

Toward Exploratory Understanding of Software using Test Suites

Dominik Meier
dominik.meier@student.hpi.uni-
potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Toni Mattis
toni.mattis@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@hpi.uni-
potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

ABSTRACT

Changing software without correctly understanding it often leads to confusion, as developers do not understand how the change corresponds to the new observed behaviour of the system. Today, many software systems are equipped with a test suite. Test suites document code and give feedback on changed program behaviour. We explored ways to use test suites for software comprehension and implemented a tool that provides additional visualisation and gives immediate feedback on software changes. Information about changes in the software and their implications to the test suite are collected using mutation testing. The tool uses this information to present relevant test cases for developers, and additionally prioritise test executions for immediate feedback. Our research indicates that entropy metrics can find test cases that are relevant for a specific context in the source code. Additionally, simple test case prioritisation strategies can already lead to a significant decrease in feedback time. Based on our case study we argue that test suites are not only useful for regression testing but can be used to generate meaningful information for software comprehension activities.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments.**

KEYWORDS

program comprehension, mutation testing, immediate feedback, test prioritisation

ACM Reference Format:

Dominik Meier, Toni Mattis, and Robert Hirschfeld. 2021. Toward Exploratory Understanding of Software using Test Suites. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming* (<Programming> '21 Companion), March 22–26, 2021, Virtual, UK. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3464432.3464438>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
<Programming> '21 Companion, March 22–26, 2021, Virtual, UK
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8986-0/21/03...\$15.00
<https://doi.org/10.1145/3464432.3464438>

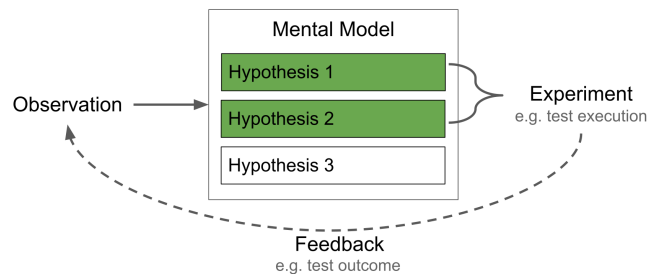


Figure 1: Iterative two phase process of generating a mental model of software systems

1 INTRODUCTION

To effectively work with a software system, developers need to create a mental model of the software system. The workflow of developers creating such a model can be conceptualized in two phases shown in Figure 1: observation and experimentation. Observation is building hypotheses about the system. This is done mainly by reading lines of code. Hypotheses can also be generated out of experience from the programmer, general assumptions, or by watching how the system behaves. The granularity of the hypotheses varies with the developers' context. The context describes what part of the software they currently try to understand or change and in what detail. They do not need to understand the whole system in detail if only specific parts are relevant.

Experimentation is testing whether a subset of the build hypotheses correctly predicts the system's behaviour. This is often done by changing source code and subsequently checking if the system behaves in the predicted way. Other ways are debugging the program to check if the variables take the predicted state or checking if tests fail.

1.1 The Role of Tests

Writing tests is a well-known best practice in software engineering [4]. However, with current standard test runners, they are primarily used at the end of the development cycle to prevent regression, simplify refactoring, or document code.

Tests can be viewed as hard-coded experiments. Especially unit tests efficiently check if a system behaves in a certain way. One could argue that they therefore could be used in the formulation and checking of hypotheses in the mental model of developers.

Currently, however, they are run late in the process after several edits have already taken place to check if the system is still working correctly because standard test runner systems have long feedback times, as they run all tests without test case prioritisation.

Additionally, test failures are not set into the context, making it hard for developers to draw correct conclusions from the set of test outcomes.

Due to the long feedback time, the test cases will only reveal the errors after they have been made, as developers relying on a model based on wrong or incomplete assumptions will generate errors as they can not predict the consequences of their actions correctly. The late feedback from the executed tests additionally obstructs finding the correct mental model as there is no direct mental connection between changes in the code and testing errors.

In the following, we explore possibilities to help developers in both phases of program comprehension using test suites. During observation, they should be aware of existing experiments (i.e. tests) related to their current context. This can help to formulate better hypotheses about the system. During experimentation, they should receive immediate feedback on the relevant experiments to be able to mentally connect changes to the consequences for the system. This enables better and faster feedback on whether underlying hypotheses are correct.

We designed a prototype that uses previously analysed data about the software using the technique of mutation testing. Entropy metrics are used to rank the possible gain of information on tests for the current context of developers. To give immediate feedback, we used machine learning algorithms on the mutation data to predict which tests will fail with high probability. These can then be executed with higher priority, gaining faster feedback for developers. We tested different techniques for visualising the generated feedback to increase the benefit for developers.

The paper is structured into seven sections. To better understand the general context, Section 2 gives an overview of the background and related work. Section 3 presents the design space of our approach and the proposed workflow using our prototype. Section 4 explains the technical implementation of the prototype. In Section 5, we discuss findings for testing different relevance metrics, our measured feedback times, and different visualisation techniques for our use case. Section 6 introduces ways to further evaluate the approach and to improve the data generation and interaction techniques of the prototypes.

2 BACKGROUND AND RELATED WORK

2.1 Exploratory and Live Systems

Different models of program comprehension were outlined in studies by Mayrhauser *et al.* [6]. They all include that programmers combine long term knowledge with external information, for example, code, in order to create a mental model. One special type of external information for program comprehension is fast feedback to developers actions.

Giving developers the feeling of changing a running program has a long research history. Rein *et al.* [7] found different motivations for liveness, including program comprehension and exploration of software systems. Our prototype especially addresses these motivations by giving developers additional visual information. Tanimoto [10] describes four different levels of liveness. A scenario where developers execute the test suite manually can be classified as a level two live system, since the test suite is executable, but the

execution is not automatically triggered. This tends to create long feedback cycles leading to late error recognition.

Therefore, we propose to use a level three live system that gives feedback at trigger points. Developers are not always ready for feedback, e.g., while typing in the name of a variable, but only at specific task boundaries. We try to approximate these boundaries by linking our feedback to certain editor operations, like CTRL-S. Some test runner systems already support the automated triggering on source code changes. While this decreases the feedback time for developers as tests get executed more frequently, it is hard for developers to utilise during program comprehension as the feedback is not specifically adapted to their current context. Therefore, it is hard to gain helpful information from a set of test failures and successes.

Because our prototype acts on certain trigger points, it is important to understand when it should help developers with generated feedback. The feedback should not interfere with the ongoing thought process too strongly.

2.2 Feedback Timing

Dabrowski and Munson [1] distinguish between control tasks and conversational tasks. The classification of the task decides on how the feedback time should be chosen. We see the observation phase as a control task. Developers want to explore the system without interaction. The delay here should be as small as possible. We do not want the additional visualisation, in this case, to make developers aware of relevant tests, to stop the observation phase from being received as a control task.

The experimentation phase is a more conversational task. Developers interact with the system, wait for a response, and act accordingly. Conversational tasks can benefit from delays, as they keep the conversation between human and computer fluent. We trigger the test feedback on the saving of the current state of the program. This is an approximation of a finished sub-task for developers. We assume that at this moment, developers are ready for feedback as they wrote out changes. Because we assume a sub-task boundary, there should be no further delays, and the feedback should come immediately. Seow [9] categorizes immediate feedback time between 0.5s and 1s.

2.3 Test Case Prioritisation

The short time frame for immediate feedback does not allow to run the whole test suite of most programs. Test case prioritisation techniques try to reorder the schedule of test executions in order to find errors faster.

We want to measure how much information gain can happen immediately. Our optimization problem is therefore to produce a set of tests that give the most feedback in under 1s. The order of the tests after that time span does not matter much, as they can not be used for immediate feedback. A survey by Lou *et al.* [4] lists different metrics to compare different prioritization strategies. However, the traditional prioritisation measurements integrate over all test cases, most prominently the Average Percentage of Faults Detected (APFD) and variations of it. Additionally, in our scenario of live feedback the execution time of the model starts to become

relevant; if the model already uses a big fraction of a second, its results need to be better in order to justify the long execution time.

Lightweight, change-based prioritisation strategies have previously been researched by Mattis *et al.* [5], which can be efficiently calculated for changes. The performance of the models was unfortunately only tested with the APFD metric that does not focus on immediacy.

To evaluate our results, we measured when the first failure in the test suite is detected and when the last failure is detected.

2.4 Mutation Testing

Mutation testing has become more and more popular as a source to generate source code changes leading to faults similar to real-world faults [2]. The standard procedure creates mutants by changing a small part of the source code of the program. The test suite is now executed for the mutant and checked if it detects the change. A study by Just *et al.* [3] found a significant correlation between mutant detection and real-world fault detection. This suggests that mutation testing data is also useful in analysing the relevance of tests to specific parts of the code or predicting their failures in response to changes close to mutation sites. Static analysis is not needed to generate mutation testing data, so it is applicable also for dynamic languages.

3 APPROACH

Our proposed workflow gives developers feedback in both phases, observation and exploration. During observation, their current context is analysed and based on it, the relevance of each test is scored for the current context. Developers see which tests give good hints on the functionality of the current context and help to formulate good hypotheses. During experimentation, we give immediate feedback by executing tests in a prioritised way. By saving source code, developers trigger the action to obtain feedback about the current status of the system.

3.1 Design Space

As shown in Table 1, we differentiate between two types of feedback in our design space.

- (1) The *relevance of the test* ranking the importance of the test for the current context
- (2) The *test outcome* describes whether a test has failed or succeeded.

The design space can be subdivided into two layers.

- (1) The data query describing how to generate the relevance and immediate feedback data.
- (2) The type of visualisation showing the data.

3.2 Data Queries

To create the information needed for the visualisation, the dataset gets queried differently dependent on the program comprehension phase, i.e., observation and experimentation.

Observation. To calculate the test relevance, the context is approximated by the currently visible source code in the editor. We experimented with two different levels of granularity, file-based and line-based.

We used the mutation-coverage combined with the context information as our input data to the relevance rating. For a test t , and a context d , which can be a source code file or a subset of a source code file, we define $f_{t,d}$ as the number of failures from the context d . From all failures of the test that were collected, we count only the failures from mutants that changed something inside d . Our first relevance ranking was $f_{t,d}$, with d set to the approximation of the context of developers, i.e., what they have currently opened in the editor.

However, this approach ranks tests that fail very often overall higher than tests that are specific for the current context but fail less frequently on average. We, therefore, created an additional ranking following the **Term Frequency-inverse Document Frequency (TFIDF)** approach. TFIDF is widely used in information retrieval [8]. Usually, it is used to rank the importance of a term in a set of text documents. Some terms in text documents have overall higher average occurrence frequencies than others. This is similar to our scenario where some tests have higher average failing frequencies than others. TFIDF applies a weighting scheme that allows approximating the relevance more precisely.

The **Term Frequency (TF)** of a test t , and a source code file d is the logarithm of $1 + f_{t,d}$, so for $f_{t,d}$, only a logarithmic scaling is applied.

$$\text{TF}(t, d) = \log(1 + f_{t,d}) \quad (1)$$

The **Inverse Document Frequency (IDF)** is generated to filter out tests that often fail. While D is the corpus of all source code files, the number of source code files is divided by the number of times the test failed on all mutations.

$$\text{IDF}(t, D) = \log\left(1 + \frac{|D|}{|\{d \in D | t \in d\}|}\right) \quad (2)$$

Multiplying both results in the overall relevance score for a given test t , document d , and all source code files D .

$$\text{TFIDF}(t, d, D) = \text{TF}(t, d) \cdot \text{IDF}(t, D) \quad (3)$$





Experimentation. To give faster feedback during the experimentation phase, we use lightweight test case prioritisation techniques. The changes that are made during the mutation testing are classified by different features like the changed line of code and the filename.

The features for each mutant are used as training data for the test failure prediction algorithm. A given set of changes is split down into single lines of code changes. The same features are generated for these changes, and it is predicted if tests are likely to fail given these. All tests that are predicted to fail are reordered to the beginning of the test run. We analysed the performance of a decision tree algorithm, as well as one more advanced machine learning model, a random forest.

3.3 Different Visualisation Techniques

The results need to be visualised for both the relevance and the failures. The visualisation should provide developers with clear feedback, but if developers are currently not ready for feedback, it should be unobtrusive. Additionally, feedback types are not independent of each other. If a test is presented as irrelevant in the

Table 1: Design Space of the proposed Workflow

	Observation What tests give most information?	Experimentation Which failures give good feedback?
Data Query	How to approximate the context? • file-based • line-based	Which tests will fail? • Decision Tree • Random Forest
	How to rate relevance? • Mutant Coverage • TFIDF	
Visualisation	How to display relevance? • table  • scatter  • embedding 	How to display failure? • color coding 

current context, developers will have little interest in whether it is failing.

To allow developers to switch back and forth between feedback and the source code, we created a panel visualisation that takes around one-fifth of the screen. This is similar to other panels in code editors that provide the developer with additional information like software versioning information or debugging panels. In this panel, we experimented with different visualisation and layout techniques.

Colour Coding. All visualisations had in common that colour coding was used to denote test failures (red) from test successes (green). We integrated another colour that shows that a test is predicted to fail but not yet executed (orange). Additionally, to the hue of the colour that differentiates successes from failures, we included the option to desaturate the colour of not executed tests over time. This highlights the recent test results visually more important than older results. Additionally, developers get an indication of how old the last feedback they received got.

Table. As the most basic representation, we implemented a table of test failures that are sorted by relevance and contains the test’s name.

Scatter. In the scatter visualisation, a circle is drawn for each individual test case. The position of the circles on the y-Axis was set by the relevance given the current context and the x-Axis by the overall failure frequency of the individual test.

Embedding. For the embedding visualisation, we used the covariances of the test cases as a distance metric and calculated a t-distributed stochastic neighbour embedding (t-SNE) used for dimensionality reduction [11]. Because the calculated embedding position contains pairs of points with very close distance, we post-processed the positions using a force-based layout to guarantee a minimum distance between each point pair. For each test, a directional force towards the original embedding position was generated. Then, collision forces were introduced to prevent circles from overlapping.

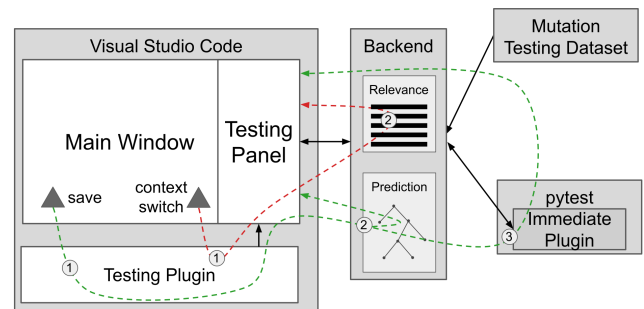


Figure 2: Architecture of the plugin

4 IMPLEMENTATION

We designed a functional prototype to enable exploratory feedback using testing suites as a Visual Studio Code Plugin with a micro-service backend architecture.¹ In the following sections we map out the general functionality, and show what happens internally if developers interact with the prototype. An architectural overview can be seen in Figure 2. Communication between the services happens via the Socket.IO protocol.² When we talk about sending data we mean communication via this protocol. The micro-service architecture does not only allow separation of concerns but also opens up the possibility to improve the performance of test executions by moving the service to a server.

4.1 Mutation Testing Dataset

To enable live feedback, we collect static data about the software suite using mutation testing. We chose to use the python micro-framework flask³ as an example of a project that is popular and well tested. We used a self implemented tool called Mutester⁴ which instruments pytest, the standard python testrunner, and mutmut,⁵ a python mutation testing tool.

¹The source code of the prototype is available as a public repository at <https://github.com/XPerianer/ImmediateTestFeedback>

²<https://github.com/socketio/socket.io-protocol>

³<https://github.com/pallets/flask>

⁴<https://github.com/XPerianer/Mutester>

⁵<https://github.com/boxed/mutmut>

It applies a mutation via mutmut, executes all tests, and records test failures and additional features of the current mutation in a dataframe.

The additional features are used as training data for the test failure prediction algorithm. They include features like file name and line number of current mutation change. The trained model on the dataset, in the simplest case a decision tree, is made available for the *Backend*.⁶

4.2 Architecture

In the following, we describe the architecture of the plugin, as seen in Figure 2 and show the information flow through the system that can be triggered by developers. Developers start the prototype by opening up a Visual Studio Code instance that has the *Testing Panel* loaded. The *Testing Plugin* provides it with the necessary code to display the content. The visualisation inside the panel is implemented as a JavaScript website using the D3.js⁷ framework.

The plugin integrates different editor hooks available that can track developers actions, like opening other editor windows or saving. This information is sent to the *Testing Panel* and forwarded to the *Backend*. The *Backend* coordinates the actions that are required on editor trigger points. On startup, it loads the relevance and prediction models from the *Mutation Testing Dataset*. This allows the following queries to be able to profit from cached data.

Observation. If developers save changes (see Figure 2: context switch), for example, by opening up another file, the *Testing Plugin* recognises the change (1). The context switch information is forwarded to the *Backend* via the *Testing Panel*, which queries the relevance information for the given context (2). The relevances are sent back to the *Testing Panel*, which then updates the visualisation.

Experimentation. If developers save the source file (see Figure 2: save), for example, via the CTRL-S shortcut, this will also be recognised by the *Testing Plugin* (1) and forwarded to the *Backend*. The same features collected previously for the mutations are now generated again for changes that are currently present in the source code. For each of these changed lines, the prediction model is run, and the failures are collected. This gives a list of tests that are likely to fail (2). Now the information flow is split into direct feedback to the *Testing Panel* and test reordering information for pytest. The *Testing Panel* can use the information list of likely failed tests to already give visual hints on what tests might fail. Also, the list is integrated into the call to pytest. The *Immediate Plugin* reorders the tests after the given list of tests and then waits for pytest to execute them (3). If a test fails, the *Immediate Plugin* is directly hooked after the failure and sends the information to the *Backend*, which forwards it to the *Testing Panel*.

5 DISCUSSION

5.1 Relevance Metrics

Figure 3 plots the different relevance metrics for the context based on file granularity. The y-Axis shows the relevance for the given context; the colour denotes the overall failure frequency of the

⁶Internally, this is done by serialisation and deserialisation of the decision model using the python library joblib <https://github.com/joblib/joblib>

⁷<https://d3js.org/>

test. For the file *json/tag.py* the annotated example shows that *test_appcontext_tearing_down_signal* is ranked highest when using *f_{t,d}*. While this test often fails generally, it has no particular connection to the JSON module in flask. However, *test_duplicate_tag* is very relevant, as this tests serialization function inside the JSON module. This is only ranked high when using **TFIDF**.

In general **TFIDF** ranks tests that have lower overall failures higher, which is an indication that the ranking is more specific to the context, and therefore better applicable to the given use case.

5.2 Feedback Times

During observation, the measured feedback loop time from opening a file and the beginning of the visual feedback is around 200ms, which feels instantaneous.

The immediate test feedback is more time-critical as the different processing steps add up. The three main steps are analysing the changes, executing the prediction model, and executing the tests. More changes lead to a higher execution time, as every line is analysed and predicted for. We measured the flask repository with 20 lines of code removed from different files and measured averages over ten runs. The setup code that analyses the changes in the repository and enters them in a data frame takes on average 119ms to run. The decision tree needs less than 3ms for its prediction, while the random forest takes an average of 117ms. With additional optimisation, this could certainly become faster.

For our measurements, we split test execution time in general setup time and test execution time. Using our current setup, which uses the pytest plugin architecture to report test failures, we have quite a high general setup time for pytest of 886ms on average till the first test report is logged into the *Backend*. Unfortunately, pytest is currently not optimised for immediate feedback.

Measuring the performance of the prioritisation algorithms for only ten test cases is not meaningful as the performance relies not only on the number of changes but also on their content. To analyse the model performance, we, therefore, split our mutation testing dataset into a train and test set. The models were trained on the training set and then evaluated on the test set. We summed up the previously recorded test execution times until the first or last error detection, given the scheduling induced by the predicted test failures of the model.

The native execution following the order of discovery in the test suite takes 1.2s on average till the first failure is detected. Even a simple decision tree algorithm decreases the time to the first test failure to about 300ms, as seen in Figure 4a. With the additional general test setup time, this test failure report will be very close to the 1s maximum duration of immediate feedback. More advanced models, like the random forest, can then further decrease the feedback time. Our prototype currently only implements two active features for the changes: the filename and the line number of the change. With additional features, the performance of the machine learning algorithm might further be increased.

Figure 4b compares the duration of the different processing steps added up. Even though the native execution order can skip the first two processing steps, the decision tree model outperforms it for the first failure found and the last failure found. The random forest is again better in both measurements. It is also visible that the general

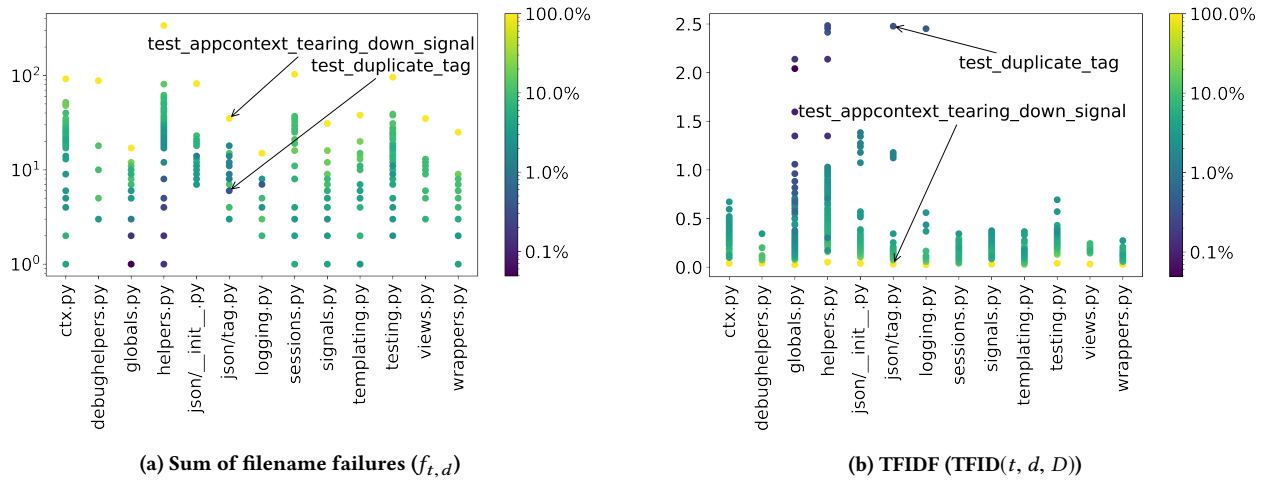


Figure 3: Test case relevance ranking of different metrics. The colour indicates the general test failure frequency

test setup step makes up a big percentage of the feedback time. It is more important to reduce the test runner setup time than to find a better machine learning model to give developers a faster feedback time. If the setup time could be reduced to half, which would still be over 400ms, the decision tree would have an average feedback time for first failure of under 1s.

5.3 Different Visualisation Techniques

We observed several differences how different types of visualisations could support developers.

The table representation (Figure 5a) allows the most efficient presentation of textual information and is also very space-efficient. Additionally, it is a common representation of data that does not need much additional explanation.

However, not all test cases can be represented at the same time, and when switching the context, it is hard to see which tests got more relevant in the new context.

To give developers the possibility to compare changes in the context to changes in the relevance, we created the animated scatter plot (Figure 5b). It has the advantage that besides the relevance of the test on the y-Axis, a different variable can be displayed on the x-Axis. The idea to plot the overall failure rate of the tests on the x-Axis is based on the assumption that developers would be able to distinguish between very specific tests that would have a lower overall failure score and less specific tests, e.g., integration tests. This should enable developers to choose between focusing on test cases that are more general if they need to get an overview of the system and concrete information about one method. The scatter visualisation thus enables developers to clearly track the changes in relevance on context switches and see the currently relevant test cases. However, irrelevant test cases strongly overlap each other, making them visually indistinguishable.

Trying to emphasise the connections between the test cases, the embedding visualisation focuses on the relevance. We switched the mapping of the relevance from a positional mapping to the size of

the drawn circles. This gives the best indication of how the current context is positioned in the overall software system. While smaller sub-modules like the `JSON` module only show certain small parts to be relevant (Figure 5c), more general files like the software entry point `app.py` show that they are important for almost all test cases (Figure 5d). We think that this visualisation intuitively gives us some feeling of the impact of the current context.

6 FUTURE WORK

Different options exist to evaluate and extend the approach. They can be categorized in evaluation, improvements in the underlying data model, and improvements for the developers experience. In the following, each of them is discussed in more detail.

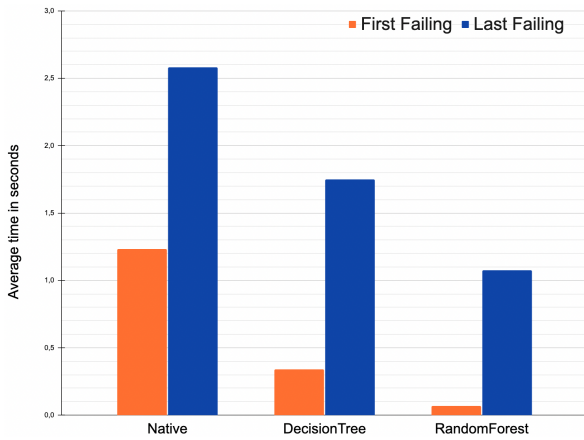
6.1 Evaluation

User Study. We did measure the feedback times of our system but did not measure how well the reaction times and ranked tests help developers better understand software systems. This would give evidence in how much this new type of software tool can improve developers speed and confidence in understanding and changing software systems.

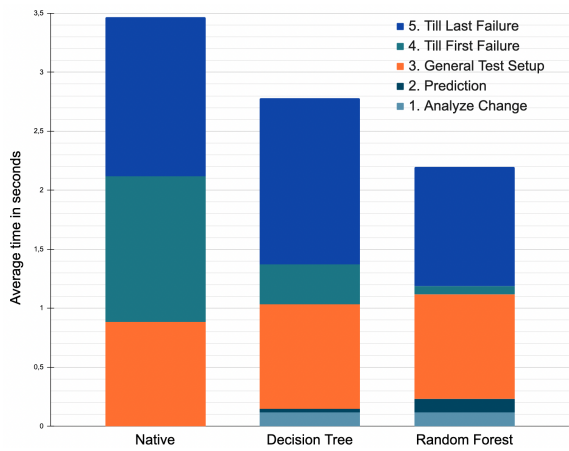
Comparison to Other Test Case Prioritisation Techniques. Showing immediate test feedback worked already well with our basic decision tree model trained on the mutation testing dataset. It does, however, not perform well in common metrics that consider the whole test suite, like APFD. We want to test whether other algorithms also perform in the immediate setting, or if they can be modified to provide even better results.

6.2 Data Model Improvements

Modeling of Context. Currently, the approximation of the context depends on what is seen by developers. The decision on what is relevant to developers could be better approximated if the different granularity of knowledge in the context is taken into account. If a method is studied thoroughly, tests highly correlated to that method



(a) Improvements in Test Execution Timings



(b) Time Relation between Different Workflow Steps

Figure 4: Comparison of performance of different test case prioritisation algorithms

should be ranked relevant. If developers switch back and forth between different files, tests that explain the interplay between the different files could be ranked higher instead of highly specific tests.

Incremental Updates. Currently, the data source that is used to generate the relevance scores and train the prediction model is static and gets worse if the underlying software system is changed. The mutation testing process for flask takes around one hour if executed in parallel on a server. One way might be to allow incremental updates on the data source, for example, to only re-run mutation testing on changed methods to allow for faster update times.

6.3 User Experience

Improved Visualisation. Improving the visualisation techniques can happen by increasing expressiveness or by increasing effectiveness for developers. One idea to increase the expressiveness is to additionally visualise the confidence in the current state of a test case. If a test has been executed just now, its outcome is

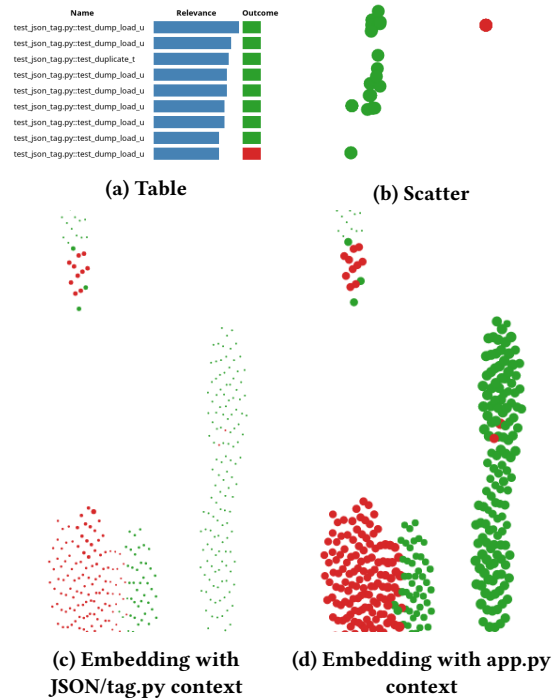


Figure 5: Appearance of different relevance metric visualisations

certain. Tests that were not executed for a long time are more uncertain. Even tests that have higher execution time than necessary for immediate feedback could give interesting information if the visualisation supported the notion of 'very likely to fail'. Developers could then look at these tests if they are relevant to the context. The effectiveness could be increased by removing information that is unnecessary for developers. This could include grouping or hiding irrelevant test cases so that developers can identify the relevant test cases faster.

Additional Interaction Techniques. Additional interaction to help developers in the observation phase might be valuable. An example would be an automated debug session that is automatically started showing the values variables take in the source code for the highest relevant test. This would give developers even more information that is closely linked to the source code about their current context.

Fine-tuning Through Developer Feedback. The prediction model is currently not learning from the test runs that happen if developers edit code. It could be retrained on the feedback of current test runs and then prioritise tests that failed previously higher.

7 CONCLUSION

We proposed a new model to think about feedback from test suites that differentiates between the two phases of observation and experimentation during software comprehension. Furthermore, we explored how to adjust feedback according to the two phases.

Our presented prototype uses the testing framework of existing software projects as a tool to generate immediate feedback and

enable exploratory programming. It is especially useful for getting to know a software project fast and testing assumptions live in the system.

In our case study, we used the **TFIDF** model to rank test cases for developers to help them understand their context during observation of the source code. We discovered that existing metrics for test-case prioritisation are sub-optimal for immediate feedback, as they make weak statements about the feedback time for developers.

We implemented simple mutation-based test-case prioritisation techniques and showed that these techniques already provide the possibility to shorten feedback cycles enough to provide faster feedback.

Our approach is widely applicable, as it has low requirements that have to be met to enable it. Based on our case study, we argue that test suites can be combined with machine learning to provide developers with additional tools for exploratory programming.

ACKNOWLEDGMENTS

This research has been supported by the Federal Ministry of Education and Research of Germany (BMBF) in the KI-LAB-ITSE framework (project number 01IS19066).

REFERENCES

- [1] Jim Dabrowski and Ethan V. Munson. 2011. 40years of Searching for the Best Computer System Response Time. *Interact. Comput.* 23, 5 (Sept. 2011), 555–564. <https://doi.org/10.1016/j.intcom.2011.05.008>
- [2] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37 (09 2011), 649–678. <https://doi.org/10.1109/TSE.2010.62>
- [3] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) (FSE 2014). Association for Computing Machinery, New York, NY, USA, 654–665. <https://doi.org/10.1145/2635868.2635929>
- [4] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. Chapter One - A Survey on Regression Test-Case Prioritization. *Advances in Computers*, Vol. 113. Elsevier, 1 – 46. <https://doi.org/10.1016/bs.adcom.2018.10.001>
- [5] Toni Mattis and Robert Hirschfeld. 2020. Lightweight Lexical Test Prioritization for Immediate Feedback. *The Art, Science, and Engineering of Programming* 4, 3 (Feb 2020). <https://doi.org/10.22152/programming-journal.org/2020/4/12>
- [6] Anneliese Mayrhofer and A. Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28 (09 1995), 44 – 55. <https://doi.org/10.1109/2.402076>
- [7] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. (07 2018). <https://doi.org/10.22152/programming-journal.org/2019/3/1>
- [8] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523. [https://doi.org/10.1016/0306-4573\(88\)90021-0](https://doi.org/10.1016/0306-4573(88)90021-0)
- [9] Steven C. Seow. 2008. *Designing and Engineering Time: The Psychology of Time Perception in Software* (1 ed.). Addison-Wesley Professional.
- [10] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127–139. [https://doi.org/10.1016/S1045-926X\(05\)80012-6](https://doi.org/10.1016/S1045-926X(05)80012-6)
- [11] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).