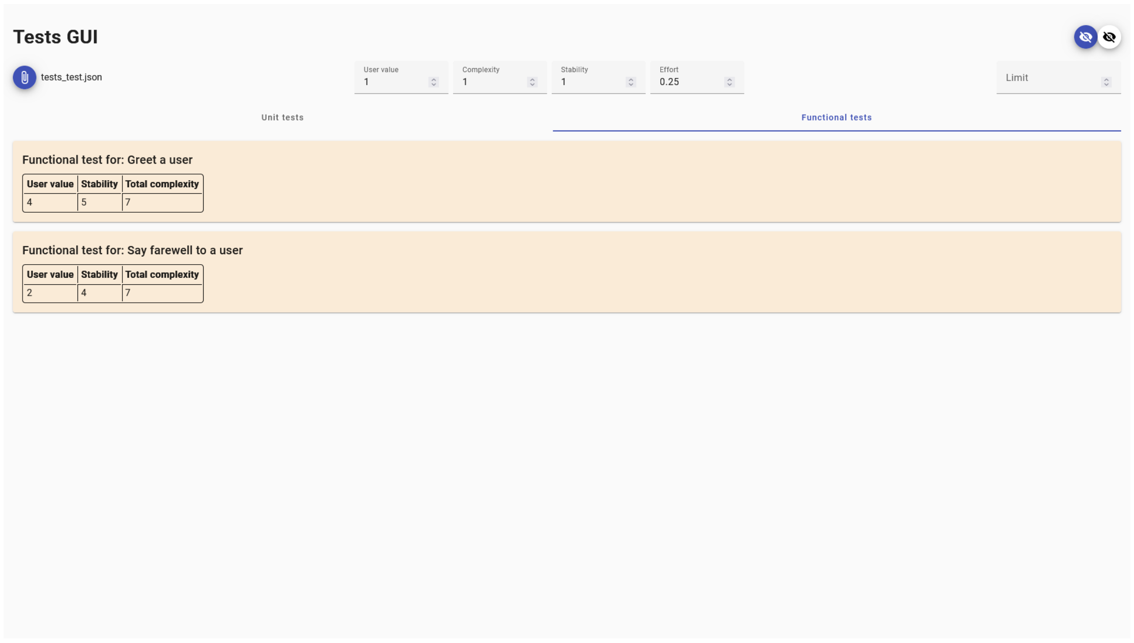
Unit test for: example.ts_GreetingsController_hello Complexity: **4**

Function description	User value	Stability
Greet a user	4	5

Unit test for: example.ts_GreetingsController_bye Complexity: **4**

Function description	User value	Stability
Say farewell to a user	2	4

FIGURE 7.2: Screenshot of TGUI showing unit tests with metadata and weighing factor configuration



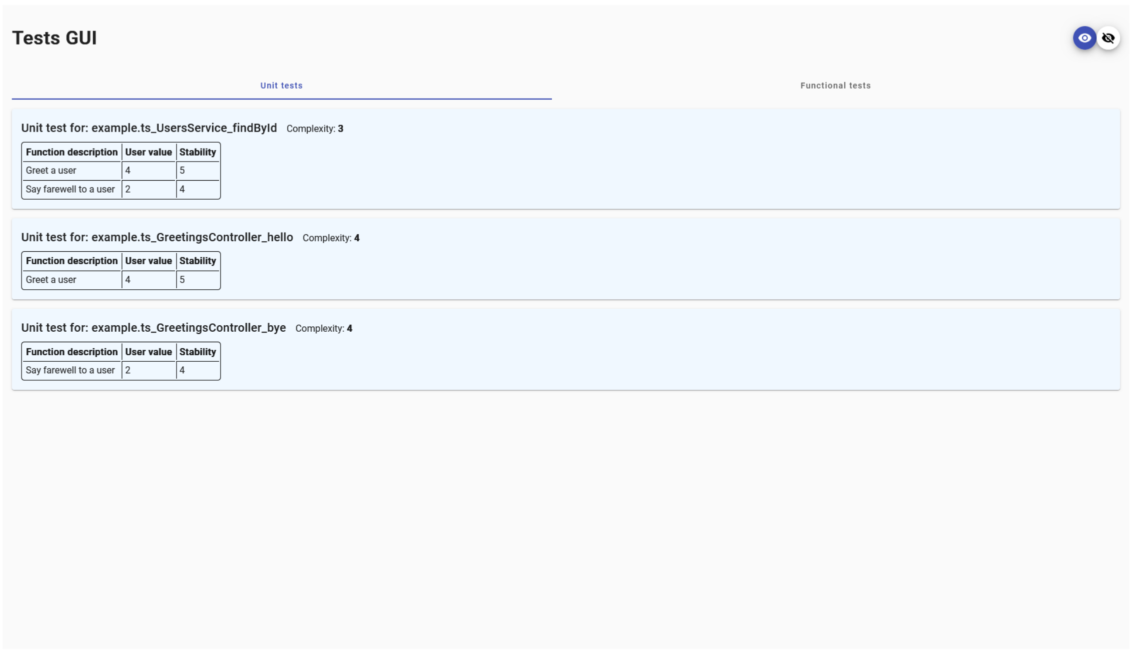
Functional test for: Greet a user

User value	Stability	Total complexity
4	5	7

Functional test for: Say farewell to a user

User value	Stability	Total complexity
2	4	7

FIGURE 7.3: Screenshot of TGUI showing functional tests with metadata and weighing factor configuration


Tests GUI

Unit tests | Functional tests

Unit test for: example.ts_UsersService_findById Complexity: **3**

Function description	User value	Stability
Greet a user	4	5
Say farewell to a user	2	4

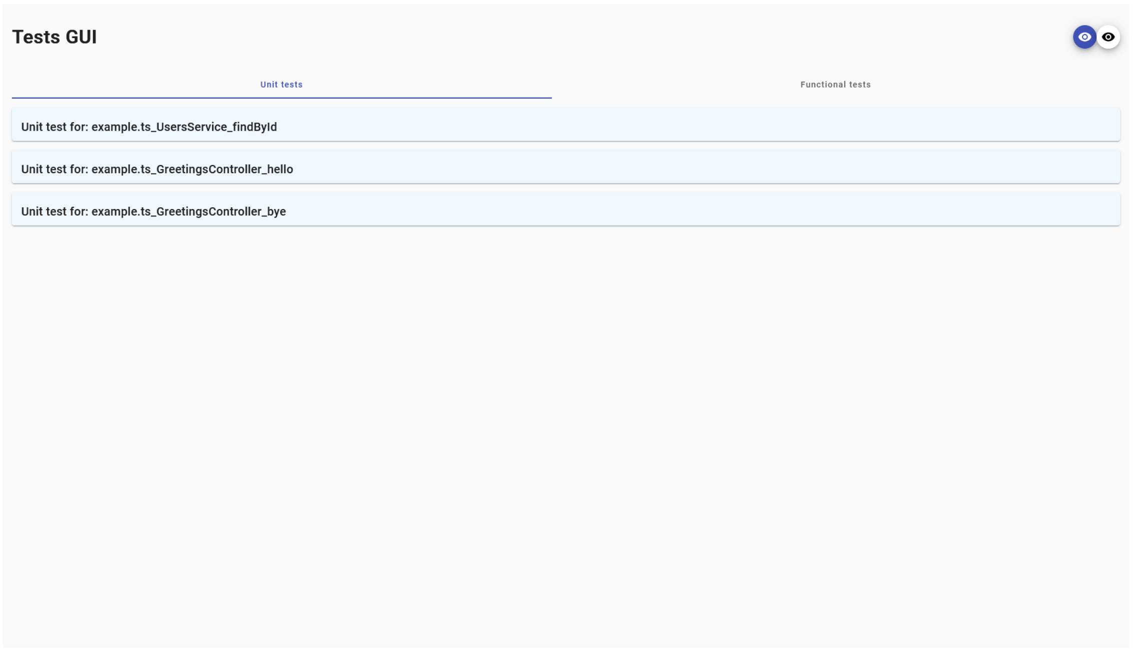
Unit test for: example.ts_GreetingsController_hello Complexity: **4**

Function description	User value	Stability
Greet a user	4	5

Unit test for: example.ts_GreetingsController_bye Complexity: **4**

Function description	User value	Stability
Say farewell to a user	2	4

Figure 7.4: Screenshot of TGUI showing unit tests with metadata


Tests GUI

Unit tests | Functional tests

Unit test for: example.ts_UsersService_findById

Unit test for: example.ts_GreetingsController_hello

Unit test for: example.ts_GreetingsController_bye

Figure 7.5: Screenshot of TGUI showing unit tests without metadata

Chapter 8

Case studies

In this chapter, we first describe the general setup of the case studies. Then we describe what the developers and product owners had to say about their experience with applying UPSS and TIES. Afterwards, we describe the outcomes of the evaluation interviews held with several developers per case study project. In Chapter 9 the results are further discussed.

8.1 Overview

We asked the maintainers of the two case study projects to apply UPSS with the code annotations of TIES to a part of their codebases. For this we supplied manuals that are attached in Appendix F. The code annotations were first added to the code by a developer, after which a Product Owner filled in the user value and stability values. It was chosen to focus only on parts of codebases, and not the entire codebases. This choice was made because of limited availability of the developers and product owners.

We also asked the developers and product owners that were providing the annotations/information to note the observations they had while performing the work. These observations could range from aspects of the approach that they liked, to problems they ran into while applying the approach. We asked them to share these notes with us. To summarize their findings, we held short interviews after their work was done focusing on their experiences. With this process Aspects 5 and 6 from Table 4.1 are covered.

Then, after the annotations were placed, we put the codebase into TIES. To cover Aspects 1, 2, 3, and 4 from Table 4.1 we held evaluation interviews with developers of the case study projects. Since these developers are acquainted with the codebase and with how they would normally write tests, we deemed their input valuable.

8.2 General setup case studies

The two case study projects introduced in Section 2.2 were available for this research. The expectation was that because of their different objectives, there would be different thoughts about the prioritization of tests. Later, in Chapter 9, we discuss the observed differences and similarities.

In both projects, a developer and Product Owner offered their services to provide code annotations to the codebase. For the TypeScript project, the developer and Product Owner was the same person, as this person is an active developer on the project.

In both codebases one or two clusters were chosen as the parts of the codebase that the case study would focus on. Considering the complete codebases would have required too much time from the developers, the Product Owners, and us. A cluster contains the integral code related to, e.g., a specific component/data type or functions of a certain kind. By choosing clusters in this way we tried to make sure that the case studies covered several levels of method call depth, such that TIES's approach of letting a method inherit its caller's function-links could be evaluated. By choosing clusters that contained a significant number and varying types of methods and functions, we attempted to ensure that the performance of UPSS and TIES was fairly evaluated and that the internal threat to validity was kept as low as possible. With varying types of functions we mean, for example, varying levels of complexity and dependence on varying numbers of methods. An example is an endpoint that only retrieves an entity by making one call to a repository and returning it, versus an endpoint that uses several services to validate and process a request.

8.3 Developer & Product Owner experiences

To evaluate the ease of use of our code annotations, interviews were held with the aforementioned developers and Product Owners that created the annotations and filled in the user-assigned values. These interviews were held in a semi-structured manner, and the questions asked and summaries of the interviews can be found in Appendix Section D.3. Since the developer who wrote the code annotations and the Product Owner for the TypeScript project is the same person, these 2 interviews were combined into one.

Both developers had no issues with identifying the entry points of their applications and found it easy to do. Both applications serve as an Application Programming Interface (API) and therefore have clear endpoints that serve as the application's entry points. These endpoints and their related functions were well known to the developers, and it was therefore easy to formalize the links between them.
Both the developer of the Java project and the developer of the TypeScript project found it easy to write the annotations in javadoc and JSDoc, respectively. The developer of the TypeScript project defined functions in a class centrally (using `functionDef`), and then referred to them from the entry points, as he found this to be a well-organized approach. The developer of the Java project found the approach with using javadoc minimalistic and pleasant.

After being asked if they would use the same approach for a complete codebase, both developers responded that they could. One of the developers also said that he sees usefulness in doing so, as the approach could provide good insight into where one should focus with implementing tests or making sure that code is correct. Both developers did, however, think that it should be worth their time, as they think that the amount of effort required to do so would be quite large. One of the developers suggested applying the approach to small portions of a codebase, where the value achieved would be the highest.

One developer also mentioned that he thinks the most laborious task is defining the functions and where they start and begin. He said that if he were to apply the approach again, he would firstly define all functions (and assign user value and stability values to them) before going into the code. We assume that he would do this such that he does not miss a function. In addition, he would better define what a function does and does not entail.

The Product Owner of the Java project found it easy to assign user value and stability to the functions, since he thought the function descriptions were clear and he knew what was

meant and how much this function is used.
The Product Owner of the TypeScript project found it a bit of a challenge to decide on the user value and stability. He thought it was a bit vague what the values exactly mean, and what scale should be used. He defined a system of classes for himself, where user value 5 would mean that the application could absolutely not do without this function working correctly, and user value 1 would mean that it does not really matter to anyone if the function would not work correctly.
Lastly, the Product Owner of the Java project stated that the values for functions should only be changed when the functions or their values actually change, not to shape the result to one's liking.

8.4 Evaluation interviews

In this section we first describe the setup of the interviews held with developers of the case study projects, and then describe the outcomes.

8.4.1 Interview setup

For the interviews with developers, a script with questions and tasks was developed. This script with some explanations is given in Figure 8.1. The interviews were held in a semi-structured manner; this gave the room for discussion that was not directly related to the questions asked. This room for discussion was used plentifully. If possible the interviews were held in-person, as this was easier for performing the tasks and simplified discussions where participants wanted to point out specific examples. In the end, 8 interviews were held, distributed equally over both case study projects. One of the interviews was held online due to scheduling constraints. However, no tasks were performed during this interview due to inexperience of the participant with the specific part of the codebase that the case study focused on. The summary of this interview is given in Appendix Section E.1.3.1. All evaluation interview summaries and the data collected with the tasks performed can be found in Appendix Section E.

The script starts of with welcoming the participant and inquiring on their experience levels with the code used for the case study and test automation. Then, the objective of the test sorting (sorting suggested tests on a balance between usefulness and estimated implementation effort) is explained to the participant. We explicitly do **not** tell the participant about possible factors that could be used for test prioritization, or the factors that UPSS uses. This prevents possible bias with the participant. After this, the participant is handed suggested tests without metadata. The metadata of a suggested test comprises the estimated complexity of the method(s) it covers, and the user-assigned user value and stability values. With the suggested tests available, we checked whether the participant felt that these are the appropriate tests to be suggested for their application. This is done to evaluate UPSS on Aspect 1.

Then, the participant is asked to rank the suggested tests based on which test they would implement first with the objective in mind, still without having the metadata available. This is done to examine what the participant would base their test rankings on without knowing what factors UPSS uses. This way we get answers that are not influenced by UPSS when we ask how they ranked the tests. Afterwards, the same task was given to the participant with a different set of suggested tests, but now with the test metadata available. By giving the participant the methods' complexity values and the user value and stability

1. *Welcome and thank participant. Explain that interview is semi-structured, and that the participant can make remarks or ask questions at any time.*
2. **Question:** on a scale of 1-10, how much experience do you have with the codebase of `<application name>`?
3. **Question:** on a scale of 1-10, how much experience do you have with the code of `<specific part of the application that the case study focusses on>`?
4. **Question:** on a scale of 1-10, how much experience do you have with test automation?

   With the first questions, we inquire into the related experience that the participant has. This information could become useful for making observations on the results obtained in the interview with this participant.
5. *Explain participant the objective of the strategy/tool: to suggest tests that can be implemented, sorted on a balance between usefulness of the test and the amount of effort required to implement it*

   We explain the objective of the test sorting to the participant. This information is required for the participant to perform their tasks. If the participant would not have this information, they might order the suggested tests with a different objective in mind.
6. *Hand participant randomly sorted list of suggested tests* ***without*** *metadata*[a].
7. **Question:** disregarding the order of the tests, are these the kind of tests that you would implement for `<specific part of the application that the case study focusses on>`?
8. **Task:** arrange suggested tests (without metadata), such that the tests that you think should be implemented first are on top (with the stated objective in mind)
9. **Question:** shortly, what made you arrange the tests in this specific manner?
10. *Explain metadata and role of Product Owner, and hand participant randomly sorted list of suggested tests* ***with*** *metadata.*

    We give the participant the metadata with the values as assigned by the Product Owner. This way we do not introduce different opinions on user values and stabilities as a variable.
11. **Task:** arrange suggested tests (with metadata), such that the tests that you think should be implemented first are on top (with the stated objective in mind)
12. **Question:** shortly, what made you arrange the tests in this specific manner?
13. *Hand participant lists of suggested tests with different sortings: as sorted by UPSS, sorted with $W_e = 0$, sorted with $W_c = 0$ & $W_s = 0$, sorted with $W_s = 0$, sorted with $W_{uv} = 0$ & $W_s = 0$, and randomly ordered.*
14. **Task:** using pairwise ranking, rank these sortings based on how you feel that they fulfill the aforementioned objective
15. **Question:** shortly, why did you rank the sortings in this way?
16. *Hand participant list sorted by UPSS, and explain.*
17. **Question:** with the objective in mind, would you implement the tests in this order? Why or why not?
18. *Show participant TGUI and give a short introduction. Let participant play around if desired.*
19. **Question:** would you use this way-of-working? Why or why not?

[a]The metadata of a suggested test comprises: the estimated complexity of the method(s) it covers (as this is the source of the fault-proneness metric), and the user-assigned user value and stability values.

FIGURE 8.1: Script for the user study interviews

values as they were assigned by the Product Owner, we intended to eliminate differences in opinion on the functions' user value from being a factor in the evaluation. After ranking the tests with the metadata available, we again asked the participant what made them rank the tests in the way that they did.

In the Data sections in Appendix E and Table 8.2 later in this chapter, we compare the rankings given by the participant with the ranking as produced by UPSS(1,1,1,0.25). These weighing factors are used as a benchmark since they value the test usefulness factors equally, and give a weight to the unit tests' estimated relative implementation effort that we felt was reasonable. A short interview with the Product Owner of the TypeScript project was held, in which the results of instances of UPSS with different combinations of weighing factors were compared, to confirm that UPSS(1,1,1,0.25) gave suitable results.

The comparison of participant rankings with UPSS rankings was done quantitatively using Spearman's rank correlation coefficient[45]. This is a measure that gives an indication on how well two rankings correlate, and it gives two calculated values: r_s and a two-tailed p-value. r_s can range from -1 to 1. If the two rankings contain the exact same values (as is the case in our comparisons) r_s is -1 when the rankings are exact opposites (when the values in one rank increase, they decrease in the other), and 1 when the rankings are exact copies (when the values in one rank increase, they also increase in the other). The p-value indicates the statistical significance of the result. Generally, a p-value < 0.05 is deemed to indicate a statistically significant correlation.

If the conversation and the data gave cause to do so, we also compared the test rankings from the participant with different instances of UPSS. For example, if a participant answered that they based their rankings mostly on complexity, we compared their rankings with an instance of UPSS with a higher complexity weighing factor (e.g. UPSS(1,2,1,0.25)). This allowed us to evaluate whether changing the weighing factors of UPSS had the desired effect, and also gave us some insight into the ability of participants to rate the different factors (such as complexity) of their code, especially with the rankings done without metadata available.

The next task for the participant was ranking sortings made by instances of UPSS with different weighing factors (and one random ordering serving as control group). This ranking was performed through pairwise ranking. In this process the participant was given two sortings, and asked which of the two sortings they felt better performed at sorting the tests according to the objective. This was repeated until all pairs of sortings were compared for both unit and functional tests. The sorting of UPSS(1,1,1,0) was only used for unit tests, as for functional tests it gives the exact same results as UPSS(1,1,1,0.25), as implementation effort is not considered for the prioritization of functional tests.

The purpose of this task was twofold. Firstly, getting these rankings from the participants gave us the opportunity to match their qualitative input with data and getting an insight into what ranking (and thereby weighing factors) they prefer. Secondly, the ranking process served as a conversation starter. It gave the participants insight into what UPSS and TIES would mean for *their* applications, and led them to weigh the different factors that could be used for test prioritization. Asking the participants afterwards why they would or would not use the rankings given by UPSS(1,1,1,0.25) led to insightful explanations. We think that having spent some time and effort on ranking tests and comparing tests rankings themselves gave the participants a better and more-educated feeling on what they preferred and did not prefer.

Finally, the participant was asked whether they would use the way-of-working with UPSS and TIES, and why or why not.

8.4.2 Outcomes

Eight interviews were held, distributed equally over the Java project and the TypeScript project. Table 8.1 lists all interviewees and their answers to the experience-related questions, as well as refering to the Appendix Section that contains the summary of their interview and the raw data collected during this interview. With one participant from the Java project, the task of ranking different UPSS functional test sortings was skipped due to time constraints. With another participant from the Java project, only a qualitative interview was held and no tasks were performed, due to this participant's unfamiliarity with the part of the codebase that the case study focused on.

Interviewee	Appendix Section	Experience with the codebase	Experience w.r.t. specific part of the codebase that the user study focuses on	Experience w.r.t. test automation
The Java Project				
1-1	E.1.1	9	9	7
1-2	E.1.2	7	9	5
1-3	E.1.3	7.5	1	4
1-4	E.1.4	10	6	7.5
The TypeScript Project				
2-1	E.2.1	7	6	7
2-2	E.2.2	7	7	5
2-3	E.2.3	8	7	9
2-4	E.2.4	9	7	5

TABLE 8.1: Overview of the interviewees and their stated experience levels

Test suggestion

All participants, without exceptions, understood the list of suggested tests and said that they could implement the tests as they were suggested. Multiple developers from the TypeScript project shared that unit test and functional tests are the types of test that they would implement for their project. One participant from the TypeScript project shared that if he had unlimited time available, he would implement all suggested tests. Almost all participants shared that they saw a large difference in usefulness in the tests, saying that they would not implement some of them because of a lack of usefulness (for example because they only retrieved data from a source and directly returned this data). Two participants commented on the naming of the unit tests, saying that they would name the unit tests differently. One of these participants said that he would give the unit tests a more human-readable name.

One participant from the TypeScript project commented that he thinks it is necessary to clearly define functions. Especially with an API, it needs to be made explicit whether the functions are defined from the perspective of a human (frontend) end-user, or from the perspective of the application(s) interacting with the API.

A question from a participant was whether private and public methods were considered, as he said unit testing private methods was generally not done. Currently, UPSS and TIES make no distinction on the method visiblity. During other interviews the topic of method visibilty did not come up, and the quantitative results also do not give us any indication to make observations on the topic.

Ranking suggested tests

Table 8.2 lists all results from the tasks of ranking suggested tests per interviewee, combined with their answers to the experience-related questions. It shows the Spearman's rank correlation coefficient r_s and the accompanying p-value in parentheses. If a correlation is statistically significant, it is printed in bold. Every ranking of each interviewee is firstly compared with UPSS(1,1,1,0.25). If possible a set of weighing factors was found such that the results of UPSS and the ranking of the interviewee correlated (even) stronger. Note: this was **only** done if the conversation with the interviewee also gave cause to think that they weighed the different prioritization factors non-equally. So, for example, if an interviewee mentioned basing their ranking mostly on user value and slightly on complexity, and combination of weighing factors was found that support this statement and also yield strong correlation with the interviewee's rankings, the r_s and accompanying p-value are also shown in the table. This is done to show the ability of UPSS to cope with different prioritization strategies by adapting its weighing factors. If on the other hand no suitable weighing factors could be found to support an interviewee's statements, no extra results are shown in the table, even if there would be a combination of weighing factors that gave stronger correlation between the results of UPSS and the interviewee's ranking.

We observe that especially for the Java project, it was hard to find suitable combinations of weighing factors. We think that this is mostly due to participants not agreeing with the user values assigned by the Product Owner and an issue relating to how functions are used in UPSS combined with how a part of the Java project is implemented. Namely, there is a set of entry points in the Java project that all perform very similar tasks and share a lot of underlying code. However, all these functions were assigned different functions (as the Product Owner was advised to do so by the manual we gave him), and therefore the underlying methods of these entry points are linked to many functions. As these functions were also assigned high user values, tests for these methods and functions got high priority values. Using combination of weighing factors with the weighing factor for user value set low or even to 0 did not remedy this issue, as the participants in the user study did incorporate user value into their test sortings, however not in the way that UPSS did.

Ranking suggested tests without metadata All participants were able to rank the suggested tests without metadata. Factors that the participants noted were influential to their rankings and how often they were mentioned are listed in Table 8.3.

The explanations given after ranking the tests without metadata already gave interesting insights. Especially two opinions were given that contradict the ideas behind UPSS. Namely, several interviewees stated that code that had been in the codebase for a long time and would stay there, they were less likely to test, as it was proven empirically that it functions well. This contradicts the original idea behind UPSS that code that will be stable for a long time has a higher priority for testing, as less effort has to be spent on maintaining the test. Also, interviewee 2-3 from the TypeScript project said that paradoxically, some methods/functions were so important that if they were to fail it would be immediately noticed when the application was released to the project's acceptance environment, where

Interviewee	a	b	c	d
The Java Project				
1-1	0.04 (0.51)	0.12 (0.80)	**0.63 (0.01)**	0.35 (0.20)
1-2	0.11 (0.84)	0.43 (0.12)	0.25 (0.36)	0.01 (0.97)
1-3	-	-	-	-
1-4	0.44 (0.10)	0.50 (0.06)	**0.54 (0.04)**	**0.66 (0.01)**
	0.66 (0.01)*	**0.58 (0.03)***		
The TypeScript Project				
2-1	0.37 (0.18)	**0.57 (0.04)**	**0.54 (0.04)**	**0.81 (0.00)**
2-2	0.39 (0.16)	−0.44 (0.13)	**0.61 (0.01)**	0.23 (0.44)
	0.19 (0.49)§	**−0.55 (0.04)**†	**0.70 (0.00)**§	**0.95 (0.00)**†
2-3	0.44 (0.10)	0.50 (0.06)	0.40 (0.15)	0.14 (0.62)
	0.60 (0.02)‡	**0.60 (0.02)**‡	0.27 (0.33)‡	**0.87 (0.00)**‡
2-4	0.45 (0.09)	0.15 (0.68)	−0.32 (0.25)	0.14 (0.74)
			0.90 (0.00)¶	**0.86 (0.00)**¶

a: r_s of unit test ranking without metadata compared to UPSS(1,1,1,0.25) (p-value)
b: r_s of functional test ranking without metadata compared to UPSS(1,1,1,0.25) (p-value)
c: r_s of unit test ranking with metadata compared to UPSS(1,1,1,0.25) (p-value)
d: r_s of functional test ranking with metadata compared to UPSS(1,1,1,0.25) (p-value)

*: Compared with UPSS(0,1,0,0.25)
§: Compared with UPSS(1,0,-1,0.25)
†: Compared with UPSS(1,1,-1,0.25)
‡: Compared with UPSS(1,0.5,0,0.25)
¶: Compared with UPSS(2,0,-1,0.5)

TABLE 8.2: Experience and test sorting data from evaluation interviews (with statistically significant results in bold)

people test the application before it is released to production. This example indicates that there are differences in prioritization between projects because of environmental factors, such as how the release pipeline is constructed.

A different view was shared by interviewee 2-4; he took the approach of first grouping tests that he deemed similar, and afterwards looking at if he would implement a test, whether that would also cover other test items. This is a factor that UPSS is not fully able to incorporate. Another prioritization factor that UPSS currently does not incorporate was mentioned by interviewee 2-2. He mentioned that he prioritized tests for functions/code that he knows currently contain bugs the highest.

If a participant had different ideas on how tests should be prioritized, their test ranking would obviously have little to no positive correlation with the ranking of UPSS(1,1,1,0.25). Interviewee 2-4, who noted that he ranked the tests based mostly on the impact to customers, gave a unit test ranking with a significant strong correlation with UPSS(2,0,-1,0.5). This shows that he indeed valued user value relatively high, and that he seems to agree with his Product Owner on user value values (this participant also indicated high experience levels with the codebase and test automation).
Interviewee 1-4, who indicated that he is highly experienced with his respective codebase (giving himself a 10 out of 10 on his experience level), stated that he ranked the tests mostly

Factor	**Amount of times mentioned by developer from the Java project**	**Amount of times mentioned by developer from the TypeScript project**
How often a method/function is used	1	1
Whether the participant's experience tells them that a method or function contains/-contained a lot of faults	0	2
Effort required to implement the test	1	3
Complexity of the code	1	1
Whether the method/function is related to security	0	2
Stability	0	1
User value	1	1
Whether the test would cover multiple methods/functions	0	1
Whether a fault in the method/function would immediately be noticed in the acceptance environment	0	1

Table 8.3: Factors that interviewees mentioned were influential to their test sortings

based on his knowledge of how much logic/complexity the code contains. This statement matches well with his actual suggested test rankings, as his rankings showed a significant high correlation with rankings made by UPSS(1,2,1,0.25).

An observation that was made was that multiple participants ranked unit tests for high-level controller methods highly, even though these methods contain barely any logic and are mostly a conduit to the service method below, and thus have a low complexity value and are therefore given a low priority by UPSS implementations that weighed complexity positively.

Interviewee 2-2 shared that seeing the methods and functions with their complexity values could be a good starting point for refactor prioritization.

Ranking suggested tests with metadata It is clearly visible from the data that with the metadata available, the test rankings of the participants correlated a lot stronger with the ranking of UPSS than when the participants did not have the metadata available to them. For 4 out of the 7 unit test rankings, and 2 out of the 7 functional test rankings, there were significant strong correlations with the ranking of UPSS(1,1,1,0.25). This indicates that these participants have a similar way of prioritizing tests as UPSS.

For interviewee 2-2, who described seeing a high stability value as an indication that the code does not need testing, and who also indicated that he valued security and impact to customers highly, there was an extremely high significant correlation between his rankings and the ranking of UPSS(1,1,-1,0.25).

Interviewee 2-3 noted that he could argue stability both ways; either as having a positive relation with test priority since the use of spending effort for tests that would need to be refactored soon can be deemed low, but also as having a negative relation with test priority, as when one is going to refactor code, it can be good to have regression tests. He says that the environment (composition of team, quality of team members) could also influence how stability should be used. Interviewee 1-4 shared similar views; he said that when the stability is high, one could say that the code would exist for a long time, and that therefore there is extra usefulness in testing it. However, he also realizes that one could say that as the code would not change for a long time, automated testing is not that useful after the function has been tested manually once. For functions with a low stability value, he said that he would not be inclined to write tests for them, since this function would probably only be a trial and would not be around for long. If a stability value would be low because the code was due for a refactor, he would not be inclined to write a unit test for it, since it would have to be rewritten.

Interviewee 1-4 (with the stated 10 out of 10 experience level) stated that he would have given different user value values to some functions than his Product Owner had, based on his experience with the functions' usage. He also notes a function that he thinks should have been described more precisely, as currently it seems that the function is only linked with one entry point, while its description would suggest multiple entry points being involved. A developer from the other project team also shared that he felt that he would have given different user values to some functions than those that were given by his Product Owner.

Ranking different test sortings

In Tables 8.4 the results of the rankings of the test sortings are aggregated. Per combination of sorting and rank it is listed how many interviewees put the sorting on this rank. In the case of ties (two or more sortings getting the same number of upvotes and downvotes in the pairwise ranking) all tied sortings are placed in the column of the lowest rank they occupy. So if there is a two-way tie for first place, both sortings are placed in the #1 column. This is because there is no better way of displaying such a tie in a table. However, for the average rank calculation the average rank of the tied sortings is taken. So for the same two-way tie for first place, for both sortings a "rank" of 1.5 ($(1 + 2)/2$) is used for the average rank calculation, as this is more fair to the lower placed sortings.

It should first of be noted that we think that the sample size, especially for the results from the Java project(and therein especially the results for functional tests), is too low. We do therefore not want to draw hard conclusions from this data, but merely want to identify trends and state possible causes. With a low sample size, the influence of external factors, such as the perceived quality of the random ordering, becomes larger.

We can see that for unit tests, UPSS(1,1,1,0) scores the highest, followed by UPSS(1,1,1,0.25). This seems to indicate that either the participants did not regard estimated effort beneficiary for their comparisons, they deemed UPSS's effort estimation incorrect, or that the value of $W_e = 0.25$ was chosen too low, causing the effect of the effort estimate to be negligible. For unit tests it can be seen that the approaches that combine multiple factors (UPSS(1,1,1,0.25), UPSS(1,1,1,0), and UPSS(1,1,0,0.25)) score quite a bit better than approaches that only consider one factor (UPSS(1,0,0,0.25) and UPSS(0,1,0,0.25)). This seems to indicate the developers find it useful to consider multiple factors, instead of only one.

The same observation as made about estimated effort can also be made for stability. UPSS(1,1,0,0.25)was not ranked much worse than UPSS(1,1,1,0.25)and UPSS(1,1,1,0). This indicates that either the value of including stability with weighing factor $W_s = 1$ is low, or that the opinions on whether stability should have a positive or negative relation to test priority almost cancel each other out. The possibility that a negative weighing factor

Unit test sortings, the Java project							
Sorting	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**Average rank**
UPSS(1,1,1,0.25)	1	1	1	0	0	0	2.17
UPSS(1,1,1,0)	1	1	1	0	0	0	2.17
UPSS(1,0,0,0.25)	0	0	0	2	0	1	4.67
UPSS(1,1,0,0.25)	0	0	2	0	1	0	3.83
UPSS(0,1,0,0.25)	2	0	0	0	1	0	2.50
Random	0	0	0	0	1	2	5.67

Unit test sortings, the TypeScript project							
Sorting	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**Average rank**
UPSS(1,1,1,0.25)	2	0	0	1	1	0	3.13
UPSS(1,1,1,0)	3	0	0	1	0	0	2.13
UPSS(1,0,0,0.25)	0	1	1	0	2	0	3.75
UPSS(1,1,0,0.25)	2	0	1	1	0	0	2.50
UPSS(0,1,0,0.25)	0	1	0	1	1	1	4.38
Random	0	0	1	0	1	2	5.13

Unit test sortings, total							
Sorting	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**Average rank**
UPSS(1,1,1,0.25)	3	1	1	1	1	0	2.71
UPSS(1,1,1,0)	4	1	1	1	0	0	2.14
UPSS(1,0,0,0.25)	0	1	1	2	2	1	4.14
UPSS(1,1,0,0.25)	2	1	2	1	1	0	3.07
UPSS(0,1,0,0.25)	2	1	0	1	2	1	3.57
Random	0	0	1	0	2	4	5.36

Functional test sortings, the Java project						
Sorting	**#1**	**#2**	**#3**	**#4**	**#5**	**Average rank**
UPSS(1,1,1,0.25)	0	0	0	2	0	4.25
UPSS(1,0,0,0.25)	1	0	0	0	1	3.00
UPSS(1,1,0,0.25)	0	1	1	0	0	2.75
UPSS(0,1,0,0.25)	1	1	0	0	0	1.75
Random	0	1	0	1	0	3.25

Functional test sortings, the TypeScript project						
Sorting	**#1**	**#2**	**#3**	**#4**	**#5**	**Average rank**
UPSS(1,1,1,0.25)	1	0	3	0	0	2.63
UPSS(1,0,0,0.25)	2	1	1	0	0	1.88
UPSS(1,1,0,0.25)	0	2	1	1	0	2.88
UPSS(0,1,0,0.25)	1	0	0	2	1	3.63
Random	1	0	0	0	3	4.00

Functional test sortings, total						
Sorting	**#1**	**#2**	**#3**	**#4**	**#5**	**Average rank**
UPSS(1,1,1,0.25)	1	0	3	2	0	3.17
UPSS(1,0,0,0.25)	3	1	1	0	1	2.25
UPSS(1,1,0,0.25)	0	3	2	1	0	2.83
UPSS(0,1,0,0.25)	2	1	0	2	1	3.00
Random	1	1	0	1	3	3.75

Tables 8.4: Results for the ranking of sortings by interviewees. The #1,...#6 columns state the number of times that a sorting was ranked in this place. E.g., a 3 in the column with header #1 means that this sorting was ranked first three times

could be best for stability was not considered beforehand, and therefore the evaluation methodology did not account for it.

For functional tests the observation that the interviewees seemed to prefer sortings that combine factors does not hold, as UPSS(1,0,0,0.25) ranks highest and UPSS(1,1,1,0.25) ranks lowest. From the data, it seems that for functional tests interviewees find it important to consider the user value of functions. This is confirmed by the conversations that were had during the interviews, as multiple interviewees mentioned that they deemed user value as the most important factor for sorting functional tests. Looking back at the unit tests, the sorting based mostly on user value (UPSS(1,0,0,0.25)) ranked the worst of all non-random sortings, and this indicates that the weighing of the different factors by the participants change depending on the type of tests concerned.

For the interviewee that was the most vocal in not agreeing with a positive weighing factor for stability (interviewee 2-2), we see that his view is reflected in his rankings of the test sortings, as sortings with $W_s = 0$ score well.

During and immediately after the ranking of test sortings also other views were shared by the interviewees. Interviewee 2-4 noted on a functional test with a high complexity value that even though this test's complexity value was high, he thinks that writing the test would not cost more effort compared to writing other functional tests. Interviewee 2-2 also stated that the complexity of a function would not directly influence the amount of effort required to implement a test for this function. Interviewee 2-4 also noted unsolicitedly that he thinks that writing unit tests for more complex code would cost more effort. 2-2 openly debated whether function user values should be summed together for a unit test or not. In the end he thinks they should, as the functions are seperate items that should be considered seperately.

Interviewee 1-4 said that he felt that the current complexity estimation was too simple, and that possibly a combination of cyclomatic complexity and counting LOC would be better. Interviewee 2-3 shared that for at least one method he thought that the complexity was rated too highly. He says that this method mostly calls libraries and should be easy to understand for a developer.

Thoughts about way-of-working

At the end of the interview, the interviewees were asked to "play around" in TGUI and to share their thoughts on the way-of-working with UPSS and TIES.

The previously mentioned interviewee that did not agree with the positive weighing factor for stability, interviewee 2-2, said that even though he did not agree with the weighing factors used for UPSS in the evaluation, he thought that there potentially could be a combination of weighing factors that would fit his opinions almost perfectly. When using TGUI, he was quickly able to find such a combination of weighing factors.

All participants shared that they were positive about UPSS and TIES, but not all would immediately start using it. Special positive attention was given to being able to adjust weighing factors and to the insights that the approach provided. Interviewee 1-1 valued the insights that the approach with combining user value and complexity gives, since, he says, complexity is normally "underwater" and less tangible. Four interviewees shared explicitly that weighing factors should be tuned per project. Interviewee 2-2 said that the weighing factors could evolve as a project and/or the organization it is developed in evolve. For example, he says, when focus shifts from innovation to keeping the application stable, the weighing factors could adapt.

Some fears were shared that applying UPSS with TIES to a complete codebase would cost a significant amount of effort. Several participants suggested the use of LLMs to lighten the workload. Also quite some other suggestions were made. Interviewees 1-3 and 2-3 suggested integrating with code coverage tooling. Interviewee 1-4 shared that he would prefer using a scale of 1 to 10 (instead of 1 to 5) for the functions' user value and stability.

Two interviewees elaborated on the possiblity of using UPSS also for functions that are yet to be developed. Interviewee 1-2 suggests setting up scaffolds in the codebase for new code that could then already be given user-assigned values for user value, stability, and complexity. Interviewee 1-4 suggested integrating UPSS into the Scrum process. The "business side" of the team (e.g., Product Owner) could assign an importance value to backlog items that could be used as user value, and the developer team assigned story points could be used as an indication for the function's complexity.

Chapter 9

Results discussion

In this section the results of the evaluation interviews are discussed in more detail. In the process, the research questions from Section 1.1 are answered. Items that we think could be researched in the future are discussed in Section 10.4, while possible improvements to UPSS and TIES are discussed in Section 10.5. These items are therefore not all discussed in this chapter.

Test suggestion

Research question RQ1.1 asked how to suggest possible tests in an automated manner. This topic has two parts to it, namely the quality of the test suggestions (Aspect 1), and the ease with which they were achieved (Aspect 6).

We have shown that UPSS is able to suggest tests to a satisfactory degree for the two case study projects. All participants in the case studies understood the tests as they were suggested, and multiple participants explicitly noted that the types of tests suggested would be the types of tests they would implement.

We were not able to design an approach for test suggestion that is fully automated, as we found no way of suggesting satisfactory functional tests without some user input. In the case studies, two developers were tasked with writing code annotations to define functions in their codebases, such that TIES could use these definitions to suggest tests. Both developers found it easy to do so and found it easy to find the entry points in their applications, but also stated that they would only perform the task for a complete codebase if they believed that it was worth their time. One of the developers stated that he thought that the task of defining functions and what they entail required quite some effort. However, both developers did state that they liked the approach of TIES using code annotations. We think that, without the use of AI/LLMs, there is currently no approach available that would give the same results with less effort required, as the task of defining functions cannot be taken away and we think that the approach of using code annotations is as low effort as an approach could get.

Answering RQ1.1, we can say that unit tests can be suggested in a fully automated manner by analyzing the codebase and suggesting unit tests for all methods found. Functional tests however cannot.

Test usefulness calculations

Research question RQ1.2 and Aspect 2 cover the perceived value of UPSS's test usefulness calculations. It seems that with fault-proneness and user value the factors that are deemed the most important are covered, as can be seen by the high correlations achieved between sortings of UPSS and those created by users.

Mostly the approach of using the LOC of a method for its fault-proneness estimation was found suitable by user study participants. One participant thought that the fault-proneness metric should be extended; we talk about this in Section 10.5.5.

Not all participants seemed to regard stability as a useful factor for test prioritization. There were conflicting opinions on whether the relation between stability and test priority should be positive or negative, and it seems that compacting stability into one numerical value seems to be insufficient. A low stability value could be due to an upcoming refactor (in which case some teams would like to write regression tests), or it could be due to the function being removed soon (in which case no team would want to write tests for this function). While UPSS's approach with weighing factors, that can also be negative or zero, is able to deal with different opinions on whether a factor should be a positive or negative for test prioritization, it seems that the value of regarding stability for test prioritization in the way that it is implemented in UPSS could be low.

Some different opinions were found on how the user value for a unit test should be calculated, when the tested method contributes to multiple functions. The approach chosen for UPSS, where a method's unit test gets a higher calculated user value when the method contributes to multiple functions seemed mostly liked by user study participants. One of the Product Owners of the case study projects said that he preferred the approach of taking the maximum of the functions' user value values, as he thinks that a "highest-level" user value would always outweigh "lower-level" user values, no matter how many of them there are.

The choice to use the Fibonacci sequence to non-linearly map user values to values that are used in the priority calculation is difficult to evaluate individually through the case study as we performed it. However, as (after selecting a suitable combination of weighing factors) high correlations were achieved between the sorting from UPSS and the sortings done by user study participants, we can say that using the Fibonacci sequence seems to not have been detrimental to the results. We can however not say whether no better method exists.

It is found that having more input for the test prioritization (such as how many library calls a method makes or whether a function is security-related) could improve the results of said test prioritization. However, this extra input comes at the cost of requiring extra effort to get into the strategy. We thinks this extra effort would not be worth it. This is also due to indications that the current approach with defining functions might already be pushing the limits on the amount of effort that a development team is willing to spend for applying UPSS.

Implementation effort estimation

To answer RQ1.3 we looked at existing research, and found that correlations have been found between the LOC of code under test and its accompanying unit test(s)[43, 50, 13, 49]. We cannot conclusively say whether estimating implementation effort for unit tests using the LOC of their tested methods is a suitable approach or not based on the results from the user studies, as participants did not clearly prefer test sortings that had the effort estimation incorporated.

Discussions with the participants showed that some think that unit test implementation effort for a method is indeed related to the size/complexity of the method, but others say that the amount of overhead required to scaffold the unit test outweighs the implementation of the actual unit test assertions. While less granular, suggesting unit tests on a per-class basis instead of per-method could alleviate the issue, and is therefore a suggestion for future work that we make.

We do not currently see an approach that is as lightweight as ours and is able to provide similar or better results. Approaches that we did find that could possibly yield better fitting estimates (such as TPA) require a significantly larger amount of information and thus effort.

Combining test-prioritization factors

Aspect 4 is possibly the most important aspect as it covers the crux of UPSS's approach, the way it combines different factors into a final priority value that can be used to sort the suggested tests. The related research question is RQ1.4.

It is shown that developers do indeed want to consider multiple factors for their test prioritization; while ranking tests purely on e.g. fault-proneness might be able to discover a higher number of bugs, developers seem to also care about what bugs are found and how the users of their applications are affected.

The approach with using weighing factors to be able to put extra weight on factors that are deemed important for the project at hand is well-liked, and seems essential for UPSS to be usable as otherwise situations such as stability being able to be argued as both a positive and a negative for test prioritization could not be handled.

It is recommended that users adjust the weighing factors to their own project and environment, since we have shown that the right balance in combining the different test prioritization factors can be different between projects. It can be influenced by many factors, such as the objective of the project (e.g. production vs. demonstration software), but also the quality of the development team. We therefore do not suggest one specific combination of weighing factors, but recommend a user to start from UPSS(1,1,1,0.25), and tweak the weighing factors to their preferences before first using the strategy to suggest tests.

It was also observed that user study participants seemed to give different weights to the test prioritization factors based on the type of test that was being prioritized. Whether it is desired or not to prioritize the different types of tests differently is left to the user. A user could choose to use different weighing factors for the different types of tests. However, we feel that weighing factors should not change between types of tests. Both types of tests have the same goal of discovering the same faults that could be in the codebase. Therefore, we think that both types of tests should be prioritized in the same way.

Assigning user value and stability values

Evaluating UPSS and UPSS on Aspect 5, the ease of use of assigning user value and stability to functions, we find mixed results. The Product Owner of the Java project found it easy to decide on and assign user value and stability values to functions, while the Product Owner of the TypeScript project found it a bit of a challenge as he thought it was vague what the values exactly mean and what scale should be used. Nevertheless, both Product Owners were able to assign values, and it seems that user study participants mostly agreed with them.

Two user study participants, one from the Java project and one from the TypeScript project, noted occasions where they would have given a function a different user value than their Product Owner had. It is hard to say whether this is a problem, as it comes down to who you think is the authority on deciding user values for functions. We would say that this is the Product Owner. It could be possible that a Product Owner misunderstood what a function entailed from its description, and had therefore assigned improper values to a function, but discussions with the participants indicate that this was not the case.

The Product Owner of the Java project did find it easy to deduce what function was meant by the function's description, even though he was not the person that had created the definition.

The results seem to indicate that some improvements to the instructions for the Product Owners is required. It should be better defined what is meant by the different values, and how they should come to numerical values.

Opinions on way-of-working

While no aspect was created for this topic, a significant amount of feedback was gathered on the way-of-working with UPSS and especially TIES. As said, recommendations for improvements are gathered in Section 10.5. Special positive attention was given to the insights that the approach provided. The approach of collecting and showing method & function user value and complexity values provided a lot of useful insight to the project teams. This insight could be used for example to prioritize refactors, but it was also stated that the act of communicating clear user value values from the Product Owner to the development team, and being able to show the Product Owner complexity metrics, could support conversations.

Chapter 10

Final remarks

10.1 Conclusions

In this research we designed and evaluated UPSS, a strategy for suggesting and prioritizing automated tests for existing codebases. The aim of this strategy was for the strategy to be able to be implemented in a automated tool, such that the amount of effort required to apply the strategy would be as low as possible. During this research we also implemented and evaluated such an automated tool, TIES.

It is shown that UPSS provides satisfactory test suggestions by suggesting unit tests for all methods in the codebase, and functional tests for all defined functions. We were not able to fully automate the test suggestion process, as the function definitions still had to be manually given. We do think that have reduced the amount of effort required to suggest both unit and functional tests to the lowest amount possible without using techniques such as LLMs.

The test prioritization approach designed incorporates fault-proneness, user value, stability, and estimated effort metrics and shows promising results. The largest assets of the approach seem to be its inclusion of both fault-proneness and user value factors for both unit and functional tests, and its ability to use weighing factors to adapt to different projects. In case studies we were mostly able to select a combination of weighing factors such that the results of UPSS correlated stronly with rankings made by case study participants. However, the stability factor used in the prioritization approach needs further refinement. Different opinions are found on how it should be factored into the priority of a test, and it seems that our approach of combining function and code stability into one numerical value is insufficient. Test implementation effort estimates were incorporated into the strategy through the lines-of-code of the code under test. We were not able to produce conclusive results on the performance and value of these effort estimates.

TIES uses a novel approach of using code annotations to define functions in a codebase, providing the ability to automatically link functions and methods. This linking of functions and methods is essential to giving UPSS the ability to use complexity metrics and user value metrics for both unit and functional tests easily. Having all this information insightful gives users insights not only for test prioritization, but also for refactor prioritization and communication between different parts of a team, among others.

Finally, we present topics for possible future research, as well as a list of recommendations for the improvement and use of both UPSS and TIES.

10.2 Research considerations

10.2.1 Why not use AI?

The issues concerning, e.g., the availability of test generation tools and the effort required to define functions in a codebase could possible be resolved by using a well-trained AI. As mentioned in Section 3.1.1, AI-techniques are on the rise and will most likely continue to lessen the workload of developers. Solutions that can suggest (implementations of) tests are shown to exist. "Why not use AI-techniques for this research?" is therefore a fair question that we got asked often. Besides the issues with automated test case generation discussed in Section 5.1.1, there are several specific reasons why the choice was made not to incorporate AI in this research.

Explainability/exhaustiveness

A problem with AI-techniques is their lack of explainability. Even though the result may appear to be satisfactory, due to the "black-box" workings of a LLM, it cannot be verified that the result is exhaustive. For example, in the use case of this research, an AI might suggest all but one of tests that a developer would want to implement and prioritize them correctly, however that missing one test could be the most important test to be written. In this research, we designed the strategy to be exhaustive. For every function and every method a test is suggested.

Dependence on external providers

At the moment, most well-functioning LLMs can only be run on machines with a large amount of specific computing power. While it is possible for entities to set up such machines for themselves, this could be too expensive and require skills that might not be present in the organization. Therefore, it seems to make the most sense to use an external provider. However, such dependence could lead to issues with costs, confidentiality, and a lack of certainty for the continued support for the platform, for example.

High energy usage

Whether running externally or on-premise, the energy usage of a LLM is relatively high compared to dedicated software[27, 52]. Besides leading to higher costs, such high energy usage solutions could be undesirable due to companies having a sustatainability progam in place.

10.3 Threats to validity

10.3.1 Internal threats to validity

Limited number of case study interviews

In total 8 case study interviews were held, spread equally over the two case study projects. In one of these interviews, a large part of the interview script was skipped due to a lack of experience of the interviewee, and in another interview, one task of the script was skipped due to time constraints. This meant that the number of interviews and thereby data gathered was relatively limited. This could lead to a bias in the results due to the participants being outliers incidentally.

Possible shortcomings in user study design

An observation was made that several user study participants in the task of ranking different test sortings created by UPSS only looked at the positions of a small set of specific tests in the sorting. It theoretically could be that the entire sorting could be exactly to their liking, except for those specific tests, and that the participant would thereby disregard the entire sorting. We have no evidence that this has occured or has influenced the results significantly, but it could have happened.

Dependence on quality of PO values

The rankings of different test sortings and correlations between rankings made by user study participants and rankings made by UPSS with some set of weighing factors is influenced by the alignment of the participant and the respective Product Owner on the user-assigned values. Two participants, one from the Java project and one from the TypeScript project noted slight differences in how they would have assigned some values and how they were assigned by the Product Owner. While strong correlations could often still be found and the issue was not raised more often/severely and we therefore think that the influence on the results was not extremely large, it is an issue inherent to the design of the user study.

Dependence on quality of code analysis

Extensive effort was spent on making sure that our implementations of CAT gave complete and correct results for the case study projects. It was manually checked whether all methods and links between methods and functions in the part of the codebase that CAT was run on were actually present in the results. For the Java project, no issues were found. However, for the TypeScript project, it was found that one method was not properly linked to functions that used it.

As discussed, the TypeScript project is an API backend for a web application. The backend uses so-called middleware; methods that are called for every incoming request by the framework that handles all requests (NestJS[1]). In one of these methods a check was done on whether the endpoint handling the request was annotated with a certain decorator, and if so, the aforementioned method (that was missing links to functions) was called. Therefore, every time an entry point that was annotated with the decorator was used, the method was called. As the method prepared some data that was later used by methods called from the entry point, we do say that the entry point was dependent on the method. However, as there was no explicit call from the entry point (or any of its underlying methods) to the method in question, our implementation of CAT was not able to infer this dependency.

We think that inferring such a dependency is non-trivial, and could only be done by tools that are extremely tailored to the actual framework(s) in use.

For this method we analyzed its influence on the test prioritizations of the functions it should have been linked to, and found that this influence was insignificant, as the complexity of the method is low. We accepted the improper prioritization of the unit test suggested for the method (as since it was missing links to functions, its user value was deemed low by UPSS), since there were still plenty of other methods and unit tests in the case study, and we deemed the influence of this one test being improperly prioritized low.

[1]https://nestjs.com/

As said, extensive effort was spent on checking whether similar occurences were present, but it cannot be completely ruled out that somewhere a mistake was made.

10.3.2 External threats to validity

Limited number and types of case studies

We have performed our case study on two projets. These projects are written in specific programming languages and have specific environmental factors such as the objectives of the projects and the quality of their teams. It is therefore possible that if the same case study was performed on a different project, different observations and different conclusions would be made.

10.4 Future work

10.4.1 More exhaustive validation

This research evaluated UPSS mostly in an exploratory manner. While the preliminary results indicate that using UPSS with an implementation such as TIES would be helpful to development teams, further validation is required to test whether it actually is. Due to time constraints, it was not feasible in this research to let a development team commit to using UPSS with TIES for a complete codebase and for a significant amount of time. We think that only then it can be fully shown whether or not the tests prioritization of UPSS gives the desired results, and whether the relative effort estimations for unit tests are suitable.

Also, only then could the possible issue of the act of writing code annotations and explicitly defining functions requiring too much effort be evaluated.

10.4.2 Exploring the suggestion of unit tests on a class-level

Some user study participants said that the amount of overhead required to scaffold the unit test outweighs the implementation of the actual unit test assertions. While less granular, suggesting unit tests on a per-class basis instead of per-method could alleviate the issue, and is therefore a suggestion for future research that we make.

Users could still **implement** the unit tests on a per-method basis, but it could be better to take the entire class that the method lives in into account for prioritization.

10.4.3 Other ways of linking code and functionality

For UPSS to work, the methods in a codebase need to be linked to functions that the codebase provides. In this research we designed and used an approach based on code annotations, as described in Chapter 6. While we have shown that this approach was generally liked by the developers that used it, we cannot say for sure that it is the *best* approach available. Future work could study other ways that code and functionality can be linked. An idea would be to have a tool automatically extract entry points from the codebase, and present them in a graphical manner to the user, who then could link these entry points with functions. The defined links could then also be stored in a seperate file, which would eliminate the (slight) cluttering of the code with annotations.

A nice to have feature would be the ability to have it be possible that functionality is shared over multiple applications (that could also be written in different programming languages). This could possibly be done with only slight adaptations to CAT, but it should be researched if sharing functionality over multiple applications is beneficial and even entirely feasible for test suggestion and prioritization.

10.4.4 Function groups

As discussed, a situation that was encountered during the case study was that there was a set of functions that were highly similar, but sometimes had slightly different user value and stability values. These similar functions all had different entry points, but used the same underlying methods. These underlying methods were therefore linked to all these functions, and thereby unit tests suggested for these methods got a high user value score (as function user value values were summed). This led to these tests getting a high score and being ranked highly, possibly "unfairly" so.

A solution to this problem could be the addition of "function groups" in some manner. Functions could then be placed under a function group, and a intermediate user value for this function group could be calculated. An example implementation could be that UV_{fib} is changed such that it does not sum the user value values of all linked functions of a method, but sums the intermediate user value values of all linked function groups. The intermediate function group user value value could for example be calculated by taking the maximum user value value of all functions in the function group.

10.4.5 Being able to consider functions that are yet to be implemented

An interesting topic for future research could be how to extend UPSS such that it able to include functions that are not yet (fully) implemented. A possible approach could be to use an approach such as TPA (as discussed in Section 3.3.1), that performs extensive preliminary analysis on a function to be able to estimate aspects of its test(s). Obviously, this would go against the goal of UPSS being as lightweight and low-effort as possible, and it would be interesting see whether an approach can be devised that requires as little effort as possible.

During the user studies two possible approaches were suggested. One suggestion was setting up scaffolds in the codebase for new code that could then already be given user-assigned values for user value, stability, and complexity. Another suggestion was integrating UPSS into the Scrum process. The "business side" of the team (e.g., Product Owner) could assign an importance value to backlog items that could be used as user value, and the developer team assigned story points could be used as an indication for the function's complexity.

10.5 Recommendations

We have several recommendations concering extensions and the usage of UPSS and TIES.

10.5.1 Integrating with AI/LLMs

A topic that was brought up often in interviews and the user study was the possible use of AI/LLMs to automate further parts of the test suggestion and prioritization process. In Section 10.2.1 we discuss why we chose not to include such technologies, but that does not mean that the technologies could not be useful.

Items in UPSS where AI/LLMs could possibly lessen the workload are, among others:

- infering functions without them being defined explicitly,
- linking entry points and functions automatically,
- giving (partial) test implementations,
- deriving more human-like names for unit tests.

10.5.2 Improvements to the naming of unit tests

Unit tests are currently simply listed as "Unit test for: <name of the method>". In practice, the name of a unit test often includes information on what the method should or should not do. For example, a method that parses strings to numbers could have unit tests called "Should parse a positive number correctly", "Should parse 0 correctly", and "Should return NaN for input without numbers".

With the information that is currently available in UPSS, achieving such feats would not be possible. With the inclusion of AI/LLMs it might be.

10.5.3 UX improvements for TGUI

Being able to lock weighing factors

Multiple participants in the user studies said that they think that the weighing factors should be "locked" after they are chosen for a software project. This would prevent users from changing weighing factors such that the results are more to their liking. A user study participant did however not (and we agree with him on this) that weighing factors could change while a project evolves. For instance when the team working on the project changes, or when the focus shifts from quickly getting to the market with the product to keeping the product online and stable.

A possible way of achieving this could be implementing a "admin interface" for the weighing factors in or besides TGUI, that could, e.g., be locked behind a password. This way only the person with the authority to do so could change the weighing factors.

Integrating code coverage

It would be interesting for TIES to integrate with code coverage tooling, such that the currently achieved code coverage for the code related to a test can be shown. This way, a user could see when code is sufficiently tested, and could therefore then skip creating the suggested test.

Displaying and being able to filter on method visibility levels

In Section 5.1.3 we discussed the issue of whether to include private methods in UPSS. We chose to do so, and we stand by this choice as we have found no issue with including private methods. A good addition to TIES would be to incoporate method visibility, such that users can make their own choices on whether to write tests for private methods and/or use the information displayed as insight for refactoring purposes or not.

To be able to do so, CAT would need to be extended to also save method visibility information, and TSST would need to be adapted to assign this information to its output where relevant. Finally, TGUI would need to be extended with the ability to display method visibility for unit tests, and with the option to filter out unit tests for private methods if desired.

Displaying more intricate details of methods

Besides visibility, there could be more details on methods that could be displayed to the user. For example, details on the branches in a method or how much of the method consists of dependency calls could be of interest.

10.5.4 Use a different scale for user value and stability assignment

For the evaluation in this research we recommended the Product Owners to assign user values and stability values to the functions on a scale of 1 to 5. In the case study on the Java project, we saw that a lot of functions were assigned user values of 5. Because of this lack in difference in user values, functions (and thereby methods linked to them) could only be differentiated on other metrics, even though there could possibly be (subtle) differences in the user values of these functions. We think that it would have been better to use a scale with a higher granularity, such as the scale of 1 to 10. As there are no limitations on the scale used for user-assigned values in UPSS and TIES, no adaption of either is strictly necessary to do so. However, the current method of using the Fibonacci sequence to non-linearly map user values to values that are used in the priority calculation would probably not be suitable if a different scale would be used. A possible approach could be using a method where some exponential factor e is used, e.g.:

$$UV_{exp}(m) = \sum\nolimits_{f \in F_m} (uv_f)^e$$

This would have a similar effect to the Fibonacci sequence, but could be tuned to suit different scales. This approach was not conceived before the evaluation of UPSS was started, and therefore it was chosen to keep using UV_{fib}.

10.5.5 Other/better fault-proneness metric(s)

While estimating fault-proneness by LOC was chosen because of its simplicity and demonstrated performance in literature (as discussed in Sections 3.2 and 5.2), it could be that different fault-proneness metrics would lead to better fault-proneness estimations. In the user studies, some examples were found where methods implemented practically the same functionality, but were written quite differently. This difference in way of implementation led to quite different complexity estimations and confusion with participants. Now it could be that there is an actual difference in fault-proneness between the different implementations, or it could be that this difference is non-existent or negligible. Especially one participant noted methods that he thought had an inflated complexity

Future applications of UPSS could try using different fault-proneness metrics. Examples of such fault-proneness metrics could be calculating the method's cyclomatic complexity, using object-oriented code metrics, considering what share of the method is library calls only, or regarding metrics based on the fault or modification history of the method. Possibly a combination of several metrics could perform best. The only part of TIES that would need to be significantly changed is CAT, as its implementations currently do not gather other metrics on code than the length of methods. A slight modification to TSST would be required, as it currently uses the methods' lengths for assigning the complexity of tests, and it would then need to use a different property that is given to it by CAT.

We expect that the influence on the results of UPSS would be minimal. As as long as there is a positive relation between the fault-proneness metrics, the exact priority value for a test might change, but its ranking compared to other tests would not. Some preliminary experimentation with the Java project's codebase show that indeed a test sorting with cyclomatic complexity as complexity estimation correlates well with a test sorting with LOC as complexity estimation.

Bibliography

[1] IEEE guide for software requirements specifications. *IEEE Std 830-1984*, pages 1–26, 1984. doi:10.1109/IEEESTD.1984.119205.

[2] Allan J Albrecht. Measuring application development productivity. In *Proc. Joint Share, Guide, and Ibm Application Development Symposium*, pages 83–92, 1979.

[3] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software*, 86(8):1978–2001, 2013.

[4] Eduardo Aranha, Filipe de Almeida, Thiago Diniz, Vitor Fontes, and Paulo Borba. Automated test execution effort estimation based on functional test specifications. In *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques, MUTATION*, volume 7, pages 67–71, 2007.

[5] Erik Arisholm and Lionel C Briand. Predicting fault-prone components in a Java legacy system. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 8–17, 2006. doi:10.1145/1159733.1159738.

[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015. doi:10.1109/TSE.2014.2372785.

[7] Kent Beck. *Test-Driven Development: By Example.* Addison-Wesley Professional, 2003.

[8] Guru Bhandari, Ratneshwer Gupta, and Satyanshu Upadhyay. An approach for fault prediction in SOA-based systems using machine learning techniques. *Data Technologies and Applications*, ahead-of-print, September 2019. doi:10.1108/DTA-03-2019-0040.

[9] Arnamoy Bhattacharyya and Timur Malgazhdarov. PredSym: Estimating software testing budget for a bug-free release. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, pages 16–22, 2016. doi:10.1145/2994291.2994294.

[10] Ilona Bluemke and Agnieszka Malanowska. Tool for assessment of testing effort. In *Engineering in Dependability of Computer Systems and Networks: Proceedings of the Fourteenth International Conference on Dependability of Computer Systems DepCoS-RELCOMEX, July 1–5, 2019, Brunów, Poland*, pages 69–79. Springer, 2020. doi:10.1007/978-3-030-19501-4_7.

[11] Ilona Bluemke and Agnieszka Malanowska. Software testing effort estimation and related problems: A systematic literature review. *ACM Computing Surveys (CSUR)*, 54(3):1–38, 2021. doi:10.1145/3442694.

[12] Barry W Boehm. Software engineering economics. *IEEE transactions on Software Engineering*, (1):4–21, 1984. doi:10.1109/TSE.1984.5010193.

[13] Magiel Bruntink and Arie van Deursen. An empirical study into class testability. *Journal of systems and software*, 79(9):1219–1232, 2006. doi:10.1016/j.jss.2006.02.036.

[14] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *15th International Symposium on Software Reliability Engineering*, pages 113–124. IEEE, 2004. doi:10.1109/ISSRE.2004.18.

[15] Daniel Guerreiro e Silva, Bruno Teixeira de Abreu, and Mario Jino. A Simple Approach for Estimation of Execution Effort of Functional Test Cases. In *2009 International Conference on Software Testing Verification and Validation*, pages 289–298. IEEE, 2009. doi:10.1109/ICST.2009.47.

[16] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 329–338. IEEE, 2001. doi:10.1109/ICSE.2001.919106.

[17] Michael Felderer and Ina Schieferdecker. A taxonomy of risk-based testing. *International Journal on Software Tools for Technology Transfer*, 16:559–568, 2014. doi:10.1007/s10009-014-0332-3.

[18] N.E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, Aug./2000. doi:10.1109/32.879815.

[19] Norman E Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on software engineering*, 25(5):675–689, 1999. doi:10.1109/32.815326.

[20] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419, Szeged Hungary, September 2011. ACM. doi:10.1145/2025113.2025179.

[21] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 178–188, Zurich, June 2012. IEEE. doi:10.1109/ICSE.2012.6227195.

[22] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Transactions on Software Engineering and Methodology*, 24(4):1–49, September 2015. doi:10.1145/2699688.

[23] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000. doi:10.1109/32.859533.

[24] James Grenning. Planning poker or how to avoid analysis paralysis while release planning. *Hawthorn Woods: Renaissance Software Consulting*, 3:22–23, 2002.

[25] Charitha Hettiarachchi and Hyunsook Do. A Systematic Requirements and Risks-Based Test Case Prioritization Using a Fuzzy Expert System. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 374–385, Sofia, Bulgaria, July 2019. IEEE. doi:10.1109/QRS.2019.00054.

[26] Tilman Holschuh, Markus Pauser, Kim Herzig, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects in SAP Java code: An experience report. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 172–181. IEEE, 2009. doi:10.1109/ICSE-COMPANION.2009.5070975.

[27] Max Hort, Anastasiia Grishina, and Leon Moonen. An Exploratory Literature Study on Sharing and Energy Use of Language Models for Source Code, July 2023. arXiv:2307.02443.

[28] Gustav Karner. Resource estimation for objectory projects. *Objective Systems SF AB*, 17(1):9, 1993.

[29] R. Kavitha, V.R. Kavitha, and N. Suresh Kumar. Requirement based test case prioritization. In *2010 INTERNATIONAL CONFERENCE ON COMMUNICATION CONTROL AND COMPUTING TECHNOLOGIES*, pages 826–829, October 2010. doi:10.1109/ICCCCT.2010.5670728.

[30] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[31] Agnieszka Malanowska and Ilona Bluemke. ISO 25010 support in test point analysis for testing effort estimation. *Integrating Research and Practice in Software Engineering*, pages 209–222, 2020. doi:10.1007/978-3-030-26574-8_15.

[32] Roberto Meli. Simple function point: A new functional size measurement method fully compliant with IFPUG 4.x. In *Software Measurement European Forum*, 2011.

[33] Glenford J Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[34] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, pages 452–461, 2006. doi:10.1145/1134285.1134349.

[35] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change Bursts as Defect Predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/ISSRE.2010.25.

[36] Namita Panda and Durga Prasad Mohapatra. Test scenario prioritization from user requirements for web-based software. *International Journal of System Assurance Engineering and Management*, 12(3):361–376, June 2021. doi:10.1007/s13198-021-01056-4.

[37] Santosh S. Rathore and Sandeep Kumar. A study on software fault prediction techniques. *Artificial Intelligence Review*, 51(2):255–327, February 2019. doi:10.1007/s10462-017-9563-5.

[38] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 179–188, Oxford, UK, 1999. IEEE. doi:10.1109/ICSM.1999.792604.

[39] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006. doi:10.1109/MS.2006.91.

[40] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw, and Gordon Fraser. How do automatically generated unit tests influence software maintenance? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261. IEEE, 2018. doi:10.1109/ICST.2018.00033.

[41] Ashish Sharma and Dharmender Singh Kushwaha. A metric suite for early estimation of software testing effort using requirement engineering document and its validation. In *2011 2nd International Conference on Computer and Communication Technology (ICCCT-2011)*, pages 373–378. IEEE, 2011. doi:10.1109/ICCCT.2011.6075150.

[42] Martin Shepperd and Darrel C Ince. A critique of three metrics. *Journal of systems and software*, 26(3):197–210, 1994. doi:10.1016/0164-1212(94)90011-6.

[43] Emad Shihab, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, and Robert Bowerman. Prioritizing the creation of unit tests in legacy software systems. *Software: Practice and Experience*, 41(10):1027–1048, 2011. doi:10.1002/spe.1053.

[44] Parmanand Singh. The so-called Fibonacci numbers in ancient and medieval India. *Historia Mathematica*, 12(3):229–244, 1985. doi:10.1016/0315-0860(85)90021-7.

[45] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. arXiv:1412159, doi:10.2307/1412159.

[46] Hema Srikanth, Sean Banerjee, Laurie Williams, and Jason Osborne. Towards the prioritization of system test cases. *Software Testing, Verification and Reliability*, 24(4):320–337, 2014. doi:10.1002/stvr.1500.

[47] Hema Srikanth, Charitha Hettiarachchi, and Hyunsook Do. Requirements based test prioritization using risk factors: An industrial study. *Information and Software Technology*, 69:71–83, 2016. doi:10.1016/j.infsof.2015.09.002.

[48] Marcel Steinbeck and Rainer Koschke. Javadoc Violations and Their Evolution in Open-Source Software. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 249–259, March 2021. doi:10.1109/SANER50967.2021.00031.

[49] Valerio Terragni, Pasquale Salza, and Mauro Pezzè. Measuring Software Testability Modulo Test Quality. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 241–251, Seoul Republic of Korea, July 2020. ACM. doi:10.1145/3387904.3389273.

[50] Fadel Toure, Mourad Badri, and Luc Lamontagne. A metrics suite for JUnit test code: A multiple case study on open source software. *Journal of Software Engineering Research and Development*, 2:1–32, 2014. doi:10.1186/s40411-014-0014-6.

[51] Erik Van Veenendaal and Ton Dekkers. Test point analysis: A method for test estimation. In *Project Control for Software Quality: Proceedings of the Combined 10th European Software Control and Metrics Conference and the 2nd SCOPE Conference on Software Product Evaluation, April 27-29, 1999, Herstmonceux, England*, pages 47–59. Shaker-Verlag, 1999.

[52] Roberto Verdecchia, Luís Cruz, June Sallou, Michelle Lin, James Wickenden, and Estelle Hotellier. Data-Centric Green AI: An Exploratory Empirical Study. In *2022 International Conference on ICT for Sustainability (ICT4S)*, pages 35–45, June 2022. arXiv:2204.02766, doi:10.1109/ICT4S55073.2022.00015.

[53] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. TimeAware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 1–12, Portland Maine USA, July 2006. ACM. doi:10.1145/1146238.1146240.

[54] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering*, pages 1–27, 2024. doi:10.1109/TSE.2024.3368208.

[55] Nicholas C Zakas. *Maintainable JavaScript: Writing Readable Code.* " O'Reilly Media, Inc.", 2012.

[56] Xiaofang Zhang, Changhai Nie, Baowen Xu, and Bo Qu. Test Case Prioritization Based on Varying Testing Requirement Priorities and Test Case Costs. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 15–24, Portland, OR, USA, 2007. IEEE. doi:10.1109/QSIC.2007.4385476.

[57] Zhaoyang Zhang. *The Benefits and Challenges of Planning Poker in Software Development: Comparison between Theory and Practice.* PhD thesis, Auckland University of Technology, 2017.

[58] Xiaoqi Zhao, Haipeng Qu, Jianliang Xu, Xiaohui Li, Wenjie Lv, and Gai-Ge Wang. A systematic review of fuzzing. *Soft Computing*, 28(6):5493–5522, March 2024. doi:10.1007/s00500-023-09306-2.

[59] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9. IEEE, 2007. doi:10.1109/PROMISE.2007.10.

Glossary

Endpoint

An exposed point of an API that fulfils a specific function.

Entry point

The method where a request or command from a user enters the program. Further described in Section 6.3.

Function

In the context of this research, a "function" refers to a single piece of functionality. This would often be an action that a user could perfrom in a program. Not to be confused with method.

Functional test

A test that tests a (part of a) software system on conformance to specified functionality. Further described in Section 2.1.

Functionality

The tasks that a software program is able to do.

Method

Also called "function" or "subroutine"; a distinct piece of code that can be called. Not to be confused with function.

Product Owner

The team member in an Agile software development project that aligns the backlog of the developers with the wishes of the stakeholders. In doing so, they prioritize backlog items and keep track of future work for the project.

Unit

A (frequently small) piece of code that can be tested isolated from the rest of the system. Often a specific method.

Unit test

A test that tests a specific unit independently on conformance to expected behavior. Further described in Section 2.1.

Acronyms

Abstract Syntax Tree (AST)

A data structure that describes the structure of a software program.

Application Programming Interface (API)

A software program that has a defined way that other programs can interact with it. Often this service is provided with HTTP endpoints.

Artificial Intelligence (AI)

A technology where computer programs are capable of performing complex tasks that previously only intelligent beings could perform[2].

Code Analysis Tool (CAT)

Tool that should extract method and function information from a codebase. Further described in Chapter 7.

Large Language Model (LLM)

A type of machine learning model that excels at, a.o., natural language processing.

Lines of code (LOC)

The number of lines of code of a specific (part of a) software system.

System under test (SUT)

The system that is being tested by a test suite.

Test Indicator for Existing Software (TIES)

Suite of tools that implements UPSS. Further described in Chapter 7.

Test Suggestion and Scoring Tool (TSST)

Tool that suggests and scores tests based on method and function information. Further described in Chapter 7.

Tests Graphical User Interface (TGUI)

Tool that displays the suggested tests. Further described in Chapter 7.

[2] `https://www.britannica.com/technology/artificial-intelligence`

User-value Prioritized Suggestion Strategy (UPSS)

The test suggestion and prioritization strategy devised in this research. Further described in Chapter 5.

Version Control System (VCS)

A software tool that assists software maintainers to keep track of different versions of their code.

Appendix A

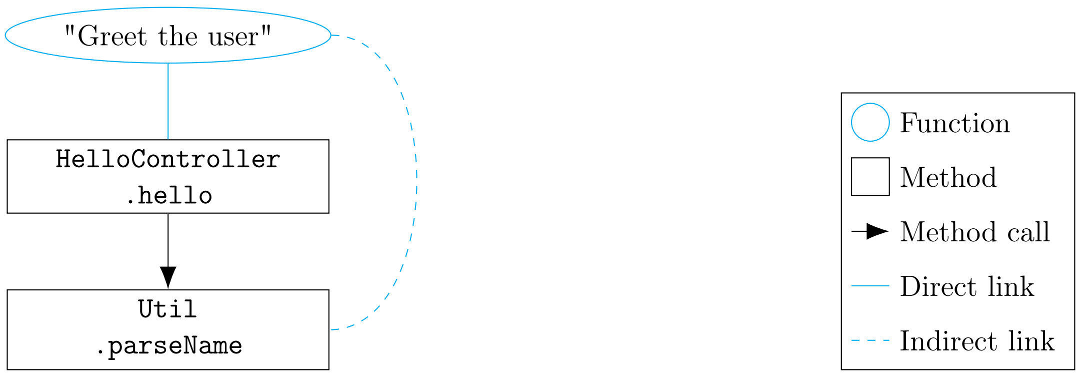
Code examples

A.1 Single method, single function

A.1.1 Code

```
class HelloController {

    /**
     * Greets the user.
     *
     * @param name The user's name
     * @return The user's name prepended with "Hello"
     * @function "Greet the user" 2 4
     */
    public String hello(String name) {
        return "Hello␣" + Util.parseName(name);
    }

}
```

A.1.2 Graph

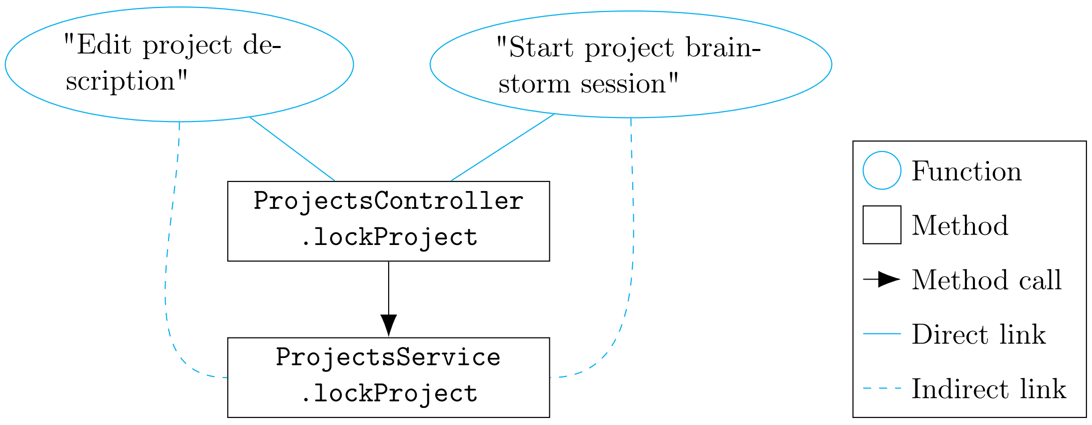


A.2 Single method, multiple functions

A.2.1 Code

```
class ProjectsController {

    /**
     * Locks a project.
     *
     * @param projectId The projects's id
     * @function "Edit project description" 2 5
     * @function "Start project brainstorm session" 2 3
     */
    @PutMapping("/lock/{projectId}")
    public void lockProject(@PathVariable int projectId) {
        return projectsService.lock(projectId);
    }

}
```

A.2.2 Graph

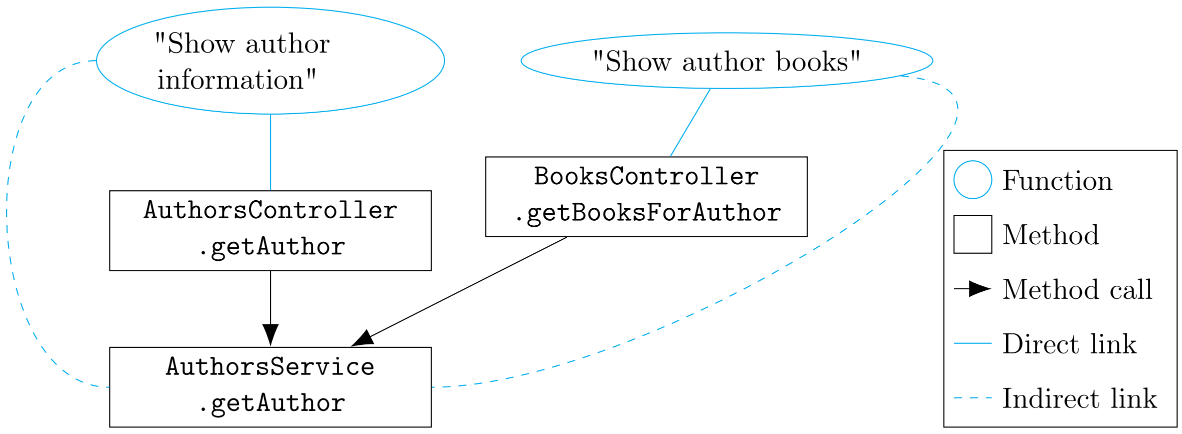

"Edit project description"
"Start project brainstorm session"
ProjectsController .lockProject
ProjectsService .lockProject
Function
Method
Method call
Direct link
Indirect link

A.3 Shared dependency

A.3.1 Code

```
class AuthorsController {

    /**
     * Get information of author.
     *
     * @param authorId The author's ID
     * @return The author
     * @function "Show author information" 4 5
     */
    @GetMapping("/author/{authorId}")
    public Author getAuthor(@PathVariable int authorId) {
        return authorsService.getAuthor(authorId);
    }

}

class BooksController {

    /**
     * Get books written by author.
     *
     * @param authorId The author's ID
     * @return A list of all books written by the author
     * @function "Show author books" 3 3
     */
    @GetMapping("/author/{authorId}/books")
    public List<Book> getBooksForAuthor(@PathVariable int
       ↪ authorId) {
        Author author = authorsService.getAuthor(authorId);

        return author.loadBooks();
    }

}
```

A.3.2 Graph

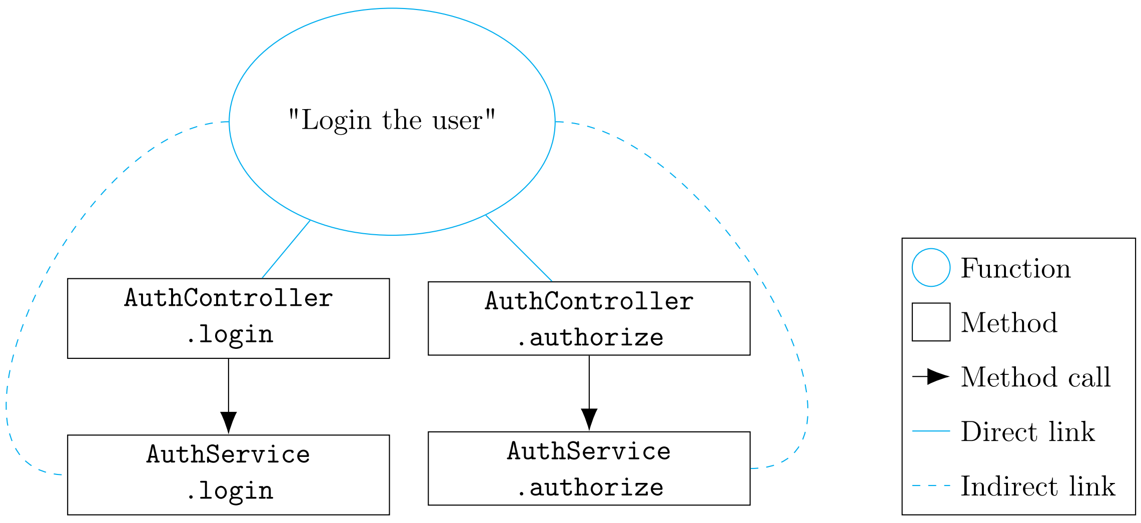

"Show author information"
"Show author books"
AuthorsController .getAuthor
BooksController .getBooksForAuthor
AuthorsService .getAuthor
Function
Method
Method call
Direct link
Indirect link

A.4 Multiple methods, single function

A.4.1 Code

```
/**
 * @functionDef userLogin "Login the user" 5 3
 */
class AuthController {

    /**
     * Logs the user in.
     *
     * @param username The provided username
     * @param password The provided password
     * @return A session token, provided authentication is
        ↪ succesful.
     * @function userLogin
     */
    @PostMapping("/login")
    public String login(String username, String password) {
        return authService.login(username, password);
    }

    /**
     * Authorizes the user.
     *
     * @param token The user's session token
     * @return The scopes the user has access to
     * @function userLogin
     */
    @PostMapping("/authorize")
    public String[] authorize(String token) {
        return authService.authorize(token);
    }

}
```

A.4.2 Graph


"Login the user"
AuthController .login
AuthController .authorize
AuthService .login
AuthService .authorize
Function
Method
Method call
Direct link
Indirect link

Appendix B

Example test suggestions

```
# Functional tests:

- Functional test for "Haal concept treinpaden op"
- Functional test for "Verwerk trein wijziging"
- Functional test for "Annuleer concept trein wijziging"
- Functional test for "Sla concept trein wijziging op"

# Unit tests:

- Unit test for "nl.prorail.trinity.pvb.
  ↪ controllers_ControllerUtils_createFunctionarisHeader"
- Unit test for "nl.prorail.trinity.pvb.
  ↪ repository_SpecifiekeDagTreinLinkedRepository_getSpecifiekeTrein
  ↪ "
- Unit test for "nl.prorail.trinity.pvb.
  ↪ model_TreinnummerreeksRange_setIsProrailTreinAsString"
- Unit test for "nl.prorail.trinity.pvb.
  ↪ repository_SpecifiekeDagRepository_removeIf"
- Unit test for "nl.prorail.trinity.pvb.
  ↪ services_InfrabeperkingService_releaseConceptInfrabeperkingLock
  ↪ "
```

Listing B.1: Example test suggestions in Markdown

Appendix C

JSON schemas

C.1 TSST input

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "methods": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          },
          "length": {
            "type": "integer"
          },
          "functions": {
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "dependencies": {
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "dependents": {
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        },
        "required": [
          "name",
          "length",
          "functions",
          "dependencies",
          "dependents"
        ]
      }
    },
    "functions": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          },
          "description": {
            "type": "string"
          },
          "userValue": {
            "type": "integer"
          },
          "stability": {
            "type": "integer"
          }
        },
        "required": ["name", "description", "userValue", "stability"]
      }
    }
  },
  "required": ["methods", "functions"]
}
```

C.2 TGUI input

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "tests": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "testType": {
            "type": "string",
            "enum": ["UNIT", "FUNCTIONAL"]
          },
          "title": {
            "type": "string"
          },
          "userValue": {
            "type": "number"
          },
          "stability": {
            "type": "number"
          },
          "complexity": {
            "type": "number"
          }
        },
        "required": [
          "testType",
          "title",
          "userValue",
          "stability",
          "complexity"
        ]
      }
    },
    "functions": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "methods": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          },
          "functions": {
            "type": "array",
            "items": {
              "type": ["object", "null"],
              "properties": {
                "name": {
                  "type": "string"
                },
                "description": {
                  "type": "string"
                },
```

```json
                "userValue": {
                  "type": "integer"
                },
                "stability": {
                  "type": "integer"
                }
              },
              "required": ["name", "description", "userValue", "stability"]
            }
          }
        },
        "required": ["name", "functions"]
      }
    }
  },
  "required": ["tests", "functions", "methods"]
}
```

Appendix D

Interviews

All interviews were held in Dutch, and therefore the questions were also written in Dutch. An English translation is provided.

D.1 Round 1

D.1.1 Questions

Dutch (original)

- In een software project, wat zijn volgens jou de meest nuttige type tests?
- Hoe bepaal je waar <type test (unit, functioneel, etc.)> voor te schrijven?
- Als je door bijvoorbeeld tijdsgebrek maar een deel van de gewenste tests kunt schrijven, welke factoren zouden bepalen welke tests je als eerste schrijft?
- Stel je een project voor waarbij functionaliteit niet volledig goed gedocumenteerd is, maar waar wel de wens is om functionaliteitstests in te voeren. Wat is je mening over het gebruiken van annotaties in de code (bijvoorbeeld Javadoc) om aan te geven welke functionaliteit door stukken code geleverd wordt?

English (translated)

- In a software project, what are the most useful types of tests according to you?
- How do you determine where to write <type of test (unit, functional, etc.)> for?
- If you could only implement a part of the suggested tests, for example due to lack of time; which factors would decide which tests you would implement first?
- Imagine a project wherein functionality is not fully well-documented, but where there is the wish to add functional tests. What is your opinion on using annotations in the code (for example Javadoc) to indicate what functionality is provided by which pieces of code?

D.1.2 Interviewee 1

Summary

Interviewee 1 states that it's best to have a seperate tester, someone other than a developer who works on the same project. Test plans were created on the basis of the demands of the software system as well as on initiative of the developers. Interviewee 1 also shares his experiences with testing on a different Ordina project where fault tolerance was lower than with the Java project. He states that he sees unit tests as purely testing classes, and only those classes, so other dependencies should be mocked. He describes that with the Java project they do have system tests where they run the application and test more at once than with unit tests, however, he also notes that those tests are less maintainable. Another major issue he mentions is state management during system tests; making sure the state of dependencies is clean. He states that he would not want to implement functional tests that automate user behavior for the Java project. While those tests seem nice, they are also very hard to maintain, which costs too much time/effort for the the Java project project. A desire he has for the Java project is having automated baseline tests, that run every time, as a kind of sanity check. With the Java project he finds that he does not have the time to design fully before he implements, causing unit tests also te be required to be rewritten when the implementation is changed later on. He states that for the central part, the model all other functionality builds upon, he deems it most important to write unit tests. He proposes determining the priority of tests on the complexity (if-statements and such) of the code they test. However, he later states that code that serves for the "most important" (to the user) functionality or that is used often should be tested first, slightly contradicting the earlier statement. Also, he states that complexity can decrease by refactoring the code, but that the importance of testing that specific code stays the same. A risk he describes is having that when a developer writes unit tests for code, the unit tests are written with the functionality too much in mind. A computer program could create more diverse tests. In the end he also states that functional tests can not be created based on looking at the source code alone.

D.1.3 Interviewee 2

Summary

Interviewee 2 states that tests are useful for code that changes a lot, is crucial to the workings of the application, or adds business value. He states that, sadly (according to him), the Java project has no automated integration tests. In an earlier project he worked on, they tried having automated functional tests with smartphone apps, however, these tests were deemed costing too much effort to maintain, and were therefore not justifiable. He states that he likes unit tests a lot, and that the help him in understanding what he has actually created and how it turned out. If possible, he would like to create unit tests for all code he writes. He states that he writes unit tests on a function [as in "method"] level. He would like to see more tests that test if the backend actually does what the frontend expects. He mentions that due to product owners always wanting to see features, there is no time for refactoring. For the Java project, he would like to first write mocks for external dependencies, such that tests can be executed without relying on these external dependencies. He finds functional tests that e.g. click through an application in an automated manner to only be useful for testing user interaction. He finds them not needed for testing functionality, since a functional unit test without interface could perform the same job. He thinks that relations between pieces of code can be determined automatically, without annotations. However, he does see use for annotations for stating the priority of functionality. He thinks that in a well-designed application, relations between code and functionality can be determined by software. However, he thinks that the importance of code or functionality cannot be deduced from the code.

D.2 Round 2

D.2.1 Questions

Dutch (original)

- Wat vind je van de gesugereerde tests?
- Wat mis je?

English (translated)

- What do you think of the suggested tests?
- What are you missing?

D.2.2 Interviewee 3

Summary

Interviewee 3 states that all methods are crucial to the application, in some sense. He states that besides unit tests he'd want functional tests or system tests; what he calls "deeper tests", that check whether something actually functions as intended. He states his interest in AI-related solutions for both suggesting and automatically generating tests. He thinks that AI would be better able to envision functional tests than a human, since a human relies on experience. He thinks the annotations are a good compact tool to communicate functionality, however, he does think that it could be cumbersome to write and maintain the annotations. He suggests letting a chatbot write the annotations, however he also sees that not all context required can be gathered from source code alone. He sees a need in describing how much effort implementing certain tests would take. Besides e.g. endpoints, Interviewee 3 thinks there are more methods in a backend that require specific functional testing, such as HLA service methods. He sees a use in also using code complexity/fault-proneness for test prioritization.

Interviewee 3 thinks that unit tests are useful, but that the accompanying overhead is not practical. When changing code, you also need to change the respective unit test(s), leading to double the amount of work, he states. In such cases, he thinks that unit tests cost more than they yield.

Interviewee 3 shares his concerns on how to deal with annotations for tests that have been implemented. He suggests adding a "checkbox" to the annotation that signifies whether a test has been created. Also, he thinks that the annotations should be as small as possible, preferably limited to one line.

Lastly, some confusion on the usage and keywords of the automated tool was cleared up. The difference between priority of a function defintion and the priority of a test suggestion was not clear.

D.2.3 Interviewee 1

Summary

After the current state and the output of the tool was explained to Interviewee 1, he was asked whether he was missing tests from the output and whether he agreed with the tests that were there. He noted that he would always keep a set of unit tests scoped to one class, and mock all dependencies. In general, he expressed his opinion that unit tests for methods that have very few branches and mostly call other classes and don't return anything are of little use. He would find it useful to have an overview of the context of a method with unit test suggestions. This context could include where the method lives, and what external outside of its class dependencies it uses. He states that giving a developer hints for writing unit tests, such as what input could possibly alter behavior of a method, would be useful.

D.3 Developer & Product Owner experiences

D.3.1 Questions for developers

Dutch (original)

- Hoe vond je het om de "entry points" van de applicatie te identificeren?
- Hoe vond je het gebruik van de annotaties in de `javadoc/JSDoc`?
- Zou je de aanpak met annotaties willen toepassen voor een complete codebase? Waarom wel of niet?
- Bij een volgende keer, wat zou je anders doen?

English (translated)

- How did you find it to identify the "entry points" of the application?
- How did you find it to use the annotations in the `javadoc/JSDoc`?
- Would you use the approach with annotations for a complete codebase? Why or why not?
- At a next time, what would you do differently?

D.3.2 Questions for Product Owners

Dutch (original)

- Hoe vond je het om gebruikswaarde en stabiliteit waarden te bedenken voor de functionaliteiten?
- Hoe vond je het invoeren van de waarden?
- Bij een volgende keer, wat zou je anders doen?

English (translated)

- How did you experience coming up with user value and stability values for the functions?
- How did you experience entering the values into the codebase?
- At a next time, what would you do differently?

D.3.3 Interviewee 3 (Developer Java project)

Summary

Interviewee 3 states that he found it easy to identify the entry points of the application, since the endpoints were easy to find. He thought the way of working with the annotations in the javadoc minimalistic and pleasant. He would use the same approach for a complete codebase, if there was a use for doing so. He states that currently, annotations would not be necessary for developers to find relations in the code, since they are clear already. In their process, using the javadoc annotations would be easy to use. Also, he found it clear what he had to do. While writing the annotations, he wondered how the approach would deal with methods that served multiple functions, since there was no example of such a case provided.

D.3.4 Interviewee 4 (Developer & Product Owner TypeScript project)

Summary

Interviewee 4 found it easy to identify the entry points of the application. As entry points he used endpoints. He found writing the annotations easy. Interviewee 4 used the method of defining all functions at class-level, and then referencing them from the methods as he found this a clear/well-organized approach. He found it a bit of a challenge to decide on the values for the user values and stabilities. He thought it was a bit vague what the values exactly mean, and what scale should be used. He defined a system of classes for himself, where user value 5 would mean that the application could absolutely not do without this function working correctly, and user value 1 would mean that it does not really matter to anyone if the function would not work correctly. Interviewee 4 thinks it is possible to use the approach for a complete codebase, and sees use in doing so. This way, one can identify where focus should be put for writing tests or making sure that code is correct. He does however think that it would take quite some effort, especially to identify what functions all exist in the codebase, and where functions start and end. Also, maintaining the annotations is also something that should be taken into consideration. He would probably not apply the approach for a large codebase due to the amount of effort required, but he thinks that it would be good to apply the approach to small (parts of) codebases where the impact would be large. If he were to go through the process again, he would first list and value the existing functions, before assigning them to entry points. He would also better define what a function contains and what it does not.

D.3.5 Interviewee 5 (Product Owner Java project)

Summary

Interviewee 5 states that he found it easy to fill in the user value and stability values, since he found the function descriptions clear and he has a good view on what they meant and how much these functions are used. Since Interviewee 5 did not fill in the values through the code annotations, the questionon filling in the values was not asked. Interviewee 5 says that if he were to go through the process again, he would not intentionally do things differently. He thinks that the values for functions should only be changed when the functions or their values actually change, not to shape the result to one's liking.

Appendix E

Evaluation interviews

E.1 The Java project

E.1.1 Participant 5

E.1.1.1 Summary

Participant 5 gives himself a 9 (out of 10) for his experience level w.r.t. the codebase of the Java project, a 9 for his experience level w.r.t. the specific part of the Java project that the user study focusses on, and a 7 for his experience level w.r.t. test automation. He understands the way the tests are suggested, only noting that they use a different naming convention for unit tests and that he would place unit tests for a certain class together in one file.

When asked what Participant 5 led to his suggested tests ranking (without metadata), he said that he looked at which functions a user of the Java project would find most important. For Participant 5's test sorting without metadata, we are not able to find weighing factors for UPSS that would give correlating results. On ranking the functional tests with metadata Participant 5said that he thinks the opinions would differ a lot on this ranking. He thinks all tests are important, and that it is therefore difficult to differentiate between them. His first and most significant priority was user value, and afterwards he also looked at the complexity values, he said.

This view is also represented in Participant 5's ranking of different functional test sortings, where UPSS(1,0,0,0.25)is ranked the highest. When asked how he ranked the different unit test sortings, he said that there was a certain set of tests that he deemed important, and that he checked whether the tests of that set were placed highly by the sorting. Mostly this was the case. When asked what formed this specific set, Participant 5 said that the methods of those unit tests served specific functions that he deemed important. When asked whether he considered implementation effort at all, he said that since he thought all tests would be easy to implement, implementation effort was no factor for him.

At the end of the interview, Participant 5 was shown TGUI. During this, he mentioned that indeed he thought user value was the most important. There were certain tests that he imagined would rise to the top, but did not. It was found that this was due to a certain collection of functions that were very similar and used a lot of the same underlying methods. This caused the user value for the unit tests for these methods to be extremely high, and therefore overshadowing the tests that Participant 5 expected to be high up. Participant

5 thought that possibly instead of summing user values, their maximum should be used for giving the user value of a method. The methods that Participant 5 expected the unit tests for to be high up, were found to have a relatively low complexity. This combined with the fact that a lot of functions got the highest user value rating of 5 meant that these tests were not at the top in UPSS's sorting. Participant 5 thought that probably the aforementioned collection of similar functions did not deserve the high user value ratings, since those functions were not a first priority for the end user, he thought. During discussion on UPSS and TIES, Participant 5 realizes that complexity is also a valuable metric, since it indicates where the bugs are actually at. He values the insights that the approach with combining user value and complexity gives, since complexity is normally "underwater" and less tangible, and thereby harder to incorporate in test prioritization.

E.1.1.2 Data

Experience questions (script items 2 to 4)

Question	Answer
On a scale of 1-10, how much experience do you have with the codebase of the Java project?	7
On a scale of 1-10, how much experience do you have with the code of specific part of the application that the case study focusses on?	6
On a scale of 1-10, how much experience do you have with test automation?	7

Tests arranged without metadata (script item 8)

Unit tests		
Test number	**Participant 5**	**UPSS(1,1,1,0.25)**
1	11	7
2	9	1
3	13	12
4	4	14
5	7	15
6	8	6
7	6	4
8	12	9
9	3	13
10	2	3
11	1	2
12	14	8
13	10	10
14	15	5
15	5	11

Average absolute difference	4.8
Spearman's rank correlation (r_s)	0.0428
Spearman's rank correlation p-value	0.5090

Functional tests		
Test number	**Participant 5**	**UPSS(1,1,1,0.25)**
1	15	2
2	13	15
3	1	12
4	12	4
5	10	10
6	5	11
7	9	8
8	7	5
9	8	9
10	2	1
11	11	6
12	6	13
13	3	7
14	4	3
15	14	14

Average absolute difference	4.13
Spearman's rank correlation (r_s)	0.1214
Spearman's rank correlation p-value	0.7955

Tests arranged with metadata (script item 11)

Unit tests		
Test number	**Participant 5**	**UPSS(1,1,1,0.25)**
1	3	11
2	2	4
3	8	6
4	1	1
5	7	13
6	12	15
7	11	9
8	6	2
9	13	10
10	9	8
11	4	5
12	5	3
13	10	12
14	15	14
15	14	7

Average absolute difference	2.93
Spearman's rank correlation (r_s)	0.6321
Spearman's rank correlation p-value	0.0143

Functional tests		
Test number	**Participant 5**	**UPSS(1,1,1,0.25)**
1	1	9
2	2	7
3	13	14
4	3	10
5	7	2
6	9	4
7	8	8
8	4	3
9	6	1
10	11	5
11	15	15
12	5	13
13	10	12
14	14	11
15	12	6

Average absolute difference	4.13
Spearman's rank correlation (r_s)	0.3500
Spearman's rank correlation p-value	0.2054

Test sorting rankings (script item 14)

Unit tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(0,1,0,0.25)	4	1
1.	UPSS(1,1,1,0)	4	1
3.	UPSS(1,1,1,0.25)	3	2
3.	UPSS(1,1,0,0.25)	3	2
5.	Random	1	4
6.	UPSS(1,0,0,0.25)	0	5

Functional tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,0,0,0.25)	4	0
2.	UPSS(1,1,0,0.25)	2	2
2.	UPSS(0,1,0,0.25)	2	2
4.	UPSS(1,1,1,0.25)	1	3
4.	Random	1	3

E.1.2 Participant 6

E.1.2.1 Summary

Participant 6 gives himself a 7 (out of 10) for his experience level w.r.t. the codebase of the Java project, a 9 for his experience level w.r.t. the specific part of the Java project that the user study focusses on, as he has just performed a refactor on the code of this part. He gives himself a 5 for his experience level w.r.t. test automation; he says that he does have experience with unit tests, but not that much with e.g. integration tests. He understands the types of tests suggested by UPSS, but says that there is a significant difference in usefulness between them. He would not prioritize writing unit tests for methods that only forward arguments to different methods and/or dependencies. He'd rather test methods that contain logic. He would rather write unit tests than functional tests.

While ranking the unit tests without metadata, Participant 6 noted that for one of the tests the team wondered whether its code was actually used by end users. Also, he found that he would rank quite some tests as having the same priority. He thinks that code that depends on a lot of preconditions in the state of the application is difficult and not that useful to test.

When ranking tests with metadata, Participant 6 notices that there are methods that have different complexity scores, even though they fulfill practically the same objective. Some investigation during the interview shows that this is due to the methods being written differently, even though indeed the functionality is practically the same. Participant 6 also notes that while some methods of themselves do not have a lot of logic in them, the chain reactions they set off throughout (other parts of) the application can be more fault-prone. Also, he notes that the difference in setup (and therefore implementation effort) needed can be large between unit tests, no matter the complexity of the method they test. He shared an anecdote about tests that were written, but were removed later when a refactor of code in a different class meant that mocks of the tests would have to be changed. As this was deemed to be too much effort, the test was removed. Participant 6 said that he ranked tests based on the impact of their code to the application.

Due to time constraints, the ranking of functional test sortings was skipped. While ranking the unit test sortings, Participant 6 notes that private methods might be hard to create unit tests for, and that is not done in general. Again, he notes that some unit tests that are ranked highly are for methods that do not contain a lot of logic. These methods are purely a conduit and only contain if-statements for e.g. type casting. Possibly using a different complexity metric could score these tests more justly, but it could also well be that the relative ratings would not change that much. He would also rank some tests higher because their methods contain a lot of logic, even though they don't get used as much as others. This reasoning is also visible in Participant 6's ranking of the sortings, where the sorting by UPSS(0,1,0,0.25), that only regards a test's complexity, is ranked the highest.

Participant 6 is shown TGUI. After a short introduction, he notes that for every project it should be defined where the focus is, and thus how the weighing factors should be tuned. He says that the Java project is backend-focused; meaning that the backend logic is the most important. This might also be due to the composition of the development team. In such a case, the user value weighing factor should be tuned down, as the user value is then less important than complexity and stability, he says. He says that the most complex logic of the Java project is in a different part of the codebase from the one the case study focuses on. All in all, he thinks that the approach UPSS uses with combining factors can be very useful. He thinks you currently should apply some manual manipulation to the results before using it as implementation guideline, and that the results should not be taken as exact law. He thinks it is extremely important to adjust the weighing factors to the project. He asks the question on how a user should deal with new features or code. He sees usefulness in being able to also consider functionality or changes that are yet to be implemented, as even for new functionality there might not always be the time/budget to implement complete test suites. Considering non-existing code is not possible with UPSS. He suggests the possibility of setting up a scaffold of non-implemented methods that could already be given user-assigned values. Participant 6 also suggests considering different parts of a codebase seperately or giving them differing overarching user value values, as these different parts might be of differing importance to the user.

At the end of the interview, Participant 6 notes that there is not only input and actions done by a human user, but that the Java project also responds to messages from a different software program. It is explained that it is also possible to put code annotations at the points where these messages enter the Java project, and Participant 6 says that it might even be more useful to test the handling of these messages than the input from the human users.

E.1.2.2 Data

Experience questions (script items 2 to 4)

Question	Answer
On a scale of 1-10, how much experience do you have with the codebase of the Java project?	7
On a scale of 1-10, how much experience do you have with the code of specific part of the application that the case study focusses on?	9
On a scale of 1-10, how much experience do you have with test automation?	5

Tests arranged without metadata (script item 8)

Unit tests		
Test number	**Participant 6**	**UPSS(1,1,1,0.25)**
1	6	7
2	3	1
3	14	12
4	15	14
5	5	15
6	7	6
7	10	4
8	9	9
9	4	13
10	1	3
11	13	2
12	11	8
13	8	10
14	12	5
15	2	11

Average absolute difference	4.40
Spearman's rank correlation (r_s)	0.1143
Spearman's rank correlation p-value	0.8360

Functional tests		
Test number	**Participant 6**	**UPSS(1,1,1,0.25)**
1	7	2
2	14	15
3	9	12
4	2	4
5	3	10
6	8	11
7	4	8
8	5	5
9	6	9
10	12	1
11	13	6
12	10	13
13	11	7
14	1	3
15	15	14

Average absolute difference	3.73
Spearman's rank correlation (r_s)	0.425
Spearman's rank correlation p-value	0.1185

Tests arranged with metadata (script item 11)

Unit tests		
Test number	**Participant 6**	**UPSS(1,1,1,0.25)**
1	2	11
2	3	4
3	4	6
4	1	1
5	5	13
6	10	15
7	4	9
8	12	2
9	15	10
10	8	8
11	6	5
12	7	3
13	5	12
14	13	14
15	14	7

Average absolute difference	4.33
Spearman's rank correlation (r_s)	0.254
Spearman's rank correlation p-value	0.3609

Functional tests		
Test number	**Participant 6**	**UPSS(1,1,1,0.25)**
1	3	9
2	13	7
3	4	14
4	1	10
5	11	2
6	2	4
7	7	8
8	8	3
9	9	1
10	14	5
11	15	15
12	10	13
13	12	12
14	5	11
15	6	6

Average absolute difference	4.93
Spearman's rank correlation (r_s)	0.0107
Spearman's rank correlation p-value	0.9697

Test sorting rankings (script item 14)

Unit tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(0,1,0,0.25)	5	0
2.	UPSS(1,1,1,0.25)	4	1
3.	UPSS(1,1,1,0)	3	2
4.	UPSS(1,0,0,0.25)	2	3
5.	UPSS(1,1,0,0.25)	1	4
6.	Random	0	5

E.1.3 Participant 7

E.1.3.1 Summary

Participant 7 gives himself a 7.5 (out of 10) for his experience level w.r.t. the codebase of the Java project, a 1 for his experience level w.r.t. the specific part of the backend of the Java project that the user study focusses on, and a 4 for his experience level w.r.t. test automation. This lack of experience with this part of the application comes due to Participant 7working on a different part of the backend that uses a different frontend. He does therefore have experience with the Java project's backend. If code becomes very complex, he does tend to write tests for it. He notes that the Java project is a difficult project to develop tests for, due to the complicated setup required for a test with a lot of manual creation of mocking data involved. Due to Participant 7's lack of knowledge the specific part of the codebase that UPSS was applied to through TIES, all items of the interview script that involve ranking of tests or sortings were skipped. Questions that aim to qualitatively examine how Participant 7 would sort tests and what he thinks of UPSS and TIES were kept/added.

Participant 7 thinks that he could implement the tests such as they are suggested, if he would look into the code a bit. For the unit tests he might miss some edge conditions. Looking at code coverage achieved could help him cover all cases, he says.

When asked what tests Participant 7 would implement first for an application such as the Java project, if he had a limited amount of time available, he says that he would start with tests that test the integration between front- and backend, so tests that test a small scenario. He states that there are business rules in the application that he would like to see be tested, since as apparently the rules were important enough to be implemented, they should also be verified to be actually implemented correctly. Participant 7 would also like to see tests for methods that are called in lots of places.

After being shown TGUI and being asked what he thinks, Participant 7 says that using an approach where one can give rankings and adjust weighing factors such that one can see where they would have to work on with limited costs seems useful to him. Notable to him were the high placement of a collection of tests that all concern the same functionality and the high placement of generic methods that are called a lot. Some discussion is held on how user values should be combined and how weighing factors are applied. Participant 7 understands the choices that were made.

E.1.3.2 Data

Experience questions (script items 2 to 4)

Question	Answer
On a scale of 1-10, how much experience do you have with the codebase of the Java project?	7.5
On a scale of 1-10, how much experience do you have with the code of specific part of the application that the case study focusses on?	1
On a scale of 1-10, how much experience do you have with test automation?	4

E.1.4 Participant 8

E.1.4.1 Summary

Participant 8 gives himself a 10 (out of 10) for his experience level w.r.t. the codebase of the Java project, a 6 for his experience level w.r.t. the specific part of the Java project that the user study focuses on, and a 7.5 for his experience level w.r.t. test automation. He says that he has touched every part of the codebase. The relatively low experience level w.r.t. the specific part of the Java project that the user study focuses on is due to this part being recently refactored by someone else.

When looking at the unsorted selection of suggested tests, Participant 8 mentions that some functional tests that only test whether some data is correctly displayed, he would not implement. Concerning the unit tests he says that he would expect a more explicit, human-readable name. For the "simple" methods that only retrieve data from a source, he would not implement unit tests. When asked to rank the tests, he mentions that he cares less about the lowest ranked tests, since they would not be implemented anyways.

After ranking the suggested tests without metadata, Participant 8 stated that he ranked the tests based on his knowledge of how much logic/complexity the functions and methods contain. This statement matches well with his actual suggested test rankings, as his rankings correlate significantly (p-value < 0.05) well with rankings made by UPSS with the complexity weighing factor turned up or with the complexity metric being the only source for sorting. His ability to estimate both the method and function complexities this well could be explained by Participant 8's extensive experience with the codebase and the project.

While ranking the suggested tests with metadata, Participant 8 notices that some functions have been given higher user value values than he would have given them based on his experience with their usage. He also notes a function that he thinks should have been described more specifically, as currently it seems that the function is only linked with one entry point, while its description would suggest multiple entry points being involved.

When being asked how he ranked the suggested tests with metadata, he said that he incorporated user value and that he considered stability only on extreme values. He said that when the stability is high, one could say that the code would exist for a long time, and that therefore there is extra usefulness in testing it. However, he also realizes that one could say that as the code would not change for a long time, automated testing is not that useful after the function has been tested manually once. For functions with a low stability value, he said that he would not be inclined to write tests for them, since this function would probably only be a trial and would not be around for long. If a stability value would be low because the code was due for a refactor, he would not be inclined to write a unit test for it, since it would have to be rewritten. He said he only partially looked at the indicated complexity values, since he felt that they did not agree with his own feelings for the complexity of the code. Despite this, there was a significant (p-value < 0.05) positive correlation between Participant 8's ranking and that of UPSS(1,1,1,0.25) for both unit and functional tests.

While comparing different test sortings, Participant 8 noticed a method that was linked to a function that had a high user value. However, he noted that if this method were to not give the correct results, the function would mostly work just as well, indicating that the given user value did not apply to all linked methods.

After being shown UPSS(1,1,1,0.25)'s test sortings (of which the unit test sorting was his best ranked unit test sorting) and asked whether he agreed with these sortings and why/why not, Participant 8 again stated that he ranked items that he thought contained little complexity lowly. He said that he felt that the current complexity estimation was too simple, and that possibly a combination of cyclomatic complexity and counting LOC would be better. When asked to compare UPSS(1,1,1,0.25)'s function test sorting with that of UPSS(0,1,0,0.25)(the sorting that scored highest in the pairwise ranking), Participant 8 said that he thought they were very comparable and that he could have picked UPSS(1,1,1,0.25)'s sorting just as well. Without knowing what the sortings were based on, he mentioned that UPSS(0,1,0,0.25)'s sorting relied more on the functions' complexity values. After being told that UPSS(0,1,0,0.25)'s sorting relied **only** on the functions' complexity values, Participant 8 noted that still functions that he deemed to have high user value were at the top. The observation that there seems to be a positive correlation between a function's user value and complexity is discussed.

After being shown TGUI, Participant 8 was asked what he thinks of the approach of UPSS and TIES. He said the complexity metric needed to be tweaked and that it is important that the user value is given by the "business" members of the project team and the stability values by the developers. If by doing so a good combination of what is important to a developer and what is important to the user is reached, Participant 8 says that he could definitely use the approach. He thinks that the approach with code annotations could cost a lot of effort, and that he would probably not do it that way. But nevertheless, he thinks the results are useful. Not only for indicating which tests exactly are the most important, but also as a means of support for conversations with e.g. product owners to show that there is usefulness in implementing tests (and not only for the pieces of code that the product owner deems important). He thinks this could lead to a more stable product, which would benefit both the developers and the product owner. He suggests integrating the approach into the scrum way-of-working, by possibly asking the "business side" of the team to give an importance value when creating backlog items and using the scrum story points as an indication for complexity. Also, he says that in a real-world application he might like to use a scale of 1 to 10 (instead of 1 to 5), and suggests that a team could firstly look at what their functionality with the highest user value would be to calibrate what would become a 10. Lastly, Participant 8 says that the simple act of asking the "business side" of the project to give user value values to functionality could give helpful insights to everyone, and especially the developers.

E.1.4.2 Data

Experience questions (script items 2 to 4)

Question	**Answer**
On a scale of 1-10, how much experience do you have with the codebase of the Java project?	10
On a scale of 1-10, how much experience do you have with the code of specific part of the application that the case study focusses on?	6
On a scale of 1-10, how much experience do you have with test automation?	7.5

Tests arranged without metadata (script item 8)

Unit tests				
Test	**Participant 8**	**UPSS(1,1,1,0.25)**	**UPSS(1,2,1,0.25)**	**UPSS(0,1,0,0.25)**
1	14	13	12	15
2	3	6	5	2
3	13	14	14	13
4	10	1	3	7
5	6	11	10	10
6	15	7	9	9
7	4	2	1	1
8	5	5	6	5
9	1	4	4	3
10	12	12	13	12
11	7	9	7	6
12	8	8	8	8
13	11	15	15	14
14	9	3	2	4
15	2	10	11	11

Participant 8 vs. UPSS(1,1,1,0.25)	Average absolute difference	3.87
	Spearman's rank correlation (r_s)	0.4393
	Spearman's rank correlation p-value	0.1014

Participant 8 vs. UPSS(1,2,1,0.25)	Average absolute difference	3.33
	Spearman's rank correlation (r_s)	0.5071
	Spearman's rank correlation p-value	0.0537

Participant 8 vs. UPSS(0,1,0,0.25)	Average absolute difference	2.53
	Spearman's rank correlation (r_s)	0.6571
	Spearman's rank correlation p-value	0.0096

Functional tests				
Test	**Participant 8**	**UPSS(1,1,1,0.25)**	**UPSS(1,2,1,0.25)**	**UPSS(0,1,0,0.25)**
1	15	15	15	15
2	8	6	7	7
3	14	14	14	13
4	1	3	2	3
5	6	1	1	1
6	12	2	4	6
7	2	7	3	2
8	11	13	12	11
9	3	10	11	10
10	10	5	5	4
11	13	12	13	14
12	7	4	6	5
13	4	8	10	12
14	9	11	8	8
15	5	9	9	9

Participant 8 vs. UPSS(1,1,1,0.25)	Average absolute difference	3.47
	Spearman's rank correlation (r_s)	0.4964
	Spearman's rank correlation p-value	0.0598
Participant 8 vs. UPSS(1,2,1,0.25)	Average absolute difference	2.80
	Spearman's rank correlation (r_s)	0.5786
	Spearman's rank correlation p-value	0.0287
Participant 8 vs. UPSS(0,1,0,0.25)	Average absolute difference	2.93
	Spearman's rank correlation (r_s)	0.575
	Spearman's rank correlation p-value	0.0300

Tests arranged with metadata (script item 11)

Unit tests			
Test number	**Participant 8**	**UPSS(1,1,1,0.25)**	**UPSS(1,2,1,0.25)**
1	14	12	11
2	5	1	1
3	3	11	12
4	12	15	14
5	6	7	10
6	2	4	6
7	7	13	13
8	15	8	5
9	4	5	3
10	13	10	8
11	1	2	2
12	8	6	9
13	11	14	15
14	10	3	4
15	9	9	7

Participant 8 vs. UPSS(1,1,1,0.25)	Average absolute difference	3.33
	Spearman's rank correlation (r_s)	0.5429
	Spearman's rank correlation p-value	0.0365

Participant 8 vs. UPSS(1,2,1,0.25)	Average absolute difference	4.13
	Spearman's rank correlation (r_s)	0.3536
	Spearman's rank correlation p-value	0.1961

Functional tests			
Test number	**Participant 8**	**UPSS(1,1,1,0.25)**	**UPSS(1,2,1,0.25)**
1	13	14	14
2	9	4	2
3	1	10	13
4	3	9	11
5	8	5	6
6	2	2	3
7	5	6	7
8	4	1	1
9	11	8	5
10	14	13	12
11	15	15	15
12	10	11	10
13	6	7	8
14	12	12	9
15	7	3	4

Participant 8 vs. UPSS(1,1,1,0.25)	Average absolute difference	2.53
	Spearman's rank correlation (r_s)	0.6607
	Spearman's rank correlation p-value	0.0073

Participant 8 vs. UPSS(1,2,1,0.25)	Average absolute difference	3.47
	Spearman's rank correlation (r_s)	0.3964
	Spearman's rank correlation p-value	0.1435

Test sorting rankings (script item 14)

Unit tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,1,1,0.25)	5	0
2.	UPSS(1,1,1,0)	4	1
3.	UPSS(1,1,0,0.25)	3	2
4.	UPSS(1,0,0,0.25)	2	3
5.	UPSS(0,1,0,0.25)	1	4
6.	Random	0	5

Functional tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(0,1,0,0.25)	4	0
2.	Random	3	1
3.	UPSS(1,1,0,0.25)	2	2
4.	UPSS(1,1,1,0.25)	1	3
5.	UPSS(1,0,0,0.25)	0	4

E.2 The TypeScript project

E.2.1 Participant 1

E.2.1.1 Summary

Participant 1 gives himself a 7 (out of 10) for his experience level w.r.t. the codebase of the TypeScript project, a 6 for his experience level w.r.t. the specific part of the TypeScript project that the user study focusses on, and a 7 for his experience level w.r.t. test automation. He understands the objective of the research. With his experience with the tests that exist in the TypeScript project, he assumed functional tests to mean end-to-end tests. He agrees with the types of tests suggested. He notes that he does not know the exact contents of all methods related to the suggested unit tests. W.r.t. the ranking of the suggested unit tests without metadata, he notes that he values tests highly that test methods that are called in a lot of places. Without having insights into the metadata, he unsolicitly identifies that a certain part of the codebase often contains faults. Methods from this part of the codebase also have high complexity indications; holding 6 spots in the top-8 highest complexity methods of the entire section of the codebase that the case study focuses on, including the top 2 spots. He also states that tests for these methods take relatively much effort to implement. This in contrast to certain methods that he also thinks are important to test, but would be easier to implement tests for. We can see the results of this reasoning in his ranking of the unit tests without metadata. In this ranking unit tests for methods that are deemed important based on user value, but are relatively non-complex such that implementation effort required is relatively low, are ranked as the tests to implement first. UPSS agrees with these rankings. It should be noted that Participant 1 at this point did not know of the user value values given to the functions by the product owner; therefore it is observed that Participant 1 and the product owner share very similar views on user value. He deems unit tests for methods that do not contain significant logic to be of relatively low value, even though they are easy to implement. For functional tests, he valued tests for functions that are most used highly, followed by tests for functions that are fault-prone. Again, it appears that Participant 1's views on user value mostly align with those of the product owner and the prioritization strategy of UPSS, as the resulting functional test rankings show a significant (p-value < 0.05) positive correlation with the ranking by UPSS.

After the metadata and principle of stability (and how it could influence the prioritization of tests) was explained, Participant 1 ranked suggested tests with their metadata available to him. Now, the correlation with the ranking by UPSS was even significantly stronger. He stated that he used the same approach to the ranking, with the added information now helping him with prioritizing tests for code that he did not fully know.

When starting with the pair wise comparison ranking of the different test sortings, Participant 1 was explained that he should rate the sortings based on how he'd feel tests should be sorted, not how he thinks that UPSS would do it. For unit tests, he rated sortings that take all factors (user value, complexity, and stability) into account the highest, followed by user value only sorting and user value + complexity sorting. The random ordering and the sorting based on complexity alone were rated the lowest. For the sortings of functional tests, the random ordering was clearly deemed the poorest. The other sortings were all deemed almost equally important, with UPSS(1,1,1,0.25) and UPSS(0,1,0,0.25)only just being ranked higher than UPSS(1,0,0,0.25) and UPSS(1,1,0,0.25). Participant 1 stated that he looked at the balance of complexity and how often functionality was used, and that he did not consider stability much. He feels that if a test has to be altered since the implementation or functionality has changed, that that simply has to be done and is not a problem. He feels that a low stability is not a reason to not write a test. He noted that about halfway through ranking the functional test sortings, he shifted his focus from complexity to user value. This was done because he realized that he feels that with functional tests you should focus on functions that are used most often, since functional tests cost more time to create. Functions that would "break" the entire application when they are not functioning correctly, should have a high priority to be tested, he feels.

After being shown the sortings of UPSS(1,1,1,0.25), Participant 1 noted again that for functional tests, he would weigh the user value factor higher. However, he does note that he deems the balance between user value and fault-proneness as important. He also states that he thinks that overall code that is more important is more fault-prone as well.

At the end of the interview, Participant 1 was let to use TGUI. He noted that again he thought that the sorting with the default weighting factors (UPSS(1,1,1,0.25)) was already very useful. He shared a fear that applying UPSS to an entire codebase would take significant effort, since user value and stability values would have to be decided for all functions. He feels that someone who has experience with the codebase could instead already prioritize tests. Finally, Participant 1 suggested using AILLMs for automating the writing of code annotations and assigning values.

As this was the first interview held, lessons were learned about how to get the best results from an interview as possible. For example, the first pairwise comparison tool used did not randomize the pairings; first showing all pairings for one test sorting. This meant that insights that a participant gains during the ranking process are unequally applied to the sortings. Because of this, a pairwise comparison tool was developed that does randomize the pairings. Secondly, it was felt that the participant might have been given slightly too much information about the workings of UPSS early on, leading to his opinions being influenced by our visions and opinions. However, in the interview he did explicitly state that he agrees with the ideas of UPSS, and most of his views were shared independently, and not as a response to something being said by us. It does mean that the quantitative results from this interview could be unfairly positive. Nevertheless, valuable insights and observations were gathered.

E.2.1.2 Data

Experience questions (script items 2 to 4)

Question	Answer
On a scale of 1-10, how much experience do you have with the codebase of the TypeScript project?	7
On a scale of 1-10, how much experience do you have with the code of specific part of the application that the case study focusses on?	6
On a scale of 1-10, how much experience do you have with test automation?	7

Tests arranged without metadata (script item 8)

Unit tests		
Test number	**Participant 1**	**UPSS(1,1,1,0.25)**
1	15	15
2	9	3
3	3	9
4	14	7
5	5	5
6	10	14
7	6	10
8	1	1
9	12	12
10	4	2
11	11	8
12	8	13
13	7	6
14	2	11
15	13	4

Average absolute difference	3.73
Spearman's rank correlation (r_s)	0.3679
Spearman's rank correlation p-value	0.1829

Functional tests		
Test number	**Participant 1**	**UPSS(1,1,1,0.25)**
1	12	14
2	10	4
3	1	2
4	4	3
5	11	10
6	14	6
7	2	5
8	5	12
9	3	1
10	7	7
11	13	11
12	8	9
13	6	8
14	9	13

Average absolute difference	2.86
Spearman's rank correlation (r_s)	0.5736
Spearman's rank correlation p-value	0.0367

Tests arranged with metadata (script item 11)

Unit tests		
Test number	**Participant 1**	**UPSS(1,1,1,0.25)**
1	12	12
2	2	14
3	8	9
4	5	6
5	10	2
6	3	4
7	6	3
8	15	13
9	1	1
10	7	5
11	4	8
12	11	11
13	9	7
14	14	15
15	13	10

Average absolute difference	2.67
Spearman's rank correlation (r_s)	0.5393
Spearman's rank correlation p-value	0.0430

Functional tests		
Test number	**Participant 1**	**UPSS(1,1,1,0.25)**
1	5	5
2	3	4
3	1	3
4	10	10
5	4	8
6	9	11
7	8	6
8	4	2
9	12	14
10	2	1
11	11	13
12	14	12
13	6	9
14	13	7

Average absolute difference	2.07
Spearman's rank correlation (r_s)	0.8119
Spearman's rank correlation p-value	< 0.0001

Test sorting rankings (script item 14)

Unit tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,1,1,0.25)	4	1
1.	UPSS(1,1,1,0)	4	1
3.	UPSS(1,0,0,0.25)	3	2
4.	UPSS(1,1,0,0.25)	2	3
5.	UPSS(0,1,0,0.25)	1	4
5.	Random	1	4

Functional tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,1,1,0.25)	3	1
1.	UPSS(0,1,0,0.25)	3	1
3.	UPSS(1,0,0,0.25)	2	2
3.	UPSS(1,1,0,0.25)	2	2
5.	Random	0	4

E.2.2 Participant 2

E.2.2.1 Summary

Participant 2 gives himself a 7 (out of 10) for his experience level w.r.t. the codebase of the TypeScript project, a 7 for his experience level w.r.t. the specific part of the TypeScript project that the user study focusses on, and a 5 for his experience level w.r.t. test automation. He does share that there might be new functionality in the specific part that he does not yet know about. He finds the types of suggested tests to be sensible and logical. He thinks that testing methods of only 1 line of code should not be done, as these methods only serve as conduits to lower level methods. He would possibly add integration tests, that would test multiple methods at once. An observation that is made is that in the case of an API, a product owner needs to make a clear distinction whether to state functionality as functions of the API itself, or functions of the entire software system that the API is part of, based on on what level tests would be implemented. He thinks that there is a risk in testing an API only from the perspective of a human user of a first-party application that uses the API, since human user input can be trusted more than input from different software systems that interface with the API. Participant 2 thinks that writing tests for code that has been in the codebase for a long time is less useful. This comes back multiple times during the interview; Participant 2 herewith differs significantly in vision from UPSS.

After ranking suggested tests without metadata available, Participant 2 noted that he prioritized tests for functions/code that he knows currently contain bugs. Then, he rated security-related items highly. Then, he listed a "quick-win"; a test that would be somewhat useful and take little time to implement. And the tests that he rated as the ones that he would implement the latest are tests for code/functionality that he knows has been in the codebase for a long time, and where there have been no reported bugs (all errors reported were user-error). This difference in view compared to UPSS is also visible in the low correlation between Participant 2's ranking and that of UPSS(1,1,1,0.25).

Participant 2 notes, after being explained the metadata, that the complexity overview could already be a good starting point for refactor prioritization. Despite the earlier observed difference in vision on stability, Participant 2's ranking of suggested unit tests has a significant (p-value < 0.05) high correlation factor with UPSS(1,1,1,0.25)'s ranking. Participant 2 states that the complexity metadata influenced his prioritization positively. Something that is discussed is a difference in implementing different versions of the same functionality. This could be done by having seperate entry points, or by having one entry point that takes an extra parameter. This influences the complexity estimations done for functional tests by the tool. While ranking the tests with metadata, Participant 2 notes that there were only a few tests that he would actually implement, and that he therefore struggled to rank them all the way through. He also suggests the term "method tests" instead of "unit tests". A remark that Participant 2 makes is that according to him "a little wasted effort is still wasted effort". He indicated that implementation effort is not that important to him. Also he would prioritize testing items that affect more customers than one. Participant 2 notes that legacy code that is no longer used should be removed, and can now spoil test suggestions and rankings.

When ranking the suggested functional tests with metadata, an observation is made that functions exist in the codebase with surprisingly low complexity values. It is reasoned that this is partly due to a lot of logic being in libraries (which do not need to be tested by the team), and partly due to logic being in pieces of code that the code analyzer (CAT) was not able to link to functions. He shares that he does not fully agree with all values assigned by the product owner; he states that he does however use the product owner's values in his ranking, however, he does find this difficult. He shares that in cases that functions overlap in implementation code, he would prioritize developing a test for one of them, as this would simultaneously test the other function. There is no significant correlation between Participant 2's ranking of the suggested functional tests and UPSS(1,1,1,0.25)'s. When asked how he ranked the suggested functional tests, Participant 2 says that he looked at the combination of user value and stability, where a higher user value would increase a test's priority, and a higher stability would decrease a test's priority. This is in line with his statement made before. For ranking functional tests, he does not use the complexity metadata much, since he states that the complexity of a function implementation would not directly influence the amount of effort required to implement a test for this function. He says that the complexity would then influence the code coverage achieved, but he deems code coverage as a non-useful metric to use as a target. When we use these statements to deduce weighing factors for UPSS by setting $W_c = 0$ and invert W_s to -1 and thereby come to UPSS(1,1,-1,0.25), we see that these rankings match extremely closely.

While ranking the different sortings, Participant 2 shares his view on why he thinks that a low stability should increase the priority of a suggested test; implementing a test for code that is bound to change would make the test serve as a regression test. Participant 2 also talks about whether function user values should be summed together for a unit test or not. He thinks they should, since the functions are seperate items. Another view the participant shares is that, according to him, larger organizations innovate a lot less than smaller organizations, and that this influences the approach to testing; if you're changing the code a lot, writing tests makes less sense than when the code is more stable. He shares that when ranking the unit test sortings, he is looking at outliers for method functions; if a method serves a function that he thinks needs testing, he prioritizes the unit test for this method higher. He also says he looks at certain specific unit tests to choose a sorting, moreso than comparing the sorting on all tests, as regarding all tests in a sorting would take too much time. He shares that he deems user value and stability as more influential than complexity. For some units, he says, no test should be written, but a refactor should occur.

The views shared by Participant 2 during the interview are also resonated in his rankings of the different sortings. Sortings with $W_s = 0$ score well, while the sorting with $W_{uv} = 0$ score poorly. In discussion afterwards, the participant understood the choices made for how to incorporate stability, however, he stood by his views. The random ordering of suggested unit tests scored relatively well; it seems that, and discussion seems to confirm this, this is partly due to the perceived quality of this specific random ordering, and partly due to Participant 2's ideas differing too much from the idea behind (the default weighing factors of) UPSS. Also, his way of looking at outliers could have influenced the rankings. He says that there could potentially be a combination of weighing factors that would fit his opinions perfectly. When using TGUI, he quickly finds such a combination of weighing factors. Some discussion is held on which lines of a method should count towards the LOC metric. Participant 2 thinks lines containing only the opening and closing brackets of a method should not be counted, as these 2 lines could influence the score calculation significantly.

Regarding the UX of TGUI Participant 2 notes that the weighing factors settings should probably be hidden away and be set only once. Having them too visible would lead a user to possibly continously change the weighing factors until a sorting of their liking is achieved. Setting the weighing factors beforehand would let the user focus on what they think is important, and then possibly let them to accept the results as is. Participant 2 says that in his current work environment he would use UPSS and TIES. In a new work environment he would not, as implementing tests would have a low priority overall there. He notes that during the evolution of a software project and the organization it sits in the weighing factors could change, as focus shifts from innovation to stability for example, and project lead get to know the developers better and therefore less code review could be necessary. Participant 2 suggests combining UPSS with LLMs. He says that he would currently not blindly trust tests generated by a LLM, but using UPSS one could prioritize the validation of tests given by the LLM.

E.2.2.2 Data

Experience questions (script items 2 to 4)

Question	Answer
On a scale of 1-10, how much experience do you have with the codebase of the TypeScript project?	7
On a scale of 1-10, how much experience do you have with the code of specific part of the application that the case study focusses on?	7
On a scale of 1-10, how much experience do you have with test automation?	5

Tests arranged without metadata (script item 8)

Unit tests			
Test number	**Participant 2**	**UPSS(1,1,1,0.25)**	**UPSS(1,0,-1,0.25)**
1	6	15	14
2	11	3	1
3	9	9	12
4	1	7	7
5	5	5	6
6	10	14	15
7	14	10	8
8	7	1	2
9	15	12	10
10	4	2	4
11	2	8	11
12	13	13	13
13	12	6	5
14	8	11	9
15	3	4	3

Participant 2 vs. UPSS(1,1,1,0.25)	Average absolute difference	3.87
	Spearman's rank correlation (r_s)	0.3875
	Spearman's rank correlation p-value	0.1617

Participant 2 vs. UPSS(1,0,-1,0.25)	Average absolute difference	4.40
	Spearman's rank correlation (r_s)	0.1929
	Spearman's rank correlation p-value	0.4911

Functional tests			
Test number	**Participant 2**	**UPSS(1,1,1,0.25)**	**UPSS(1,1,-1,0.25)**
1	2	14	11
2	6	4	9
3	3	2	5
4	11	3	1
5	4	10	10
6	8	6	12
7	13	5	8
8	10	12	6
9	14	1	3
10	9	7	7
11	7	11	14
12	1	9	13
13	12	8	2
14	5	13	4

Participant 2 vs. UPSS(1,1,1,0.25)	Average absolute difference	5.71
	Spearman's rank correlation (r_s)	-0.4374
	Spearman's rank correlation p-value	0.1252

Participant 2 vs. UPSS(1,1,-1,0.25)	Average absolute difference	6.14
	Spearman's rank correlation (r_s)	-0.5516
	Spearman's rank correlation p-value	0.0409

Tests arranged with metadata (script item 11)

Unit tests			
Test number	**Participant 2**	**UPSS(1,1,1,0.25)**	**UPSS(1,0,-1,0.25)**
1	14	13	13
2	3	15	8
3	10	9	15
4	8	6	5
5	2	2	4
6	4	4	6
7	7	3	2
8	13	14	14
9	1	1	1
10	9	5	3
11	6	8	7
12	12	12	12
13	5	7	9
14	15	10	10
15	11	11	11

Participant 2 vs. UPSS(1,1,1,0.25)	Average absolute difference	2.27
	Spearman's rank correlation (r_s)	0.6143
	Spearman's rank correlation p-value	0.0148

Participant 2 vs. UPSS(1,0,-1,0.25)	Average absolute difference	2.67
	Spearman's rank correlation (r_s)	0.6929
	Spearman's rank correlation p-value	0.0042

Functional tests			
Test number	**Participant 2**	**UPSS(1,1,1,0.25)**	**UPSS(1,1,-1,0.25)**
1	8	5	9
2	9	4	8
3	1	3	1
4	10	10	10
5	2	8	2
6	13	11	14
7	11	6	12
8	6	2	4
9	12	14	11
10	7	1	5
11	3	13	3
12	4	12	6
13	14	9	13
14	5	7	7

Participant 2 vs. UPSS(1,1,1,0.25)	Average absolute difference	4.29
	Spearman's rank correlation (r_s)	0.2264
	Spearman's rank correlation p-value	0.4388

Participant 2 vs. UPSS(1,1,-1,0.25)	Average absolute difference	1.00
	Spearman's rank correlation (r_s)	0.9516
	Spearman's rank correlation p-value	0.0002

Test sorting rankings (script item 14)

Unit tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,1,0,0.25)	5	0
2.	UPSS(1,0,0,0.25)	4	1
3.	Random	3	2
4.	UPSS(1,1,1,0)	2	3
5.	UPSS(1,1,1,0.25)	1	4
6.	UPSS(0,1,0,0.25)	0	5

Functional tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,0,0,0.25)	4	0
2.	UPSS(1,1,0,0.25)	3	1
3.	UPSS(1,1,1,0.25)	2	2
4.	UPSS(0,1,0,0.25)	1	3
5.	Random	0	4

E.2.3 Participant 3

E.2.3.1 Summary

Participant 3 gives himself a 8 (out of 10) for his experience level w.r.t. the codebase of the TypeScript project, a 7 for his experience level w.r.t. the specific part of the

TypeScript project that the user study focusses on, and a 9 for his experience level w.r.t. test automation. Participant 3 thinks the suggested tests are clear and understandable. Also, unit and functional tests are the types of tests that he would write. He does say that he would probably not write unit tests for every method in a codebase. Writing tests for small methods that do not do much more than calling a dependency would lead mostly to testing dependencies, which is deemed less useful since these dependencies should be well-tested themselves. He would rather test these small methods through other "unit tests" for larger methods that call the small method.

While ranking the suggested unit tests without metadata, Participant 3 noted that he was subconciously scoring the tests based on impact to customers. Some of the methods that unit tests were suggested for in the list that was handed to him, only serve one specific customer (i.e. provide functionality that is specific to that one customer). Since only a few users would be affected if this method would fail, Participant 3 would be less likely to write a test for it. In contrast, methods that affect the security and confidentiality of the application would sooner necessitate a test, according to Participant 3. He notes that for some methods it is the case that if they would fail, it would be immediately noticed in the acceptance environment that the company has. Therefore, he deems writing an automated test for such methods less necessary, paradoxically. Looking at his results, we can see that Participant 3 indeed seems to prioritize user value (even without knowing the assigned values), as UPSS(2,1,1,0.25) has a significant (p-value < 0.05) positive correlation with his ranking for unit tests. This seems to vouch for his high indicated experience with the codebase. For functional tests, it is hard to find a correlation with output of UPSS. It seems that possibly Participant 3 values user value highly and complexity less.

When asked to rank the suggested tests with metadata available, Participant 3 notes that he can reason using stability both ways; either as having a positive relation with test priority since the use of spending effort for tests that would need to be refactored soon can be deemed low, but also as having a negative relation with test priority, as when one is going to refactor code, it can be good to have regression tests. He says that the environment (composition of team, quality of team members) could also influence how stability should be used.

During the ranking of the different test sortings, Participant 3 noted that there was one method that was placed highly in the rankings, but that he deemed not that important to test. This method scored well on all metrics (somewhat high complexity, linked to several functions with high user value and/or high stability), but is according to Participant 3 not that important and mostly dependent on libraries. He thinks that the code of this method should be easy to understand for a developer, and thereby the method should not be that fault-prone. Another thing he mentions is that the random selection of unit tests used for the sortings all are linked to few functions; he thinks this makes the comparison of rankings more difficult. He thinks that methods that are used in many places in the application, should be tested more. He also mentions that he finds it difficult to consider the entire list of a sorting, and that he focuses on the tests that the sortings say should be implemented first. He thinks these are the most important, since these are the tests that would actually be implemented first, and the ones that would not be implemented are deemed less important to him. He does note that he looks at whether the tests that are ranked at the bottom of the sorting actually should be there, but that he focuses less on the exact ordering of the tests in the lower regions. After ranking the functional test sortings, Participant 3 noted that he valued user value the highest and that he felt complexity and stability were subordinate. He wondered whether this feeling was justifiable and what led to better code.

When shown the sortings of UPSS(1,1,1,0.25), Participant 3 noted that there were things that he liked and that he didn't like about them. He felt that with both the unit and functional test sortings the most important tests correctly made it to the top, but that there were also some tests ranked highly, that he felt test code that is not complicated enough to warrant writing automated tests (if budget is limited). In such a case he would use for example code reviews to check the code quality. He tought the tests that were rated lowest by UPSS(1,1,1,0.25) were done so correctly.

Participant 3 thinks that being able to change the weighing factors per project is important, but that they should be kept constant over time. He would like to see the UX of TGUI being extended, for example by integrating with other tooling such as test coverage. He does think that it is a very good starting point.

E.2.3.2 Data

Experience questions (script items 2 to 4)

Question	Answer
On a scale of 1-10, how much experience do you have with the codebase of the TypeScript project?	8
On a scale of 1-10, how much experience do you have with the code of specific part of the application that the case study focusses on?	7
On a scale of 1-10, how much experience do you have with test automation?	9

Tests arranged without metadata (script item 8)

Unit tests			
Test	**Participant 3**	**UPSS(1,1,1,0.25)**	**UPSS(1,0.5,0,0.25)**
1	5	12	14
2	8	4	8
3	14	13	12
4	15	14	13
5	9	3	7
6	4	10	6
7	10	9	10
8	1	5	2
9	3	6	5
10	7	8	4
11	12	11	11
12	11	1	1
13	2	7	3
14	13	15	15
15	6	2	9

Participant 3 vs. UPSS(1,1,1,0.25)	Average absolute difference	3.73
	Spearman's rank correlation (r_s)	0.4429
	Spearman's rank correlation p-value	0.0982

Participant 3 vs. UPSS(1,0.5,0,0.25)	Average absolute difference	2.67
	Spearman's rank correlation (r_s)	0.5964
	Spearman's rank correlation p-value	0.0189

Functional tests			
Test	**Participant 3**	**UPSS(1,1,1,0.25)**	**UPSS(1,0.5,0,0.25)**
1	11	2	1
2	7	5	7
3	12	9	13
4	5	13	9
5	6	1	2
6	3	7	4
7	9	11	14
8	8	10	10
9	2	3	3
10	1	8	6
11	14	6	11
12	13	4	5
13	10	14	12
14	4	12	8

Participant 3 vs. UPSS(1,1,1,0.25)	Average absolute difference	5.14
	Spearman's rank correlation (r_s)	-0.0593
	Spearman's rank correlation p-value	0.7085

Participant 3 vs. UPSS(1,0.5,0,0.25)	Average absolute difference	2.57
	Spearman's rank correlation (r_s)	0.6000
	Spearman's rank correlation p-value	0.0233

Tests arranged with metadata (script item 11)

Unit tests			
Test number	**Participant 3**	**UPSS(1,1,1,0.25)**	**UPSS(1,0.5,0,0.25)**
1	15	13	12
2	5	15	15
3	10	9	14
4	6	6	7
5	7	2	1
6	2	4	6
7	14	3	4
8	9	14	13
9	1	1	2
10	4	5	5
11	3	8	8
12	13	12	11
13	12	7	3
14	8	10	9
15	11	11	10

Participant 3 vs. UPSS(1,1,1,0.25)	Average absolute difference	3.33
	Spearman's rank correlation (r_s)	0.4000
	Spearman's rank correlation p-value	0.1396

Participant 3 vs. UPSS(1,0.5,0,0.25)	Average absolute difference	4.13
	Spearman's rank correlation (r_s)	0.2714
	Spearman's rank correlation p-value	0.3278

Functional tests			
Test number	**Participant 3**	**UPSS(1,1,1,0.25)**	**UPSS(1,0.5,0,0.25)**
1	7	5	9
2	1	3	2
3	9	10	10
4	14	11	14
5	8	4	8
6	5	8	3
7	2	12	6
8	3	7	4
9	13	9	13
10	4	13	5
11	12	6	11
12	6	2	1
13	9	1	7
14	11	14	12

Participant 3 vs. UPSS(1,1,1,0.25)	Average absolute difference	4.5
	Spearman's rank correlation (r_s)	0.1452
	Spearman's rank correlation p-value	0.6204

Participant 3 vs. UPSS(1,0.5,0,0.25)	Average absolute difference	1.4
	Spearman's rank correlation (r_s)	0.8694
	Spearman's rank correlation p-value	0.0000

Test sorting rankings (script item 14)

Unit tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,1,1,0)	5	0
2.	UPSS(0,1,0,0.25)	4	1
3.	UPSS(1,1,0,0.25)	3	2
4.	UPSS(1,1,1,0.25)	2	3
5.	UPSS(1,0,0,0.25)	1	4
6.	Random	0	5

Functional tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	Random	4	0
2.	UPSS(1,0,0,0.25)	3	1
3.	UPSS(1,1,1,0.25)	2	2
4.	UPSS(1,1,0,0.25)	1	3
5.	UPSS(0,1,0,0.25)	0	4

E.2.4 Participant 4

E.2.4.1 Summary

Participant 4 gives himself a 9 (out of 10) for his experience level w.r.t. the codebase of the TypeScript project, a 7 for his experience level w.r.t. the specific part of the

TypeScript project that the user study focusses on, and a 5 for his experience level w.r.t. test automation.

Participant 4 notes that the suggested tests are the sort of tests that he would implement for the specific part of the TypeScript project. If he had all time and budget available, he would possibly implement all of them; otherwise, he would deem some as more useful than others. If time was limited, he would write larger tests that encompass multiple of the smaller tests.

For ranking the suggested tests without metadata, Participant 4 took the approach of first grouping tests that he deemed similar, and afterwards looking at if he would implement a test, whether that would also cover other test items. Then, he ranked tests that he thought would take little time to implement, and lastly came tests that he deemed totally non-useful. When asked what influenced his effort estimations, Participant 4 said that it came down mainly to gut feeling and also the amount of data that would be influenced with the test. This is possible for him since he knows the data and database structures of the TypeScript project.

While ranking the tests with metadata, Participant 4 noticed the high complexity of a certain functional test; on this he remarked that even though the complexity is high, he thinks writing this functional test would not cost more effort compared to other functional tests. With unit tests you would test the actual code, and therefore testing more complex code would cost more effort, he thinks. When he was done ranking and was asked what influenced his ranking, he said that he now looked more at what would be more important to the users, and afterwards at the stability values; if something doesn't change, he would deem it less important to write tests for it. Lastly, he sorted on complexity.

When comparing different unit test sortings, he notes that for two specific unit tests, he would rank the one with more functions and a lower complexity (and therefore lower testing effort) higher than the other. A certain unit test he would rank lowly since he knows from experience that it is not used much, even though its user value is high. Participant 4 also said that he looked at, among others, which sorting covered the higher variety of functions. He thinks that having a higher diversity of tests would lead to a higher test coverage. Methods that mostly are a conduit to libraries are not that useful to test according to him. Most functional test sortings are of similar quality to him. After looking at the functional test sorting of UPSS(1,1,1,0.25), Participant 4 notes that he would bring forward one specific test; he does not agree with the product owner assigned user value on that function. Other placements of tests he agrees with.

He thinks using UPSS would be an option. He would weigh user value and stability more heavily, and complexity less, since he would rather look at the stability value instead of complexity for seeing whether a function contains bugs.

E.2.4.2 Data

Experience questions (script items 2 to 4)

Question	Answer
On a scale of 1-10, how much experience do you have with the codebase of the TypeScript project?	9
On a scale of 1-10, how much experience do you have with the code of specific part of the application that the case study focusses on?	7
On a scale of 1-10, how much experience do you have with test automation?	5

Tests arranged without metadata (script item 8)

Unit tests			
Test number	**Participant 4**	**UPSS(1,1,1,0.25)**	**UPSS(2,0,-1,0.5)**
1	2	12	13
2	6	4	12
3	11	13	9
4	13	14	7
5	7	3	14
6	1	10	5
7	12	9	11
8	3	5	3
9	9	6	6
10	10	8	2
11	15	11	10
12	8	1	4
13	5	7	1
14	14	15	8
15	4	2	15

Participant 4 vs. UPSS(1,1,1,0.25)	Average absolute difference	3.60
	Spearman's rank correlation (r_s)	0.4536
	Spearman's rank correlation p-value	0.0895

Participant 4 vs. UPSS(2,0,-1,0.5)	Average absolute difference	5.20
	Spearman's rank correlation (r_s)	0.0179
	Spearman's rank correlation p-value	0.9496

Functional tests			
Test number	**Participant 4**	**UPSS(1,1,1,0.25)**	**UPSS(2,0,-1,0.5)**
1	14	2	3
2	3	5	8
3	13	9	13
4	12	13	4
5	8	1	7
6	2	7	6
7	9	11	14
8	6	10	10
9	5	3	1
10	1	8	2
11	4	6	12
12	10	4	9
13	11	14	11
14	7	12	5

Participant 4 vs. UPSS(1,1,1,0.25)	Average absolute difference	4.43
	Spearman's rank correlation (r_s)	0.1516
	Spearman's rank correlation p-value	0.680

Participant 4 vs. UPSS(2,0,-1,0.5)	Average absolute difference	1.33
	Spearman's rank correlation (r_s)	0.2220
	Spearman's rank correlation p-value	0.4456

Tests arranged with metadata (script item 11)

Unit tests			
Test number	**Participant 4**	**UPSS(1,1,1,0.25)**	**UPSS(2,0,-1,0.5)**
1	8	9	9
2	7	12	12
3	4	14	14
4	2	5	5
5	14	10	10
6	5	13	13
7	11	4	4
8	10	8	8
9	12	7	7
10	15	1	1
11	1	2	2
12	3	15	15
13	13	3	3
14	6	6	6
15	9	11	11

Participant 4 vs. UPSS(1,1,1,0.25)	Average absolute difference	5.60
	Spearman's rank correlation (r_s)	-0.3179
	Spearman's rank correlation p-value	0.2483

Participant 4 vs. UPSS(2,0,-1,0.5)	Average absolute difference	1.33
	Spearman's rank correlation (r_s)	0.8964
	Spearman's rank correlation p-value	0.0000

Functional tests			
Test number	**Participant 4**	**UPSS(1,1,1,0.25)**	**UPSS(2,0,-1,0.5)**
1	5	5	8
2	2	3	1
3	9	10	10
4	13	11	14
5	6	4	9
6	1	8	2
7	3	12	5
8	7	7	6
9	14	9	13
10	4	13	4
11	11	6	12
12	8	2	3
13	10	1	7
14	12	14	11

Participant 4 vs. UPSS(1,1,1,0.25)	Average absolute difference	4.14
	Spearman's rank correlation (r_s)	0.1385
	Spearman's rank correlation p-value	0.6369

Participant 4 vs. UPSS(2,0,-1,0.5)	Average absolute difference	1.71
	Spearman's rank correlation (r_s)	0.8593
	Spearman's rank correlation p-value	0.0000

Test sorting rankings (script item 14)

Unit tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,1,1,0.25)	4	1
1.	UPSS(1,1,0,0.25)	4	1
1.	UPSS(1,1,1,0)	4	1
4.	UPSS(0,1,0,0.25)	2	3
5.	UPSS(1,0,0,0.25)	1	4
6.	Random	0	5

Functional tests			
Rank	**Sorting**	**Upvotes**	**Downvotes**
1.	UPSS(1,0,0,0.25)	4	0
2.	UPSS(1,1,0,0.25)	3	1
3.	UPSS(1,1,1,0.25)	2	2
4.	UPSS(0,1,0,0.25)	1	3
5.	Random	0	4

Appendix F

User manuals for writing code annotations

F.1 Java

F.1.1 Dutch (original)

Handleiding annotaties (Java)

Mijn onderzoek gaat o.a. over het automatisch suggereren en prioriteren van tests op basis van een code base. Deze prioritering gebeurt op basis van zowel de geschatte foutgevoeligheid van de code, als door de gebruiker aangegeven gebruikswaarde van specifieke functionaliteit. Om het nut en de bruikbaarheid van deze aanpak te onderzoeken, ontwikkel ik een tool die op basis van een code base automatisch de aanpak kan toepassen. Om dit te kunnen doen is er van de methoden in de code base bepaalde informatie nodig.

Basis-informatie over methoden (zoals hun naam, lengte, en welke andere methoden ze gebruiken) wordt automatisch opgehaald. Echter, informatie zoals welke functionaliteiten er bestaan, wat hun gebruikswaardes zijn, en welke methoden er aan bijdragen, zit niet standaard in een code base. Om dit gat te vullen wil ik werken met annotaties.

De annotaties dienen bij “zo hoog mogelijke” methoden geïntroduceerd te worden. Met dit hoge niveau bedoel ik bijvoorbeeld endpoints van een webserver. Deze methoden noem ik vanaf hier “ingangspunten”. Op deze manier hoeft een annotatie slechts één keer geschreven te worden, en kan door de call stack te volgen ook voor “lagere” methoden bepaald worden aan welke functionaliteit(en) ze bijdragen.

Wat is een functionaliteit

Onder functionaliteit versta ik iets als “Aanmaken van een concepttrein”. Een dergelijke functionaliteit is een actie met een resultaat die door een eindgebruiker uitgevoerd kan worden en ook los getest zou kunnen worden. Zie verderop in dit document voor voorbeelden. Vaak zal één functionaliteit ook maar één ingangspunt gebruiken, echter is het mogelijk en toegestaan dat één functionaliteit meerdere ingangspunten gebruikt (de relatie tussen ingangspunten en functionaliteiten is “many-to-many”). Als bijvoorbeeld bij de functionaliteit “Inloggen van een gebruiker” eerst “/authenticateUser” en vervolgens “/authorizeUser” worden aangeroepen zonder expliciete tussenkomst van een gebruiker (de client-applicatie voert deze stappen automatisch uit), dan dragen deze beide endpoints bij aan deze ene functionaliteit. Uitleg over hoe dit te noteren volgt later in dit document.

Hoe de annotaties te schrijven

De annotaties worden bij een Java code base geschreven in Javadoc. Op deze manier zouden de annotaties eenvoudig en “mooi” geïntegreerd kunnen worden in bestaande documentatie. Tevens heeft dit als voordeel dat bestaande tooling gebruikt kan worden voor de extractie van de annotaties.

Een functionaliteit heeft een aantal eigenschappen:

Naam	**Beschrijving**
Beschrijving	Een korte beschrijving van de functionaliteit.
Gebruikswaarde	Een getal van 1 t/m 5 dat de gebruikswaarde van de functionaliteit beschrijft. Een hogere waarde betekent een hogere gebruikswaarde, oftewel functionaliteit waarvan het belangrijker is dat de implementatie **correct** werkt.
Stabiliteit	Een getal van 1 t/m 5 dat aangeeft hoe “stabiel” een functionaliteit en haar implementatie is. Een stabiliteitswaarde van 5 geeft aan dat de functionaliteit en haar implementatie volledig stabiel zijn en niet meer gaan veranderen. Een stabiliteitswaarde van 1 geeft aan dat de functionaliteit en/of haar implementatie zeer binnenkort gaat wijzigen; het heeft dan weinig zin om *nu* tests voor deze functionaliteit/implementatie te schrijven.
Identifier	*Bij het geval waar een functionaliteit op een hoger niveau dan het toegangspunt gedefinieerd wordt, dient de functionaliteit een identifier gegeven te worden, waar de toegangspunten later naar kunnen verwijzen. Hierover later meer.*

Zoals vermeld kan een functionaliteit zowel één ingangspunt gebruiken als meerdere. Voor elk van deze gevallen is een aparte manier van notatie.

Één functionaliteit, één ingangspunt

Bij het geval waar een functionaliteit maar één ingangspunt beslaat, wordt de functionaliteit gedefinieerd bij het ingangspunt zelf.

In het onderstaande voorbeeld wordt de functionaliteit “List author book titles” gedefinieerd met een gebruikswaarde van 3 (gemiddeld) en een stabiliteitswaarde van 1 (wordt zeer binnenkort aangepast).

```
    /**
     * Get the titles of all books written by a specific author.
     *
     * @param authorId The ID of the author
     * @return A list of book titles
     * @functionality "List author book titles" 3 1
     */
    @MessageMapping("/getBookTitles")
    public List<String> getBookTitles(int authorId) {
        List<Book> books = this.booksService.getAuthorBookTitles(authorId);

        return OutputFormatter.formatBooks(books);
    }
```

De `functionality` tag geeft aan dat er een nieuwe functionaliteit gedefinieerd wordt. Daarna volgen de beschrijving van de functionaliteit (tussen aanhalingstekens), de gebruikswaarde, en de stabiliteit.

Door de call stack te volgen weet de tool nu dat deze functionaliteit bij zowel de `getBookTitles` methode hoort, als bij de `getAuthorBookTitles` op de `BooksService` en de `formatBooks` methode op de `OutputFormatter` class hoort. Mochten deze methoden nog weer andere methoden aanroepen, worden deze uiteraard ook gezien als bijdragend aan de functionaliteit.

Één functionaliteit, meerdere ingangspunten (en één ingangspunt, meerdere functionaliteiten)

Bij het geval waar een functionaliteit meerdere ingangspunten gebruikt, wordt de functionaliteit op een hoger niveau (in het geval van Java op class-niveau) gedefinieerd. De gerelateerde ingangspunten verwijzen vervolgens naar deze definitie via een identifier.

In het onderstaande voorbeeld wordt de functionaliteit “Login a user” gedefinieerd met een gebruikswaarde van 5 (zeer hoog) en een stabiliteitswaarde van 1 (wordt mogelijk ooit nog aangepast), en wordt deze functionaliteit gekoppeld aan twee methoden.

```
/**
 * @functionalityDef loginUser "Login a user" 5 3
 */
public class UsersController {
    /**
     * Authenticate a user.
     *
     * @param userEmail The user's email
     * @param password  The provided password
     * @return The generated access token
     * @functionality loginUser
     */
    @MessageMapping("/authenticate")
    public String authenticateUser(String userEmail, String password) {
        return this.usersService.authenticate(userEmail, password);
    }

    /**
     * Authorize a user.
     *
     * @param token The user's access token
     * @return A list of scopes the user has access to
     * @functionality loginUser
     */
    @MessageMapping("/authorize")
    public List<Scope> authorizeUser(String token) {
        return this.usersService.authorize(token);
    }
}
```

De `functionalityDef` wordt gebruikt om een functionaliteit te definiëren, zonder deze gelijk aan een methode te koppelen. Na de `functionalityDef` volgen een identifier (`loginUser`) voor de functionaliteit, de beschrijving van de functionaliteit, de gebruikswaarde, en de stabiliteit.

Bij de methoden wordt met de `functionality` tag de methode gekoppeld aan de eerder gedefinieerde functionaliteit. In het eerdere voorbeeld werd deze tag gebruikt om een nieuwe functionaliteit te definiëren en direct te koppelen aan de bijbehorende methode; wanneer er alleen een identifier als waarde bij de tag wordt gegeven, wordt deze gezien als verwijzing en wordt de bijbehorende methode gekoppeld aan de naar verwezen functionaliteit.

Het afleiden van onderliggende methoden gebeurt uiteraard hetzelfde als eerder genoemd.

F.2 Typescript

F.2.1 Dutch (original)

Handleiding annotaties (TypeScript)

Mijn onderzoek gaat o.a. over het automatisch suggereren en prioriteren van tests op basis van een code base. Deze prioritering gebeurt op basis van zowel de geschatte foutgevoeligheid van de code, als door de gebruiker aangegeven gebruikswaarde van specifieke functionaliteit. Om het nut en de bruikbaarheid van deze aanpak te onderzoeken, ontwikkel ik een tool die op basis van een code base automatisch de aanpak kan toepassen. Om dit te kunnen doen is er van de methoden in de code base bepaalde informatie nodig.

Basis-informatie over methoden (zoals hun naam, lengte, en welke andere methoden ze gebruiken) wordt automatisch opgehaald. Echter, informatie zoals welke functionaliteiten er bestaan, wat hun gebruikswaardes zijn, en welke methoden er aan bijdragen, zit niet standaard in een code base. Om dit gat te vullen wil ik werken met annotaties.

De annotaties dienen bij “zo hoog mogelijke” methoden geïntroduceerd te worden. Met dit hoge niveau bedoel ik bijvoorbeeld endpoints van een webserver. Deze methoden noem ik vanaf hier “ingangspunten”. Op deze manier hoeft een annotatie slechts één keer geschreven te worden, en kan door de call stack te volgen ook voor “lagere” methoden bepaald worden aan welke functionaliteit(en) ze bijdragen.

Wat is een functionaliteit

Onder functionaliteit versta ik iets als “Aanmaken van een concepttrein”. Een dergelijke functionaliteit is een actie met een resultaat die door een eindgebruiker uitgevoerd kan worden en ook los getest zou kunnen worden. Zie verderop in dit document voor voorbeelden. Vaak zal één functionaliteit ook maar één ingangspunt gebruiken, echter is het mogelijk en toegestaan dat één functionaliteit meerdere ingangspunten gebruikt (de relatie tussen ingangspunten en functionaliteiten is “many-to-many”). Als bijvoorbeeld bij de functionaliteit “Inloggen van een gebruiker” eerst “/authenticateUser” en vervolgens “/authorizeUser” worden aangeroepen zonder expliciete tussenkomst van een gebruiker (de client-applicatie voert deze stappen automatisch uit), dan dragen deze beide endpoints bij aan deze ene functionaliteit. Uitleg over hoe dit te noteren volgt later in dit document.

Hoe de annotaties te schrijven

De annotaties worden bij een TypeScript code base geschreven in Jsdoc. Op deze manier zouden de annotaties eenvoudig en “mooi” geïntegreerd kunnen worden in bestaande documentatie. Tevens heeft dit als voordeel dat bestaande tooling gebruikt kan worden voor de extractie van de annotaties.

Een functionaliteit heeft een aantal eigenschappen:

Naam	**Beschrijving**
Beschrijving	Een korte beschrijving van de functionaliteit.
Gebruikswaarde	Een getal van 1 t/m 5 dat de gebruikswaarde van de functionaliteit beschrijft. Een hogere waarde betekent een hogere gebruikswaarde, oftewel functionaliteit waarvan het belangrijker is dat de implementatie **correct** werkt.
Stabiliteit	Een getal van 1 t/m 5 dat aangeeft hoe “stabiel” een functionaliteit en haar implementatie is. Een stabiliteitswaarde van 5 geeft aan dat de functionaliteit en haar implementatie volledig stabiel zijn en niet meer gaan veranderen. Een stabiliteitswaarde van 1 geeft aan dat de functionaliteit en/of haar implementatie zeer binnenkort gaat wijzigen; het heeft dan weinig zin om *nu* tests voor deze functionaliteit/implementatie te schrijven.
Identifier	*Bij het geval waar een functionaliteit op een hoger niveau dan het toegangspunt gedefinieerd wordt, dient de functionaliteit een identifier gegeven te worden, waar de toegangspunten later naar kunnen verwijzen. Hierover later meer.*

Zoals vermeld kan een functionaliteit zowel één ingangspunt gebruiken als meerdere. Voor elk van deze gevallen is een aparte manier van notatie.

Één functionaliteit, één ingangspunt

Bij het geval waar een functionaliteit maar één ingangspunt beslaat, wordt de functionaliteit gedefinieerd bij het ingangspunt zelf.

In het onderstaande voorbeeld wordt de functionaliteit “List author book titles” gedefinieerd met een gebruikswaarde van 3 (gemiddeld) en een stabiliteitswaarde van 1 (wordt zeer binnenkort aangepast).

```
/**
 * Get the titles of all books written by a specific author.
 *
 * @param authorId The ID of the author
 * @return A list of book titles
 * @functionality "List author book titles" 3 1
 */
@Get('/bookTitles')
public getBookTitles(
  @Query('authorId')
  authorId: number
): string[] {
  const books = this.booksService.getAuthorBookTitles(authorId);

  return OutputFormatter.formatBooks(books);
}
```

De `functionality` tag geeft aan dat er een nieuwe functionaliteit gedefinieerd wordt. Daarna volgen de beschrijving van de functionaliteit (tussen aanhalingstekens), de gebruikswaarde, en de stabiliteit.

Door de call stack te volgen weet de tool nu dat deze functionaliteit bij zowel de `getBookTitles` methode hoort, als bij de `getAuthorBookTitles` op de `BooksService` en de `formatBooks` methode op de `OutputFormatter` class hoort. Mochten deze methoden nog weer andere methoden aanroepen, worden deze uiteraard ook gezien als bijdragend aan de functionaliteit.

Één functionaliteit, meerdere ingangspunten (en één ingangspunt, meerdere functionaliteiten)

Bij het geval waar een functionaliteit meerdere ingangspunten gebruikt, wordt de functionaliteit op een hoger niveau (in het geval van Java op class-niveau) gedefinieerd. De gerelateerde ingangspunten verwijzen vervolgens naar deze definitie via een identifier.

In het onderstaande voorbeeld wordt de functionaliteit “Login a user” gedefinieerd met een gebruikswaarde van 5 (zeer hoog) en een stabiliteitswaarde van 1 (wordt mogelijk ooit nog aangepast), en wordt deze functionaliteit gekoppeld aan twee methoden.

```
/**
 * @functionalityDef loginUser "Login a user" 5 3
 */
export class UsersController {
  /**
   * Authenticate a user.
   *
   * @param userEmail The user's email
   * @param password  The provided password
   * @return The generated access token
   * @functionality loginUser
   */
  @Post('/authenticate')
  public authenticateUser(userEmail: string, password: string): string {
    return this.usersService.authenticate(userEmail, password);
  }

  /**
   * Authorize a user.
   *
   * @param token The user's access token
   * @return A list of scopes the user has access to
   * @functionality loginUser
   */
  @Post('/authorize')
  public authorizeUser(token: string): Scope[] {
    return this.usersService.authorize(token);
  }
}
```

De `functionalityDef` wordt gebruikt om een functionaliteit te definiëren, zonder deze gelijk aan een methode te koppelen. Na de `functionalityDef` volgen een identifier (`loginUser`) voor de functionaliteit, de beschrijving van de functionaliteit, de gebruikswaarde, en de stabiliteit.

Bij de methoden wordt met de `functionality` tag de methode gekoppeld aan de eerder gedefinieerde functionaliteit. In het eerdere voorbeeld werd deze tag gebruikt om een nieuwe functionaliteit te definiëren en direct te koppelen aan de bijbehorende methode; wanneer er alleen een identifier als waarde bij de tag wordt gegeven, wordt deze gezien als verwijzing en wordt de bijbehorende methode gekoppeld aan de naar verwezen functionaliteit.

Het afleiden van onderliggende methoden gebeurt uiteraard hetzelfde als eerder genoemd.