CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Enabling test case selection of automatically generated unit tests through traceability

An emperical study

Bachelor of Science Thesis in Software Engineering and Management

MARIAM JOBE
MARIAM MAHBOOB

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Enabling test case selection of automatically generated unit tests through traceability
An emperical study

Mariam Jobe
Mariam Mahboob

Supervisor: Fransisco Gomes
Supervisor: Salome Maro
Examiner: Truong Ho Quang

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Enabling test case selection of automatically generated unit tests through traceability

Mariam Mahboob
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
Email:gusmahbma@student.gu.se

Academic Supervisor
Francisco Gomes

Mariam Jobe
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
Email:gusjobema@student.gu.se

Academic Supervisor
Salome Maro

***Abstract*—Anticipating the effects of changes/modifications to source code, is a difficult if not an impossible process, unless the right tools or methods are applied. One way of handling change impact analysis is through test case selection which can cut down the testing time, as it only selects and runs the tests that have been affected by a change. In order to realize this, the method of traceability is applied on source code and automatically generated unit tests. This approach aims at facilitating software maintenance by cutting down the time and the effort required to re-validate changes. This paper investigates the impact of traceability with the intention of evaluating the effects on debugging time and mutation kill score. By conducting an experiment with 8 subjects, the results showed that no major statistical differences were found, which is likely to change with a larger sample size. Nonetheless, to generalize the impact of traceability between code and automated unit tests, further research is required, however the paper provides insights and deeper understanding of the problem as well a guideline for future studies.**

I. Introduction

Traceability refers to the capability of relating data kept within artifacts, such as documentation, UML diagrams, source code etc. and provides a way to analyze this connection. In order to realize traceability, navigable links have to be created between the artifacts. The concept of traceability consists of trace artifacts, the traceable components, and trace links, the association between two or more artifacts. Traceability links can be visualized to provide a deeper understanding of the system under development in relation to its corresponding development activities (testing, implementation, maintenance, etc.) and artifacts (requirements, tests, source-code) [1]. In turn, software testing is an essential part of the development process, but can be quite costly and tedious. Unit testing tests a small part of the system such as a method or a class and this process can be automated or manual [2]. Many different tools exist for automating unit tests and Evosuite is popular in the test research community, being widely compered with different generation tools [1] which is a search based software testing tool.

Test-to-code traceability plays a significant role in reducing time consumption of maintenance work and software evolution [3], and can, therefore, be valuable for an organization with a complex code base, for instance a large object-oriented programming code in Java, where a change in one class affects other classes. By creating traces between these objects and their tests, it can save the developer time by scaling down the amount of regression tests they have to run during development time. It also plays a significantly important role in reducing time consumption of maintenance work [4]. Here, we propose an approach where trace links will be created between the source code and automatically generated unit tests in order to mitigate the unit testing effort. Our approach proposes a solution to this problem by using traceability to create trace links when a modified part of the code triggers a fault in another part of the system.

Software maintenance is a software artifact that endures a change to the code and related documentation as a result of a problem or when a new feature is introduced or when something needs to be upgraded. Maintenance work is also necessary to enhance quality attributes such as performance or when the software system has to settle in to a new or changed environment [5]. Some approaches to maintain the source code is solved by using regression tests, i.e. tests are generated and executed and once changes have been made to the source code, the tests are re-executed to see if the changes have broken any existing parts of the code [6]. The constant testing, along with the increasing size of the test suite, increases testing costs significantly at the unit level and deviates effort in meeting sprint deadlines and/or demonstrations.

According to Weinberg [7], one of the biggest problems in maintenance is “unexpected linkages”. This refers to how a change in one part of the program can have an effect on other parts of the system.

To explore the impact of traceability between source code and automated unit tests, an empirical study is conducted in order to evaluate our hypothesis that there is a difference in (i) debugging time, (ii) mutation kill score and (iii) the average amount debug time/per mutant, when traceability is implemented.

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

A. Problem Statement and Proposed Solution

The research problem that is being investigated in this paper is to find out what impact traceability has on test case selection and error detection. We use the example below to illustrate how our approach can be used during a development activity.

Software projects usually consist of dozens of classes. Assuming a developer is working with an arbitrary class A, he or she may be unaware that changes in A can affect other classes in the code, e.g. B. The traces would suggest that the developer should run regression tests for all of the involved classes [8], namely A, B. This allows the developer to see if their added changes have affected the involved classes but also saves them time since they don't have to run all of the test suites available. Instead, tester runs only the ones that have trace-links in between them, view Figure 1. The benefit of running a smaller amount of regression tests during development, is that it saves time and the developer can get quicker feedback of what their changes are affecting. Although running all of the tests might be more reliable, it would be infeasible in practice due to the time constraint.

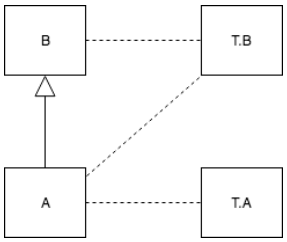


Fig. 1: Trace creation exmple: arrow represents inheritance, dotted line represents the trace-links created with Capra

Automated tracing refers to implementing traceability with automated techniques [1]. The traceability tool Capra [2] can currently create trace-links between different kinds of artifacts manually, but have yet to reach automatic execution, which we implement in our approach. This paper intends to evaluate the impact of automated tracing between source code and automated unit tests.

In our approach, the initial setup consists of an open source project, from which the automated unit tests will be generated through Evosuite [3][9], view Figure 3. Initially there are only trace links between each class and their corresponding test class. Then, these trace links are generated automatically by Capra, whenever we use Evosuite to automatically generate unit tests both before and after the source code is refactored and tested. Ultimately, faults are detected as a result of automatically executing tests of modified classes (see Figure 2), view Figure 2.

Here is a concrete scenario, during development, a developer refactors a single class, lets call it class A. In addition, there are already some trace links generated previously by Capra, as well as a set of unit tests generated by Evosuite. At this point, keep in mind that the tester has a package of classes and a package of unit tests, testing those classes. In summary, a one-to-one trace (between class and tests), in Figure 2. Once this is done, they will run a regression test on the test class that has the trace link with the current class, which in this case would be test TestA. If the test reveals any failures, the developer will have to correct them until all of the methods in TestA pass. Once the developer is finished with class A, i.e he or she get a green bar from the unit tests, they will run regression tests for all the tests that are available in the project. Once the tests are done executing, it results in a report listing all the passing and failing tests. If the current class A has triggered faults in other classes (e.g the modification in A created a fault as a side effect), lets say class B, a trace link will be created between Class A and class B's test, i.e Test B, view Figure 1. Once the traces have been created the developer can view them in a tracability diagram.

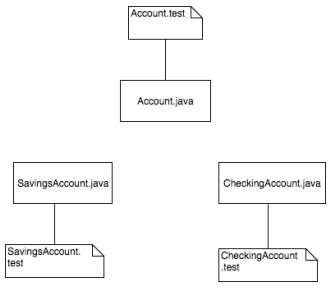


Fig. 2: Trace creation example.

Before running all the regression tests, a developer might be unaware of the dependencies between the classes(e.g the inheritance between A and B in this case). He or she can use code information to infer relationship between classes, but even so, for a big project with dozens or hundreds of classes, those relationships become overwhelming and hard to keep track of. Now, with the help of the trace-links between classes and tests, the developer has the ability to view which trace-links are available in a diagram provided by Capra. Then they will execute the regression tests that are displayed in the diagram (in our example, both TestA and TestB) to make the necessary corrections to make the tests pass. The process repeats such that once all tests linked to the classes pass, the developer will run the regression tests again, such that new traces (possible a consequences of modifying B) may be reveled (or not).

In conclusion, traces are automatically created based on faults detected from running regression tests. Therefore, to update the traces in a simultaneous flow, the technique of integration and regression testing is applied, which is widely practiced in automated unit testing to ensure acceptance and satisfaction for a given piece of code [10].

To narrow down the scope and minimize the complexity of the problem, regression testing will begin with the generated tests that are initially passing (resolved). The trace links will be updated once the code has been changed and then the regression tests will be run to provide input for the trace links. The input for the trace-links is the failing tests that have been triggered by a single class.

To solve this problem we will create automated tests with Evosuite, which is search-based software testing (SBST) tool [11]. We choose Evosuite since it is a well established tool in

[2]https://projects.eclipse.org/projects/modeling.capra
[3]https://www.evosuite.org

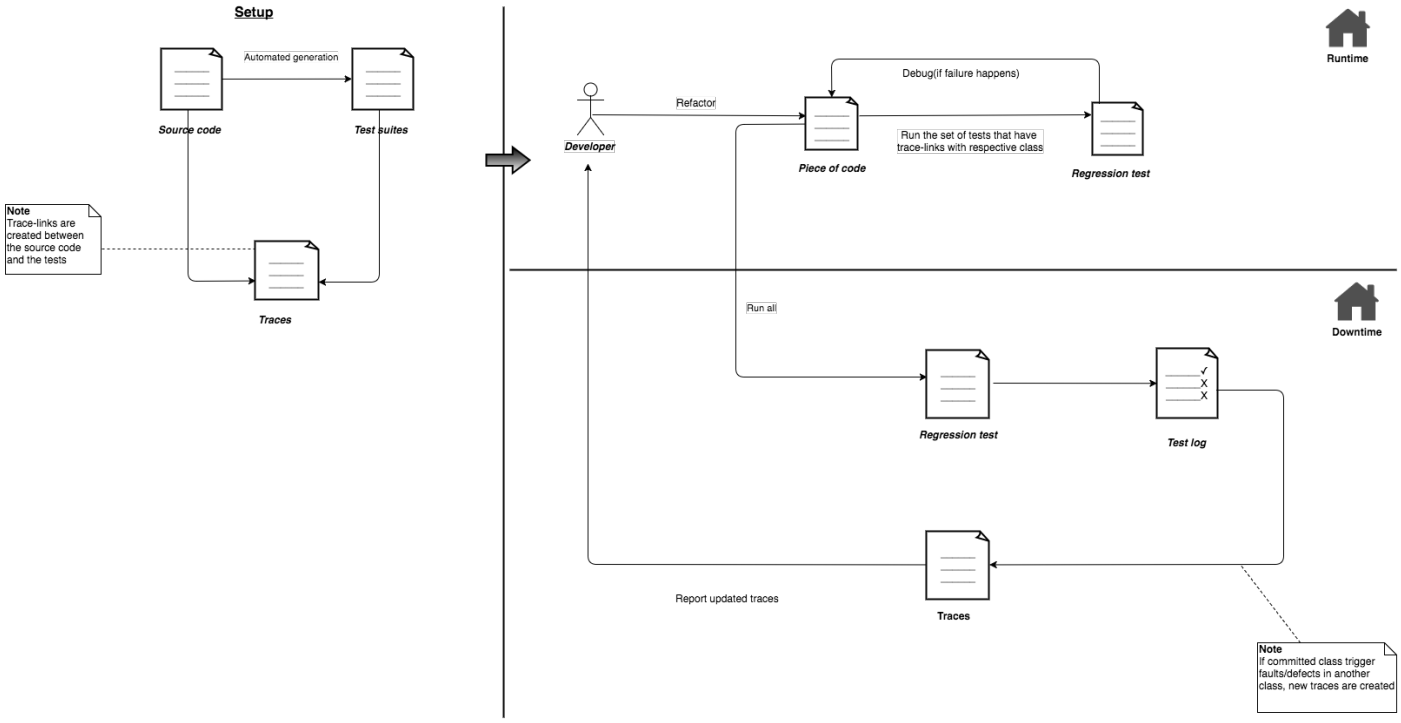


Fig. 3: Process overview

academia, which has reached a good maturity level and since it supports Java, which is a programming language familiar to us. In order to evaluate our approach, we will generate tests from an open source Bank application project and to create all of the trace links we use Capra.

B. Contributions of the paper

The scientific contribution is to fill the gap in the research field of automated unit testing and traceability. The aim is to investigate the impact of applying the technique of traceability between source code and automated unit tests.

The technical contribution, is a methodological guideline for using test case selection through traceability on automated unit tests. Additionally, we are looking to extend Capra's existing functionality by developing a feature that automatically creates trace links between source code and automated unit tests.

This thesis provides contribution towards the following development activities:

Change impact analysis: By creating traces between the code base and the automated unit tests, a visualization aspect is provided. This is helpful in identifying missing elements and helps the developer in resolving maintenance and development problems more efficiently [12]. It assists the developer in finding which classes to refactor next and gives a better understanding in how changes affect other parts of the system and can provide better estimation of development, testing and maintenance effort as well as time to completion.

Test case selection: The more regression tests that are run, the greater the chance is of having strong re-validation results, however, it is a tedious process which requires time and resources, that might not be feasible in practice [13]. Narrowing down the amount of regression tests that need to be run helps the developers save time as limited test results needs to be analysed and debugged each time. Since all of the regression tests are run during the night, quality won't be compromised to a great extent.

C. Research Questions

The following research questions will be investigated:

RQ 1 What effect does traceability have on test case selection?
RQ 1.1 How does traceability affect the developer's time to fix defects?

RQ 2 How is regression testing affected by traceability between source code and unit tests?
RQ 2.1 How does traceability influence the number of faults detected and fixed?

The rest of the paper is structured according to the following: Section II covers the background, followed by related work in section III. The research methodology, concerning the implementation of the Capra extension as well all the elements of the empirical study and survey structure can be found in section IV. The results and analysis of the descriptive statistics along with the hypothesis testing is uncovered section V. The discussion section VI goes more in depth regarding the research questions. Finally the validity threats can be found in section VII followed conclusion and future work in sections VIII and XI.

II. Literature Review

A. Background

1) Traceability: Traceability aims at creating navigable links between between data that is contained within some artifact, that otherwise wouldn't have a association. Traceability can facilitate various kinds of analysis used in the software engineering field and on top of that provides a visualization standpoint which provides a better insight of the system under development [1]. Any traceable unit of data is referred as trace artifacts which can be applied to UML class diagram, single class or a particular class operations. Traceability links between the artifacts helps in mapping high-level documents (abstracts concepts) to low-level artifacts [4]. The application of traceability between developed code and test classes is done to facilitate the process of testing with reduced time and cost consumption and better quality assurance.

2) Automated unit testing: Test and verification is an essential part of software development since it is fundamental that the code is working as anticipated and does not contain defects. Unit testing is procedure in which individual parts, such as functions or classes, of a program are independently tested to find faults. Unit tests can be manually written or automatically generated. The challenges of automated unit testing is generating significant input for the test cases and to establish if the outcome of the tests is passing or failing, which is called an oracle [2].

3) Evosuite: Evosuite is a search-based software testing (SBST) tool that provides automatic generation of test suites for Java classes by using a generic algorithm (a population of possible solutions is progressed using procedures inspired by natural evolution). The algorithm continuously generates test cases until it meets a solution or stopping state that, in turn, is related to a coverage threshold (e.g branch coverage, statement coverage, condition coverage) [11]. Evosuite generates test suites to satisfy a coverage criterion [9].

4) Regression testing: Regression testing is carried out in order to validate that a changes made in the software program has not caused any defects in the existing code. New test cases are conducted for testing new features and old tests are rerun to see if the program gives any exceptions, therefore with the evolution of program, the expense of regression testing grows, therefore, different methods have been introduced for test selection e.g modification based test selection , selection and prioritization based test selection etc. [13].

5) Software Maintenance: Software maintenance is defined as the process of changes made to a software product after delivery to the client, in order to adjust/improve the system according to new user requirements, faults detected in need of correction or to adapt the software to a new or changing environment [5] [14]. Keeping the software up to date and stable is also considered as maintenance.

Corrective maintenance is referred to modifications made to amend faults in the software[14]. This includes validation, which is the activity of confirming that changes made to the software works properly and that the changes have not broken existing parts. Object oriented programs (OOP) were believed to be the solution to maintenance problems, as it is easier to make changes to them, but OOP also introduces a set of new problems due to inheritance and other relationships between classes in the code. Changes in one part of the program can cause problems in other parts of the software which always needs to be taken into consideration when making modifications[5].

6) Maintenance of automated unit tests: Maintaining automated unit testing, such as JUnit test cases, can be an endless processes, especially in bigger companies with a large code base. Regression tests usually helps to find a large percentage of the defects of a new release. Updating or removing each failing test can consume a lot time from the developers which could be spent on developing new functionality. In an experiment by Robinson et al. [6], Randoop's test generation technique was extended to create more effective and maintainable regression tests for real software programs by comparing the Randoop generated tests with manually written tests (by humans) in terms of coverage and mutation kill score (amount of mutations[4] detected by a test suite).

By generating a test suite with string literals, removing lexically redundant tests and disabling observer functions, they found that the automatically generated regression test suite covered more than half the code and was able to detect more defects. To evaluate maintenance of the test suites, they determined the effort required to maintain them as well as analyzing the impact removing redundant tests on maintainability. The study found that the new implementation provided test suites with higher coverage and mutation kills score and the suites were easily understandable and maintainable for the developers [6]. We choose Evosuite over Randoop, because it is more efficient in terms of coverage and finding faults [9].

III. Related work

Test case prioritization can be used in order to schedule the execution of the regression tests to re-validate the functionality of the tests with the highest priority to stay within the time and budget constraints [13]. Regression testing can otherwise be a tedious and time consuming process and is also very costly, but by selecting the a suitable part of the test suite, expenses and time can be cut down significantly. The drawbacks is that the fault detection abilities can be reduced. There are different test case prioritizing techniques such as scheduling the tests in the order that reaches the most coverage in a short time, or testing first the frequently used features. The results of an experiment Rothermel et al. [15] showed that prioritizing test cases can increase the frequency of fault detection. The paper mentions many cases of prioritization but does not discuss selecting test suites based on dependencies between classes.

Regression testing using slicing is a selective method of regression testing by using program slicing [16]. Slicing is all the statements and conditionals that can have an effect on the value of a variable at a certain point. This is called a definition-use pair, such that definition is the location (i.e statement) where the variable is assigned and use is all the sub-paths where the variable is being used. The def-use pairs that have been affected by a change are used to generate and execute tests. This technique reduce costs on maintaining test suites as

[4]Mutant: To evaluate test techniques, the code is systematically modified to include a fault, named mutant. The goal of the evaluation is to compare which techniques reveals failures by triggering the mutants.

it cuts down the amount of test suites that need to be rerun whilst still achieving high coverage [16]. The paper has an effective solution to test case selection, but does not have the visual aspect that has been proven to be helpful for developers as it makes them understand connections between different artifacts better [4].

Automated unit tests uncover faults and report on the coverage, but the debugging part is still a manual procedure which can be quite intensive. In an experiment, Ceccato et al. [17], the variances of automated unit tests and manually written test was investigated in regards to debugging in terms of the ability of the developers to accurately debug the code. The study compared manual tests with automatically generated tests in Evosuite and Randoop (random test generation tool). The findings showed that complexity of the actual test cases has a bigger affect on effectiveness and efficiency of debugging rather than the meaningfulness identifiers that can be found in automatically generated test cases. These identifiers are usually easy to comprehend. The automated test cases were more effective than the manual ones, for subjects with intermediate experience while less experienced subjects felt like understanding the test code is more important to be able to debug. The general results showed that automatically generated test cases has a minor factor on the difficulty of debugging while other factors such as experience, ability and complexity of the test cases plays a much bigger part [17]. Although there was no big difference in the difficulty of debugging, having a traceability aspect could increase the efficiency of debugging and also provide defects that are more related to the developers current work.

The process of generating traces between test-and-code facilitates the process of test driven development and software evolution. It keeps the unit test cases synchronized with the source code that might continuously change throughout the development process. Each test case may be linked to several methods under test and vice-versa, therefore, the heuristic for their detection is very beneficial during development and maintenance. The study by Ghafari et.al [3], examines the relationship between source code and test code on a method level by identifying focal methods in unit test cases (the methods accountable changing the system status). Our research, however, provides visibility of the trace-links through Capra, unlike this study. Although the study is more low level, since it operates on methods, the visualization aspect and constant feedback distinguishes our approach.

IV. Research Methodology

This sections describes the extension of the traceability tool Capra that has been developed, and all the elements and factors involved in the empirical study.

A. Development

To introduce the technique of traceability in our study, we are using a software called ”Capra”, which helps to create trace links among different artifacts such as documentation, source code, tests etc. The tool currently requires manual creation of the links. We have developed an addition to the tool that can make trace links automatically between the source code and the tests. The functionality we have implemented provide two features. It creates a link between each class and their corresponding test, view Figure 4 and it also creates a link between each failing test and the class that causes it, Figure 5 .

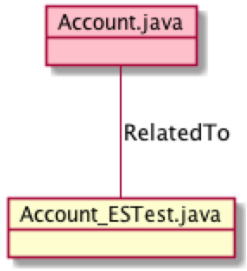


Fig. 4: Capra: Single trace link between class and test

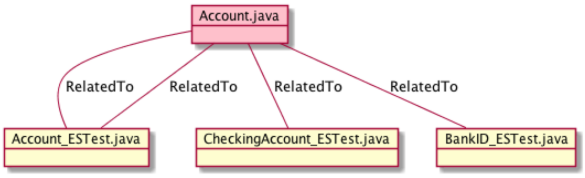


Fig. 5: Capra: Trace between class and failing tests

Before we started working on creating the trace-links we had to generate the tests with Evosuite. We used Maven to compile and build the tests and Evosuite provides different features for test generation by allowing you to choose if you want to generate tests for a class or package for example. Therefore we began by implementing a class that generates tests for all of the java classes in a package. Evosuite generates two types of test files, ”ClassName”.scaffolding, which makes sure that the tests are executed in the same consistent state (acts as a dependency of sorts) and the ”ClassName”.ESTest, which contains the actual test cases which can be executed with JUnit.

Since Capra has a plugin for Eclipse IDE, we extended the functionality by creating an plugin project. This required us to use Eclipse Java Development Tools (jdt) [5], which supports development of java applications and plugin development. We also used some additional dependencies such as Eclipse core, which provides some basic platform infrastructure and Eclipse UI for the interaction with the Eclipse Platform User Interface. To create the initial traces, which are between each class and their test, we matched them by using their names which are equal. First we had to extract all of the files and place them into lists. Then we were able to compare the lists by finding where the file names are the same. When Evosuite generates tests, they are provided with the same name as their corresponding java file.

For the second part of the implementation is where the

[5]http://www.eclipse.org/jdt/

trace links are created between the class under modification/refactoring and the unit tests effected (failed) because of the changes made to the class. To implement this we created a file where the "developer"/user writes down the class that they are currently working on/refactoring. The file is read and added to a list A. Then the regression tests are ran on all of the unit tests in a package. The tests that *fail* are added to the same list A, and then the trace links are creating by iterating through the elements of the list. This results in traces that look like Figure 5. As can be seen from the figure, there is a double trace between Account and Account_ESTest, this is because each class is initially traced with its corresponding test. The second trace signifies that it the test has failed.

In our implementation we are assuming that no new classes are added to the project. The reason being that we want to run regression testing, and if a new class is added, a test needs to be generated for that class with Evosuite. The generation assumes that the class is correct which means that all tests passed for first version of the code,thus creating a baseline test suite. So running the newly generated test would not result in a failure even if the class has a defect. This is a function of Evosuite, but is noted here as a limitation of the study.

B. Empirical Study

The scope is very limited due to time and resource constraints and the study does not aim at generalization at this stage, rather to provide insights and familiarity for later investigation of test-to-code traceability. Nonetheless, we have been careful to provide controllable execution, however the scope was too limited to perform an actual experiment.

The experimental process, according to Wohlin et.al.[18], abides by the following stages: (i) Scoping, (ii) Planning, (iii) Operation (iv) Analysis , (v) Packaging. In turn, Operation is divided into two distinct parts, namely Instrumentation and Execution, where the former is the process of deploying tools and apparatus used in the experiments, while the latter is the actual execution of the experiment.

C. Evaluation

To address the research questions we conducted an empirical study. We used an open source project as the unit, which was a simple "BankSystem" project written in Java. A total of 8 subjects from the Software Engineering and Management program participated. The goal was to determine how test case selection affects error detection time and mutation kill score. To observe the behaviour of the subjects, the computer screens was recorded and monitored during each participant's execution. The time was calculated for how long it takes for the subjects to find and kill the seeded faults (i.e mutants) added in the source code. Based on the collected data we conducted hypothesis testing and statistical analysis. We have taken into consideration that the participants might have different levels of skills and experience and this will be regarded as blocked factors, i.e are not of interest in the results.

D. Scoping

The foundation of the study is the following:

Analyze the *debugging supported by traceability*,
for the purpose of *exploring benefits and drawbacks*,
with respect to their *efficiency and effectiveness*,
from the point of view of the *developer*,
in the context of *continuous testing*

E. Planning and Operation

The following variables have been identified for our study:

- Independent variables: Traceability, which has two levels: applied and not applied.
- Controlled variable: Seeded faults are implemented to simulate faults in the code to be able to make the measurements on the dependent variables. The tests will be automated in Evosuite and traceability links are created with Capra through our implementation.
- Dependent variables: The dependent variables will measure different aspects of the generated test suites and the traceability in terms of (i) Mutation Kill Score, (ii) Time spent on debugging.

To ensure controllability of the execution, seeded faults were added to the source code to analyze the performance of our approach. Fault Seeding or be-bugging is the process of inserting artificial faults in the software. This is done to get the artificial faults which are discovered through software testing along with and real faults [19]. Faults were added in the super-class, so that the faults would propagate down the hierarchy in classes that were extended or inherited by it. By doing so, it is possible to record how long it takes to debug those faults and how many are detected.

The instruments consist of the following contents:

- A guide: stating what the subjects are going to do, how to find the selected test suites through the traceability-links, how to run the regression tests, how many defects to find and how to refactor, view Appendix A.
- The source code: each subject will be provided with the source code that they will be refactored by the subjects.
- Traceability-links: displayed in a traceability matrix or as a visual diagram.
- A survey: that will be filled by the subjects after execution is done".

F. Hypothesis

Our informal hypothesis is that code that has traceability applied with the unit tests yield different results in terms of efficiency and effectiveness in debugging time and mutation kill score as compared to source code where traceability is not applied. Based on this statement it is possible to state a formal hypothesis:

Hypothesis 1 (H1): *Null hypothesis,* H_0 *There is no difference in efficiency (measured in debugging time) when traceability applied (TA) as to when traceability is not applied (TNA).*

H_0: Debug time(TA) = Debug time(TNA)
H_1: Debug time(TA) $\neq$ Debug time(TNA)
Measures needed: debugging time (minutes)

Hypothesis 2 (H2): *Null hypothesis, H_0 There is no difference in effectiveness (measured in mutation kill score) when traceability is applied (TA) as to when traceability is not applied (TNA).*

H_0: Mutation kill score(TA) = Mutation kill score(TNA)
H_1: Mutation kill score(TA) $\neq$ Mutation kill score(TNA)
Measures needed: mutation kill score

Hypothesis 3 (H3): *Null hypothesis, H_0 There is no difference in the average debugging time of a defect when traceability is applied (TA) as to when traceability is not applied (TNA).*

H_0: DebugTimePerMutant(TA) = DebugTimePerMutant(TNA)
H_1: DebugTimePerMutant(TA) $\neq$ DebugTimePerMutant(TNA)
Measures needed: debugging time and mutation kill score

G. Design

The subjects were randomly divided into two groups. Half of them are provided with the traceability diagram which displays the selected test suites identified with traceability. These subjects only needed to run regression tests on the test suites present in the diagram. However, the other half were not provided with such a diagram. Instead they had to re-run all of the regression tests followed by debugging. Nonetheless, both groups had to do the same kind of operations i.e run regression tests on the automatically generated unit tests and debug the defects found. Seeded faults were added to the source code before hand, so the code already contained the faults when its provided to the subjects. Before the subjects started, they were provided with a guide and then we took a few minutes to further explain additional details about the procedure. After completion, the subjects were asked to answer a survey in order to collect the qualitative data.

H. Analysis and Packaging

The data obtained from the subjects is analyzed by applying statistical methods using the tool RStudio [6]. The data will be analyzed for normality using Shapiro Wilk-test and graphically through qq-plots. Depending on the how the data is distributed, we will follow up by running a two sample test with either t-test or Wilcoxcon test. The qualitative data acquired from the surveys is used to draw further conclusions about the results.

I. Survey

To gather further insight we conducted a survey and to do so we used an online tool called SurveyMonkey [7]. A combination of open-ended and closed questions were applied, in which the latter was used for both subject groups (with and without traceability) and the former was more specific questions to each group about their experience in terms of difficulty, advantages and disadvantages of each method.

1) Coding survey data: Since we included open ended questions, we decided to code the data by going through it and finding some common themes in the answers, that could be made into categories. This was summarized into a table, with three columns, (i) code, (ii) margin note and (iii) quote, where the code is the category, the margin note specifies a more direct relation to the quote, which is a quote pulled from the survey data.

J. Data collection

In this study we have gathered both quantitative data from the screen recordings we recorded during the empirical study, and some from the surveys. We were also able to gather qualitative data from the open questions in the survey.

1) Screen recordings: For both parts of the empirical study, the screens were recorded. We later analyzed those recording to get the number of mutants killed per total number of mutants, and the total time it took to debug the faults detected, view Table III and Table IV. This will in turn give an average amount of time to find and debug one mutant.

2) Survey data: The questionnaire used a combination of open an closed questions. The closed questions used a matrix where the subject had to give a rate between 1 (strongly disagree) and 5 (strongly agree). In the questionnaire, we gathered information about how the participants experienced the task they had been given in terms of understand ability and attainability. Furthermore, it included questions about the debugging process where the subject had to rate their experience. Participants were asked question regarding the understandability and usability of the tool, their experience with traceability tool. For the participants without traceability applied, we were interested in knowing how they experienced the debugging process and what they found to be most difficult. For instance, in Figure 6, the question would be "To what extent do you agree that it was difficult to debug?

	1	2	3	4	5	TotalI
Difficulty level	0,00% 0	25,00% 1	75,00% 3	0,00% 0	0,00% 0	4
Time consumption	0,00% 0	0,00% 0	100,00% 4	0,00% 0	0,00% 0	4
Laborious	0,00% 0	75,00% 3	25,00% 1	0,00% 0	0,00% 0	4
Understanding the task	25,00% 1	25,00% 1	0,00% 0	25,00% 1	25,00% 1	4

Fig. 6: Survey data: Results of traceability applied

	1	2	3	4	5	TotalI
Difficulty level	0,00% 0	25,00% 1	0,00% 0	50,00% 2	25,00% 1	4
Time consumption	0,00% 0	25,00% 1	0,00% 0	50,00% 2	25,00% 1	4
Laborious	0,00% 0	25,00% 1	50,00% 2	25,00% 1	0,00% 0	4
Understanding the task	0,00% 0	25,00% 1	0,00% 0	25,00% 1	50,00% 2	4

Fig. 7: Survey Data: Results without traceability applied

[6]https://www.rstudio.com/
[7]https://www.surveymonkey.com/

The data collected from the open questions of the questionnaire can be found in Table I and Table II. The evaluation was done through coding, where the data is put into categories and subcategories.

V. Results & Analysis

The data sets consist of the debug time and the mutation kill score, and the average time per mutant, for the traceability and non-traceability. Since the data samples are quite small, the results might be affected, refer to Validity threats section VII. The data collected from the screen recordings of the Mutant kill score and Debug time can be found in Table III and IV.

A. Descriptive statistics

The descriptive statistics provides a general understanding of the data in respect to what can be anticipated from the hypothesis testing and what kind of problems might be present due to outliers [18].

Traceability vs. Debug Time From Table V, it is possible to see that traceability applied (TA) seems to have a significantly smaller average in debug time than traceability not applied (TNA) by looking at the difference in means.

TA has a mean value of 39.75 with a standard deviation of roughly 6.9. The standard deviation show the average extent to which the data points branch of from the mean. For the TNA the mean is 50 with a standard deviation of 5.9. Since the data is bit skewed it is also of good practice to look at the medians to get a more accurate value of the central tendency, which are 40.5 for TA and 51.5 from TNA.

None of the variances is 0 which indicates that there is in fact variance in the data values of the samples. To gain better understanding of the data we used a boxplot, Figure 8. The boxplot demonstrates that the values of DebugTime (TA) is more left skewed, meaning that the data points fall lower down on the scale and DebugTime (TNA) is more right skewed. By looking at the boxplot it is clear that the TA has shorter debugging time and it is also possible to see that there are no outliers in the data.

By looking at all of the above values, it is possible to conclude that the samples are not equal and therefore it should be feasible to discover the statistical differences by using hypothesis testing. A t-test is used for this hypothesis, view results in section B.

Traceability vs. Mutation Kill Score In Table VI, for traceability applied (TA) has a mean value of 3.5 with a standard deviation of 0.58. Comparing to traceability not applied (TNA) where there is a mean value of 2.75 with a standard deviation of 0.96. The low standard deviations of the two data sets tells us that the values are very close to the mean. The variance low variance of 0.3 and 0.9 signifies that values are very similar.

To further explore the data, it is possible to see that the boxplot in Figure 9, suggest that TA is right skewed, and has data points closer to the end of the scale, with a median of 3.5. TNA on the other hand is more left skewed and has data points on the lower spectrum with a median of 2.5. After inspecting the data, it shows that there is a difference between

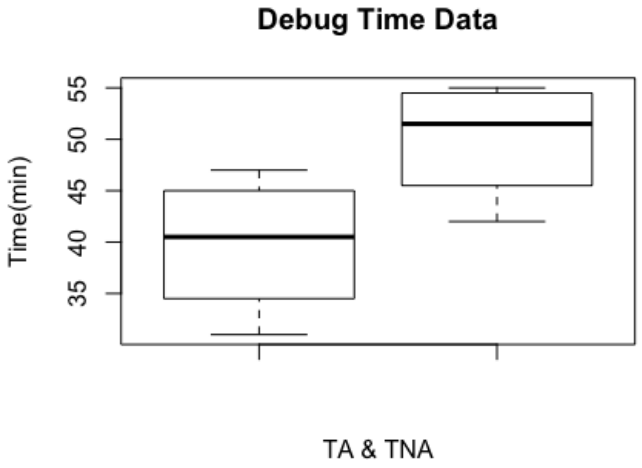


Fig. 8: Boxplot of debug time

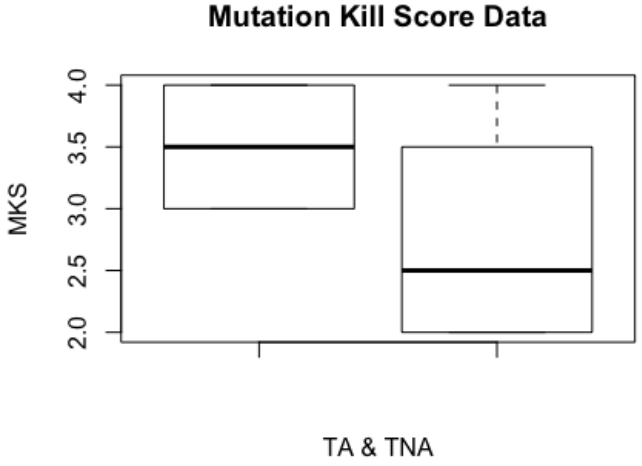


Fig. 9: Boxplot of mutation kill score

the samples, although it is quite small, but TA has a slightly higher mutation kill score than TNA. For this hypothesis a Wilcoxcon test is used, since one of the samples does not have normal distribution.

Traceability vs. Average Debug Time Per Mutant The values for the average time per mutant (ATPM) has been calculated by dividing the total debug time with the mutation kill score from each participant in RStudio.

Table VII, displays the values of central tendency of the data. The mean of ATPM is notably smaller when traceability is applied (TA), indicating that it has a shorter ATPM. The mean value for TA is 11.71 with a standard deviation of approximately 11.1, which is almost as high as the average, indicating that the is not very concentrated. The mean value for the TNA is 19.3 and has a standard deviation of 20.8, which is even less concentrated than the previous one, meaning that

TABLE I: Survey data: Traceability Applied

Code	Margin Note	Quote
Debugging Process	making it faster/efficient	It makes it easier to localize the error/problem, as it provides a a clear picture of all the components related.
Debugging Process	held back	It didn't held back in any way otherwise it looks like it would take forever to find mutant in the code.
Software Maintenance	tracking issues	It was helpful and would be even more beneficial for big systems with a lot of inheritances involved.
Software Maintenance	understanding relation between components	Traceability link view shows clear picture of classes relation once code is modified or altered.

TABLE II: Survey data: Traceability Not Applied

Code	Margin Note	Quote
Debugging Process	difficulties faced	It was hard to find the defects and took a long time to run all the tests, debugging someone else's code makes it harder as inheritances between classes is unknown at first.
Debugging Process	ease faced	The error message and failure trace made it easier to find out where the problem was lying.
Debugging Process	future facilitation	Knowing association among classes would be helpful as it would tell which test cases needs to be run. It will also be proved feasible for the process if test cases could be run altogether.

there are big differences in the data values.

What is noteworthy is that the mean and median are equal, which can be an indication that the data is normally distributed, but this is further evaluated in section B. Hypothesis testing.

The measures of dispersion shows that the variance is 3.3 in TA and 4.6 in TNA, which indicates that the values in the data sets are quite different.

Moreover, to visualize the data better we used a boxplot, Figure 10. The plot demonstrates that the TA data in comparison to TNA, suggesting that the ATPM are closer together in the data set. There are no outliers in both data sets.

B. Hypothesis testing

In this study we want to compare one factor (traceability) with two treatments (applied and not applied).

Traceability vs Debug time:For the statistical analysis of the sample "Debug_time", we started by examining if data set was normally distributed both graphically and numerically. A Shapiro-Wilk test was executed to test the null hypothesis that the samples come from a normal distribution.

The test resulted in:

$$DebugTime(TA) : p - value = 0.9022$$
$$DebugTime(TNA) : p - value = 0.43$$

Since none of the values are 0.05 or lower, the test results cannot reject the hypothesis of normality, i.e no there is no significant departure from normality.

The scatter plot shows a graphical representation of how well the plotted points follow the normal line, Figure 11a, and Figure 11a. Both figures show that the points lie quite close to the line and it is common that the points from either end of the line be a bit farther away from the line, i.e the plots point to normal distribution.

The first hypothesis is regarding quicker debug time when traceability is applied as compared to debugging without traceability, which is evaluated using two sample t-test. A t-test is a parametric test used to compare two sample means, where the design is one factor with two levels [18]. In this case the factor is Traceability, with two levels applied and not applied. The null hypothesis we are testing is that the debugging time is the same when traceability is applied and when its not applied:

$$H_0 : DebugTime(TA) = DebugTime(TNA)$$

From the results of the t-test we can see that H_0 can be

TABLE III: Empirical study data: traceability

Subject	Mutation kill score(/5)	Debug time(min)
1	3	47
2	4	31
3	4	43
4	3	38

TABLE IV: Empirical study data: without traceability

Subject	Mutation kill score(/5)	Debug time(min)
1	3	54
2	2	39
3	4	55
4	2	49

TABLE V: Debug Time: Measures of central tendency and dispersion

Factor	Mean	Median	Variance	Standard Deviation
TA	39.75	40.5	47.58333	6.898067
TNA	50	51.5	35.33333	5.944185

rejected since the p-value is lower than 0.05, view Figure VIII. There is in fact a difference in debugging time when traceability is applied and when it has not been applied. The reason behind it will be further evaluated in the Summary section.

Traceability vs. Mutation kill score: The normality analysis for hypothesis 2 also evalutes it by using the Shapiro Wilk test which produced the following results:

$$MutationKillScore(TA) : p - value = 0.02386$$
$$MutationKillScore(TNA) : p - value = 0.2725$$

Since the first one has a value under 0.05, the it can reject the null hypothesis that the sample is normal. The second value however does not reject normality as it is bigger than 0.05.

To support the numerical representation of normality, a graphical test was also issued which produced the following diagrams in Figure 13a and Figure 13b. Figure 13a, shows that there is some skweness in the data and reflects that the data is not normally distributed. The second one however,Figure 13b, the points are reletivley close to the line, except for the ones at the beginning and the end which is common in normal distribution. Together with the numerical analysis, it points to the sample being normal.

From the second test we have to run the Wilcoxcon test (not paired) as it does not assume normal distribution in the data and which is the case for one of our samples. We want to test our null hypothesis that the mutation kill score is equal when traceability is applied and when it is not applied:

$$H_0 : MutationKillScore(TA) = MutationKillScore(TNA)$$

Based on a probability level of 0.05, the null hypothesis cannot be rejected since the p-value is greater than 0.05 Figure IX. Meaning that there is not a significant difference in mutation kill score when traceability is applied and when its not applied, find reasoning in Summary section.

Traceability vs. Average Debug Time Per Mutant Like the previous hypotheses, we used a Shapiro-Wilk test for the normality testing along with the qqplots, which produced the following results:

TABLE VI: Mutation Kill Score: Measures of central tendency and dispersion

Factor	Mean	Median	Variance	Standard Deviation
TA	3.5	3.5	0.3333333	0.5773503
TNA	2.75	2.5	0.9166667	0.9574271

TABLE VII: Average time per mutant: Measures of central tendency and dispersion

Factor	Mean	Median	Variance	Standard Deviation
TA	11.71	11.71	3.326981	11.0688
TNA	19.3125	19.5	4.561501	20.80729

TABLE VIII: Results from t-test: debug time

Factor	Mean diff	Degrees of freedom	t-value	p-value
TA vc.TNA	10.25	6	-2.2513	0.03266

$$AverageDebugTimePerMutant(TA) : p - value = 1$$
$$AverageDebugTimePerMutant(TNA): p - value = 0.988$$

None of the values are below 0.05, indicating that the data has normal distribution. To further support this claim, we examined the qqplots which shows that both data sets follow the lines closely. Based on these observations, we assume that the data has normal distribution and will run a two sample t-test.

The hypothesis we are testing against is:

$$H_0 : DebugTimePerMutant(TA) = DebugTimePerMutant(TNA)$$

The t-test resulted in a p-value (0.03924) lower than the probability value 0.05, view Table X, and in so the null hypothesis can be rejected. This supports our alternative hypothesis that there is indeed a difference in the average debugging time per mutant when traceability is applied and traceability is not applied. The average debugging time is shorter when traceability is applied, this is further analyzed in the Summary section.

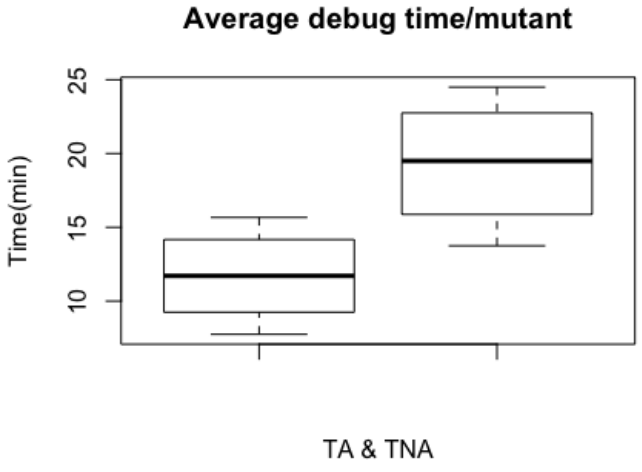
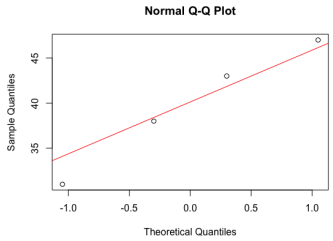
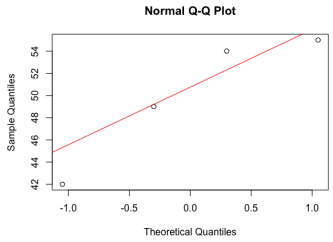


Fig. 10: Boxplot of averge debug time/per mutant

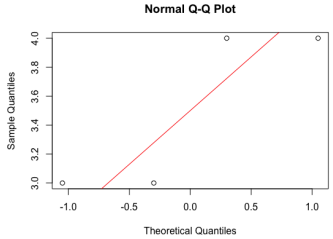


(a) Scatter plot: Debug Time (TA)

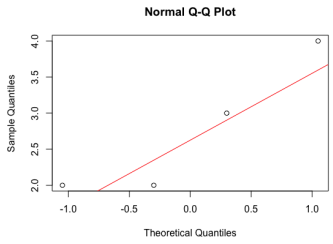


(b) Scatter plot: Debug Time (TNA)

Fig. 11: Graphical normality representation: Debug time



(a) Scatter plot: Mutation Kill Score (TA)



(b) Scatter plot: Mutation Kill Score (TNA)

Fig. 12: Graphical normality representation: Mutation Kill Score

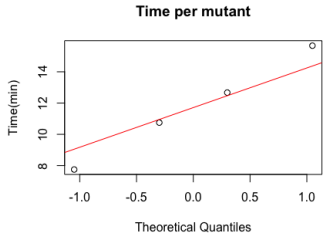
C. Summary

We have investigated two hypothesis: 1. Traceability vs. Debug time 2. Traceability vs. Mutation kill score 3. Traceability vs. Average debug time/per mutant

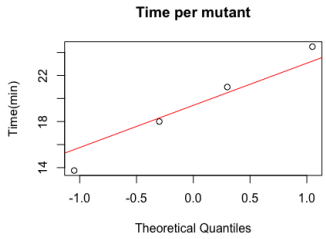
By running our hypothesis testing we were able to see that the subjects who had traceability applied had more efficient debugging time than the students without traceability. This

TABLE IX: Results from Wilcoxcon test

Factor	W	p-value
TA vc.TNA	12	0.2849



(a) Normality test:Average Debug Time Per Mutant (TA)



(b) Data: Average Debug Time Per Mutant (TNA)

Fig. 13: Scatter plots for Average Debug Time/per mutant

TABLE X: Results from t-test: AVPM

Factor	Mean diff	Degrees of freedom	t-value	p-value
TA vc.TNA	7.6025	5.4878	-2.2513	0.03924

is the result we were expecting since traceability provides a visualization aspect that is helpful for the developer. By looking at the screen recordings we could see that the subjects frequently the traceability diagram to view which trace-links where available, in in doing so found the bugs much quicker. Its not possible to make a general conclusion at this stage regarding the results because of the small sample size, but a replication is encourages for future work within the field, in particular with a larger sample and a more controlled environment.

In the second hypothesis the results showed that the subjects with traceability had almost the same mutation kill score as the subjects without traceability. From going back to the recordings we could see that both subject groups had problems with actually killing the bugs. They didn't quite know what needed to be changed in order to kill the mutant, but often they could find where the problem was. The results may have varied if:

- There were more subjects involved.
- Clearer comments in the source code of what the functions with the mutants should do.

- The inexperienced subjects might have a harder time to correct the bugs in comparison to the more experienced participants.

At this stage no generalization can be made. There are probably more reasons behind the result, but a replication of the study would be encouraged, especially with a bigger sample size.

The third hypothesis reveled that there is a difference between the average debug time/per mutant when traceability is applied (TA) as when it is not applied (TNA), the time is shorter for TA. This can be due to multiple factors, but from reviewing the recordings we could see that the subjects with TA generally had found more bugs on a shorter time period, we believe that this is due to the help of constantly being able to refer back to the traceability diagram. Furthermore, it is because they had to run significantly less regression tests, which also improved the debugging time.

To sum up the study, no generalizations can be made from the hypotheses, further studies are needed either through replication or through comparable studies to validate the results.

VI. Discussion

To asses our discoveries, lets revisit the research questions: **RQ1.** What effect does traceability have on test case selection?

Running the hypothesis tests, showed that the debugging time was faster when traceability was applied (TA) than when it was not applied (TAN) and the results provides an answer to **RQ.1.1**How does traceability affect the developer's time to fix defects?

The screen recordings showed that the subjects with TA had to run fewer tests due to being provided with the traceability diagram which showed them which tests needed to be run. By observing the screen recordings it was possible to see that the subjects referred back to the traceability diagram 12 times on average, which goes that they did find some help from it. From the qualitative data of the surveys, subjects that used the TA stated that traceability provided them with a clearer view of the connections between the classes and the tests. When asked if they thought traceability held them back in anyway, the majority was in agreement that it did not, rather the opposite, helped them to know what tests to run. On a scale from 1 (very easy) - 5 (very difficult) in difficulty level of the debugging process, 75% of the subjects with TA rated it 3/5 and 25% as 2/5, as compared to subjects with TAN, where 50% rated it a 4/5 and 25% gave it 5/5. These results provides some insights on how traceability can be helpful for maintenance of automated unit tests in terms of debugging time. As mentioned previously, the study was conducted on quite a small sample which makes it difficult to generalize the results, however the results provides a better understanding of how traceability impacts maintenance and this study can be helpful for further research.

RQ.2 How is regression testing affected by traceability between source code and unit tests?

The TNA had to run significantly more regression tests than TA. While TA ran only 5 regression tests, TNA had to run all 14 to make sure that there were no more failing tests. The results show that traceability is helpful as it cuts down the amount of regression testing quite drastically.

The sub-research question: **RQ2.1** How does traceability influence the number of faults detected? is answered by examining the mutation kill score (MKS) through a Wilcoxcon test. The results showed that the null hypothesis could not be rejected, i.e there is no significant difference between the MKS when traceability is applied (TA) and when it is not applied (TNA). The results of the hypothesis testing would probably be different if there was a bigger sample size and a larger number of mutants. We also examined the average debugging time/per mutant (ATPM) and the hypothesis testing, using a t-test, resulted in a rejection of the null hypothesis, indicating that there is a difference in the ATPM. By examining the screen recordings we could see that both subject groups were able to find most of the mutants, but as mentioned in the summary they were having trouble with actually killing them (i.e. fixing the fault itself to make the test pass). The subjects with TA actually found and fixed more mutants in a shorter amount of time than the subjects with TNA. By taking a closer look at the survey results we discovered that TNA participants found great help in the failure trace of the JUnit tests, which is how the could find the actual mutants. This was an interesting discovery as they only had could rely on the tests to find the problems.The open questions, which is also a part of the questionnaire, showed that for debugging process traceability helped them to localize the error/problem easily, as it provided a clearer picture of connections among classes and the tests.

Some participants experienced that the unit tests generated with Evosuite were a bit difficult to understand. We believe that this is mostly due to inexperience but perhaps an area of improvement for test generation tools.

VII. Validity Threats

The criteria for validity is based on Cook and Campbell [18]. *Conclusion validity* is a threat in this study due the small samples sizes. Because of this it is hard to reveal a true pattern in the data. Even though the sample size was small, we still checked for normality in the samples before determining which kind of hypothesis testing to use. Random variation due to the having traceability applied and not applied is alleviated by to the providing the subjects with the treatments randomly. We did all of the measurements through the screen recordings, which made sure that the measurements were objective.

Internal validity is a function of variables that are systematically manipulated and observed during the study[18]. For our empirical study, we made sure to properly inform the subjects about the task, both orally and textually, before hand to mitigate the threats of having different comprehension levels. We also questioned the subjects after completion of the experiment, to see on which level they understood the task. In order to mitigate compensatory rivalry and resentful demoralization, the subjects were only provided with information about their specific treatment and we made sure that the subjects were seated separately so that they didn't know about each others treatments.

For our experiment, the internal validity might threat might also be due to difference in subjects interest and enthusiasm towards testing and debugging during the experiment. The

outer interruption (e.g noise) to one group of people is also considered a threat factor to this validity.

External validity is related to generalizing the result of the experiment beyond the study itself [18]. This has occurred as we have gathered very modest sample from the project. Therefore, to make the results more applicable, further experiments should be run by working on real complex open source projects like JEdit [8] for instance, and adding mutants through automated tools supported for it e.g Mutagen.

The subjects different experience levels, also limits the ability to generalize the results to an extent , yet they are all second and third year bachelor students, meaning that they do have experience in both testing and developing. To improve validity it would be beneficial to conduct more experiments with people having multilevel of experience.

VIII. Conclusion

The research addressed by this paper aims at investigating the impact of enabling test case selection of automatically generated unit tests through traceability. The data was collected by conducting an empirical study aimed at investigating the relationship between traceability and debugging time as well as traceability and mutation kill score. Our hypothesis testing indicated that the debugging time is shorter when traceability is applied, but also showed that there is no difference in mutation kill score when traceability is applied or not applied. When traceability is not applied, tracing the bug takes more time compared to when traceability is invoked.

Based on our overall findings, future research is needed in order to be able to make a generalization. Our goal in employing statistics, even with a small sample size, was to provide a baseline methodology for future research (e.g. future Bachelor thesis projects) needed in order to be able to make a generalization. In summary, our goal with the empirical study is to provide some insights to the problem and possible aspects to think about in the future.

As stated in our introduction, we were able to achieve technical and scientific contribution by providing i) a tool integrating two novel techniques in software development to foster automation initiative, and ii) a systematic investigation within an empirical framework to enable future evaluation of extensions of our work.

Certainly both contributions can be extended, as discussed in the next section.

IX. Future work

There were certain limitations that can be extended in future projects. Our initial idea was to run all of the regression tests on a Jenkins server. This would allow the developer to do something else in the meantime, perhaps start working with another class, while the regression tests are being executed and the traces are being made. In our implementation it takes a few minutes for the regression tests do execute, but if we had more time we would have done it on Jenkins, which is a possible strategy for future work.

[8]http://www.jedit.org/

For a smaller project it is feasible to do the debugging without traceability, even though it takes a bit longer. But in further research it could be interesting to test our approach on a much larger source code. Then it will be possible to see if the subjects can even find the mutants, let along kill them, when there is such a large amount of regression tests to be ran.

For the Capra implementation it could be good to also have trace-links between the java classes and show the inheritance or composition relationships. This with an additional view, which would provide a deeper understanding of the connections, and perhaps prevent them from introducing faults in the super classes that propagate down the subclasses.

X. Acknowledgement

We would like to express our sincere gratitude to our supervisors Francisco Gomes and Salome Maro for their continuous support throughout the project. They provided us with useful feedback and made the experience a greater one. We would also like to thank all our colleagues form Software Engineering and Management (SEM) program who took their time out to help us by taking part in our study.

References

[1] J. Cleland-Huang, O. Gotel, A. Zisman *et al.*, *Software and systems traceability*. Springer, 2012, vol. 2, no. 3.

[2] M. Bach-Sørensen and M. Malm, "Automated unit testing," *Report, Aalborg University*, 2007.

[3] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept 2015, pp. 61–70.

[4] A. Qusef, "Test-to-code traceability: Why and how?" in *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, Dec 2013, pp. 1–8.

[5] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000, pp. 73–87.

[6] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 23–32.

[7] G. M. Weinberg, "Kill that code," *Infosystems*, vol. 30, no. 8, pp. 48–49, 1983.

[8] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools (ARP/AOD) 2 Vol. Set*, 1st ed. Addison-Wesley Professional, 2009.

[9] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical software engineering*, vol. 20, no. 3, pp. 611–639, 2015.

[10] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[11] S. D. G.Karthika, "Evosuite - test suite generation and coverage criteria," *International Journal of Computer Science And Technology*, vol. 5, no. 4, p. 3, 2014.

[12] A. Marcus, X. Xie, and D. Poshyvanyk, "When and how to visualize traceability links?" in *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. ACM, 2005, pp. 56–61.

[13] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*. IEEE, 1997, pp. 264–274.

[14] J. E. Henry and J. P. Cain, "A quantitative comparison of perfective and corrective software maintenance," *Journal of Software: Evolution and Process*, vol. 9, no. 5, pp. 281–297, 1997.

[15] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[16] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE, 1992, pp. 299–308.

[17] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella, "Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, p. 5, 2015.

[18] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[19] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, 2016.

APPENDIX A

Experimental guide (TNA)

Description: In this experimental session, you will debug a simple Bank application by using tests that have been automatically generated through the Evosuite tool. The source code contains seeded faults which can be found through regression testing with JUnit.

What:
The source code provided to you will contain seeded faults. There are different kinds of seeded faults such as changing an operator from * to / for example. Or incrementing a variable that is supposed to be decremented as well as negating/removing conditionals. Your job is to run regression tests with JUnit on the code and see if you can find bugs. There is a total of 2 bugs. The screen will be recorded to allow us to trace-back and help us to analyze the results better, but each subject is anonymous.

Once you are done you will be asked to fill out a survey form.

How:

1. You will find the project in the Eclipse project explorer, select a class.
2. To run the regression tests, right click on your selected test class, testName_ESTest.java, and select Run as -> JUnit test.
3. Debug the selected class.
4. Once you are done, rerun the tests to see if it has revealed more failures. Repeat this process until all of the tests pass.
5. To measure the coverage: Right click on the test and select Coverage As -> JUnit test.

Experimental guide (TA)

Description: In this experimental session, you will debug a simple Bank application by using tests that have been automatically generated through the Evosuite tool. The source code contains seeded faults which can be found through regression testing with JUnit. To find out which tests you need to run in order to start the debugging process, you will be provided with a traceability-diagram which shows all of the tests linked with your class.

Traceability refers to the capability of relating data kept within artifacts, such as documentation, UML diagrams, source code etc. and provides a way to analyze this connection. In order to realize traceability, navigable links have to be created between the artifacts. In this case there is traceability between source code and unit tests.

What:
The source code provided to you will contain seeded faults. There are different kinds of seeded faults such as changing an operator from * to / for example. Or incrementing a variable that is supposed to be decremented as well as negating/removing conditionals. Your job is to run regression tests with JUnit and see if you can find bugs. There is a total of 2 bugs and you have 30 minutes to find them. The screen will be recorded to allow us to trace-back and help us to analyze the results better, but each subject is anonymous.

Once you are done you will be asked to fill out a survey form.

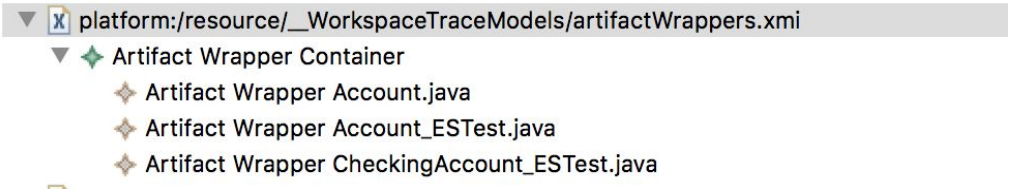
How:

1. Select a class to refactor from the project to refactor.
2. Check which tests that have traceability links with your selected class by:
   -> Select the Capra perspective in the top right corner.

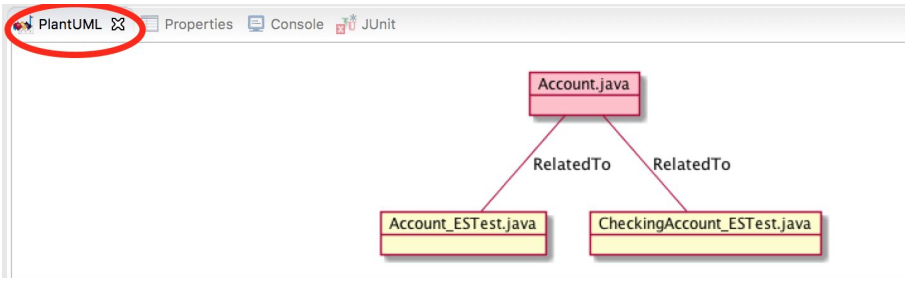
   

   -> Click on WorkspaceTraceModels and select artifactWrappers.xmi

   -> Open the dropdown menu for the Artifact Wrapper Container and select your class from the list.

   (make sure you have the plantUml tab opened, to be able to view the traces)



3. Run the tests displayed in the diagram by:
   -> Right click on the test class select “Run as -> JUnit test”.
4. Start debugging!
5. Once you are done debugging, rerun the tests and see if you have managed to fix all of the errors. If not continue the refactoring until all of the tests pass in the diagram pass.