Freie Universität Berlin

Bachelor Thesis at the Institute of Computer Sciences

Research Group Software Engineering

Improving the Quality of the Test Battery in Saros

Suzana Puscasu
Student ID: 4912120
spuscasu@zedat.fu-berlin.de

Advisor:	Franz Zieris
First Reviewer:	Prof. Dr. Lutz Prechelt
Second Reviewer:	Prof. Dr. Claudia Müller-Birn

Berlin, 7.11.2018

Abstract

Saros is a plugin for distributed collaborative programming. Within the Saros project, a framework for testing has been developed and corresponding tests using the framework have been written. This thesis summarizes two approaches to improve the quality of test battery in Saros. The existing test are analyzed and then basing on this analysis, two approaches to improve the quality of the tests and their implementations are presented. At the end, a statistical analysis on the improvement of the tests' quality is illustrated by measuring the execution time of the old tests and the tests after the implementation of the two approaches.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

7. November 2018

Suzana Puscasu

Contents

1 Introduction

Saros is an Open Source IDE plugin for distributed collaborative programming. It currently works with Eclipse, but a version for IntelliJ is being developed as well. It supports up to five participants at once, enabling this way not only distributed pair programming but also distributed party programming. It provides multiple functionalities, which are vital for these techniques. All participants have an identical copy of a project and they can all edit their files locally. Saros keeps all of the copies in sync by sending out the changes to the other participants.

The complexity of Saros has increased over the years and it was difficult to test properly. This was the main motivation for S. Szücs' diploma thesis [9]. He implemented the Saros Test Framework (abbreviated STF) that enables the comfortable writing of Unit tests for the Saros project. One just needs to understand the functionalities of the framework in order to write a test. The framework has five testers available (*ALICE,BOB,CARL,DAVE,EDNA*) . However, in the existing written tests only the first four are used. Each of them owns two robots, *superBot* and *remoteBot*. The testers can assign tasks to these robots, which send the commands to remote Saros instances. These robots are an extension of the *SWTBot*, which is the framework chosen by S. Szücs for implementing the STF.

There are 78 written tests for the Saros project. They test different functionalities and features of the project. They are organized by the functionality or feature they test. When looking for a topic for my bachelor thesis, it was brought to my attention that the run time of the Saros test battery could be improved. The goal of this thesis is to improve the quality of these tests. This thesis will focus solely on the tests themselves and not on the environment set-up that is needed for being able to test (i.e. starting the needed Eclipse instances for testing).

This thesis is structured as following: at the beginning some theoretical aspects will be presented, which are relevant for this thesis. It continues with the analysis of the existing tests in project and then with the proposed solutions for the problems, which were found in the analysis. At the end, the results of a statistical analysis on the implemented solutions and the summary of the thesis will be presented.

2 Theory of Testing

In the following chapter, some of the important theoretical aspects will be presented, aspects that I found relevant for my bachelor thesis. This chapter will start with defining the testing process and then continue with testing concepts, the characteristics of a good test case. At the end, principles for testing will be presented.

2.1 Definition of Testing

Both Myers [7] and Deutsch [4] define testing as the controlled exercise of the program code with the intention of exposing faults. The popular beliefs that testing is a process of demonstrating that faults are not present or that the purpose of testing is solely to show that a program performs its intended functions correctly are proven wrong by Meyers. A program should not be tested just to be proved that it works, but it should be assumed that the program contains faults and through testing, these faults should be found.

Myers [7] argues that understanding the true definition of testing can influence greatly the process of testing. Human beings are goal-driven and making the goal of testing clear can improve the quality of the software. If the tester's goal is to prove that the program does not have any faults, then the program will be tested in such a way that those faults will never be exposed. The human notion of success can also intervene in the process of testing. The tests' results are usually characterized as successful, if a test passed and unsuccessful, if a test failed. An unsuccessful result indicates usually something disappointing, something that is not desired. In quality assurance terminology, a test is then successful when it exposes a fault or when it eventually confirms that there are no more faults to be found. Tests are however unsuccessful when they do not manage to examine the program properly.

2.2 Testing Concepts

Bruegge [3] enumerates a series of model elements that are being used during testing. However, the definitions for *fault, mistake* and *error* will be from the IEEE-Standard 610.12-1990 [1] because there is a clearer delineation of concepts.

- A **component** is a part of the software that can be separately tested from the other components. Depending on the structure of the software, components can be objects, groups of objects, one or more subsystems.

- A **mistake** or (**error (4)** in IEEE-Standard 610.12-1990) is *human action that produce an incorrect result.* [1]

- A **fault** or (**error (2)** in IEEE-Standard 610.12-1990) is *an incorrect step, process, or data definition* [1] that might lead to a failure.

- A **failure** or (**error (3)** in IEEE-Standard 610.12-1990) is *an incorrect result* [1] or an variation from the specification.

- A **test case** is a set of inputs, contexts and expected behaviors that aims at causing failures and detecting faults.
- *A **test stub** is a partial implementation of the components on which the tested components depend.* [3]
- A **correction** is the change in the component's implementation as a result of fixing a fault. However, this can introduce new types of faults, as Brooks argues [2].

2.2.1 Test Cases

A test case is a set of inputs and expected behavior with the purpose of causing failures and in this way detecting faults. A test can have five attributes according to Bruegge [3]

Attributes	**Description**
`name`	Name of test case
`location`	Full path name of executable
`input`	Input data or commands
`oracle`	Expected test results against which output of the test is compared
`log`	Output produced by the test

Table 1: Attributes of a test case [3]

Classifications for test cases intervene mostly with different testing activities or techniques. However, Bruegge [3] chooses to classify test cases in white-box test cases and black-box test cases, depending on the context in which the test is conducted.

- **Black-box** tests usually focus on the relation between the input data and the expected behavior. Black-box tests do not address the internal structure of the components.
- **White-box** tests focus on the internal structure of the system. A white-box test does not depend on the input/output behavior and ensures that *every state in the dynamic model of the object and every interaction among the objects are tested.*[3]. The input data for white-box tests cannot be derived from the functional specifications of the component.

2.2.2 Testing Activities

In this section, different testing activities will be classified in four major categories: inspecting components, unit testing, integration testing and system testing. This will not only include the classification made by Bruegge [3] but also some other types of testing techniques and activities mentioned by Kaner[5].

Inspecting Components

The purpose of the components' inspection is finding the faults in the code. These inspections are usually conducted in formal meetings and can happen before or after unit testing. In these meetings, an overview over the code is offered by the author of the code. The author also explains what each source code statement means. The reviewers can always ask questions if they consider that the code contains faults. In practice, this method can be relatively elaborate and time-consuming because of the debates that can come from misunderstandings whether there is really a fault or not.

Unit Testing

Unit testing focuses on each object or subsystem that form the software system. There are at least three advantages that unit testing has. Firstly, the complexity of the testing itself is reduced tremendously because small units are tested and not the whole system. Secondly, testing smaller units leads to finding the faults much quicker. Lastly, more than one component can be tested simultaneously since the components are independent from each other.

There are two main categories of unit testing, which Bruegge[3] mentions:

1. **Equivalence testing** is a black-box testing method, which reduces the number of inputs, by organizing them in *equivalence classes*. The assumption, on which equivalence testing works, is that the system behaves in similar ways for all constituents of the class. This is the reason, why a specific test case is chosen for the equivalence class. This technique consists of two important steps: identification of the equivalence classes and choosing the right input data. For each equivalence class, there are at least two types of input data: a valid input, which illustrates the normal scenario, and an invalid input, which is used for exception handling. A disadvantage of equivalence testing is that input data is either valid or invalid; it is not possible to have a combination of valid and invalid data. In many cases, failures occur in software because of a combination of valid and invalid data.

2. **Path testing** is a white-box testing technique that identifies the faults in the code by testing all possible paths in the program. For identification of all paths in the code, knowledge of the source code is necessary. Myers [7] argues that in complex software, path testing cannot be very productive because of the following reasons. Firstly, the number of the unique paths in a program can be very large and therefore not all of them can be tested. Additionally, all program paths can be tested but yet, the program can still be wrong. For example, a program might be incorrect because of the missing paths. Testing all program paths will not find errors in the missing paths. Lastly, testing all program paths does not include data-sensitivity errors. *Data-sensitivity errors* are errors that only occur for a certain type of input data.

Integration Testing

After unit tests are conducted for each component, the faults are fixed and tests do not reveal any additional faults, components can be integrated into larger subsystems. The purpose of integration testing is to reveal faults that could not be detected during unit testing.This method increases the complexity of the testing and thus it is more difficult to find the location of the faults. All of the integration testing strategies can use both white-box and black-box methods, or even a combination of white-box and black-box methods, called *gray-box methods*.There are four different integration testing strategies:

1. **Big bang testing** implies testing each component in advance and then all components together in a system.
2. **Bottom-up testing** involves testing each component of the bottom layer of the system and then gradually adding components of superior layers into the system that is being tested.
3. The **top-down testing** strategy tests first the components of the top layer and integrates gradually the components of the lower layers.
4. The **sandwich testing** strategy combines the bottom-up and the top-down strategies, profiting from both strategies.

System Testing

Both unit and integration testing aim at finding faults in components and subsystems. System testing, on the other hand, ensures that after integrating all components and subsystems in a big system, the whole system functions accordingly. Bruegge[3] identifies five types:

1. **Functional testing** focuses on each function or feature being tested separately.
2. **Performance testing** focuses on finding differences between what the system was designed for and the running system. One of the most known performance testing types is *stress testing*, which aims at driving the program to failure in order to watch how the program fails. You keep increasing the size rate of the input until either the program finally fails, or you become convinced that further increases will not lead to a failure.
3. During the **pilot testing** phase, a selected group of users receives the software and they test it without being given any instructions on how to test it.
4. In the **acceptance testing** tests are performed by the customer in order to see if the requirements from the project agreement are met.
5. An **installation test** checks to see if the installed system meets all the requirements.

Kaner[5] mentions two more strategies that could be included in the system testing category:

1. **Risk-based testing** focuses on collecting as many scenarios as possible, in which the program might fail and designing tests to check, whether the program will fail or not.

2. **Exploratory testing** allows to perform any testing to the extent that the tester actively controls the design of the tests. It is advised to use this strategy when the tester knows very well the system to be tested.

2.3 Characteristics of a Good Test Case

There is no clear definition of a good test case. A test case can be good from one perspective and less good from another one. However, Kaner [5] identifies some of the characteristics as follows:

- A good test case is **powerful**. This means, that the test case is prone to revealing faults.

- Good **credibility** of a test case means that the test case is relevant for the usage of the program. The opposite of credible test cases is the *edge cases*, which occur only under special circumstances or for which the probability of occurring is very low.

- A good test case should be **easy to evaluate**. This implies that the test case specification is clear for the tester.

- The **complexity** of a test case should be as low as possible especially for the beginning phases of testing. Afterwards, the complexity can gradually increase if the system gets more stable.

- A good test should **not** be **redundant**. A redundant test case contains scenarios that are tested in other test cases.

2.4 Principles

In this section, principles that guide the testing process of a software will be presented.

1. Myers [7] argues that the probability of the existence of more faults in a section of a program is proportional to the number of faults already found in the section. Some sections of a program are more prone to containing faults than others, because of the complexity of the section. Faults tend to come in clusters. This principle gives us an insight in the frequency with which some parts of the program should be tested or in the amount of work that should be invested in testing a specific part of the program.

2. As previously mentioned at equivalence testing, it is highly important that test cases receive valid input data as well as invalid. Sometimes, failures in the system behavior might occur in cases when the input data is invalid.

3. Myers [7] states that there must be a clear definition of the expected output or result when writing a test case. If the expected result is not clearly defined, the chances of interpreting an erroneous result as a correct result are relatively high. This is definitely influenced by the human psychology; human beings will hope unconsciously to see the correct result, and therefore overlook the possible invalid result.

4. Sommerville [8] mentions that unit testing should be performed by the software developer, whereas system testing by an independent testing team. This is the case for software of high complexity, where the system testing can be very elaborate.

5. Test cases should be reused and there should be a battery of tests that should be performed after changes have been made in the software. This principle can be classified also as a testing strategy, *regression testing*, as Kaner [6] does.

6. The test case specification should delimit between the *set-up* steps for the test case, the test case itself and the *clean-up* after the test has been executed. By *set-up* steps it is meant all the preparatory steps that enable to execute the test and by *clean-up* all the necessary operations to bring back the system in a clean state, so that the next test can be executed. After the set-up steps are made, we need to make sure that the system functions normally, that there are no abnormalities. By having a clear delimitation in the test case specification, the test case result can be interpreted easier, since the possible faults from the set-up are eliminated.

2.5 Summary

- Testing is a destructive process rather than constructive, and its goal is to find faults.
- There are different types of testing and in order to find as many faults in software as possible, it is advised to have a combination of testing techniques.
- There should be a clear definition of the input and output data, in order to be able to evaluate it easily, whether the program failed or passed the test.
- A test case should not be redundant nor too complex and should be run also with invalid or unexpected input.

3 Saros Test Battery Analysis

In this chapter the focus will be on the analysis of the existing tests in the Saros project. It will start with illustrating some of the previously mentioned test types on the Saros test battery, then continue with the criteria for the tests' analysis and it ends with the way the analysis was conducted.

3.1 Test Types in Saros

Saros test cases combine multiple testing strategies. We will focus on the black-box testing types as follows:

- *Equivalence testing*: all test cases are divided into equivalence classes that represent either a special feature i.e. *follow mode* or different basic functionalities like creating folders new files in the session.
- *Functional testing*: each specific function is tested separately from the other functions. Examples for this testing types are the test cases for chat view.
- *Performance testing* especially *stress testing*: the stress test cases usually derive from a normal test case, which tests if Saros behaves accordingly in a rather simple scenario. An example for this is the test for adding multiple files in the session.
- *Risk-based testing*: there is at least one test case that takes some risks and the outcome could either be positive or negative. `EditFileThatIsNotOpenOnRemoteSide` checks what problems the editing of a file for one user and then saving it might cause if the second user's editor is closed.
- *Exploratory testing*: there are test cases that were written as result of the exploratory testing. For example, the test `UnintentionalCursorMove` was written because a programmer tested freely. I came to this conclusion because of the comment written in the test case itself that referenced to a bug report. This test case checks if the follower's focus is stolen when the *followee* [1] switches from one file to another.
- *Regression testing*: the tests battery is derived from the planned tests, from exploratory tests that revealed faults in Saros. Test cases are reused and tests are re-run. One of the huge advantages is that a specific standard is kept before officially releasing the plugin. It ensures also that all the changes that have been made, do not destroy some other important functionalities.

3.2 Focus Points of the Analysis

In the next section I will present the points I focused on when I analyzed the Saros test battery.

[1] A *followee* is a user, who is followed by another user that previously activated the follow mode for the followee. When editing a file, the follower can track the followee's cursor.

1. **Execution time** of the tests. It is important for the tests to be executed in a short period of time so that tests can be run more often.
2. **Similarities of the set-up and clean-up methods in test cases** which might be time consuming because they are executed for each test.
3. **Important test cases**, which other test cases derive from.

3.3 Execution of the Analysis

3.3.1 Execution Time

For this focus point I started by analyzing the most used methods in the test cases. They come from two main classes:

- `Util.java` implements different Saros activities that a user does, for example setting up a Saros session, activating follow mode. There are different methods for setting up of a Saros session:
  - `setUpSessionWithProjectAndFile`
  - `setUpSessionWithJavaProjectAndClass`
  - `setUpSessionWithJavaProjects`
  - `createProjectWithEmptyFile`.

  However, all these methods rely on two other methods:

  1. `buildSessionSequentially` establishes a Saros session between an *inviter* [2] and several *invitees*[3]. Every *invitee* is invited one by one.
  2. `buildSessionConcurrently` also establishes a Saros session between an *inviter* and *invitees*. However, all invitees are invited simultaneously.
- `Internal.java` implements activities that a user does in Eclipse: creating a Java project, or creating a file. The most used methods in tests are: `createFolder`, `createFile` and `createJavaProject`. As the names suggest, they implement creating a folder in a project, a file in a given project and a Java project.

Time Measurements

For the time measurements I used a stopwatch from `org.apache.commons.lang.time` that was reset and started before and stopped after each method that was previously mentioned. The time difference in milliseconds was then written in a file.

```
stopwatch.reset();
stopwatch.start();
stopwatch.stop();
stopwatch.getTime();
```

[2]The *inviter* is the user that initiate the session, the host of the session.
[3]The *invitees* are the participants that receive the invitations from the *inviter*.

Results' Visualization

The following box plots show the execution time in milliseconds for the five important methods: `createFile`, `createFolder`, `createJavaProject`, `buildSessionConcurrently`, and `buildSessionSequentially`. In the *Appendix* one can find the box plot statistics for each method (Table 4(a), Table 4(b) Table 4(c), Table 4(d), Table 4(e)).

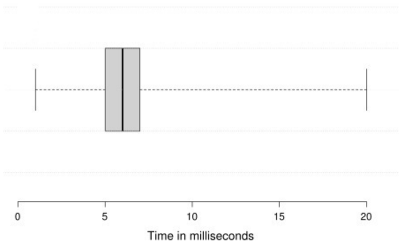
In Figure 1(a), in 50% of the cases, the file creating method varies from 5 milliseconds to 7 milliseconds. The time for the creation of the file varies depending on the size of the file to be created. The minimum for the file creation is 1 millisecond and it corresponds to creating a small-sized file. In Figure 1(b), one can see that in 75% of the cases, the time for the folder creation varies between 1 and 7 milliseconds and in 50% of the cases only between 1 and 3 milliseconds. In Figure 1(c), the execution time of `createJavaProject`. This varies between 65 and 120 milliseconds in 50% of the cases. None of these functions represent real problems for the execution time of the whole tests battery.

The main causes for the long execution time of the tests battery are the two methods that build up a Saros session. In Figure 1(d) one can see that the execution time for the concurrent building up of a Saros session varies between 1837 and 2128 milliseconds in 50% of the cases. The variation between the minimum (1511 ms) and the maximum (3638 ms) execution time could be explained by the number of the participants in the session. In most of the test cases a session between two participants is normally required but there are also test cases that require four participants. In Figure 1(e) the execution time for `buildSessionSequentially` is presented. In comparison to `buildSessionConcurrently` the execution time is greater because each participants is invited by the host one after another. It varies from 10 seconds to aproximately 12 seconds in 50% of the cases.

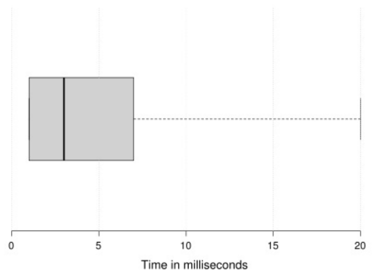
3.3.2 Similarities of the Set-up and Tear-down Methods in Test Cases

After analyzing the execution time of the methods and analyzing the code for each test case, it came to my attention that for each test case, the Saros environment was built up from scratch every time and was destroyed completely after each test's execution. Before each test, the method `select(AbstractTester tester, AbstractTester... testers)` is called. This method is composed of three other methods:

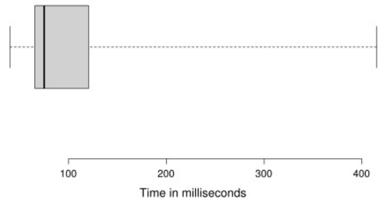
1. `initTesters` that registers the given testers to be participants of the current test.
2. `setUpWorkbench` brings the Eclipse workbench to an initial state before beginning the test, by:
    - activating the Saros instance workbench;
    - closing all opened pop-up windows and all unnecessary views;
    - closing all opened editors;
    - deleting all existing projects;
    - closing the Eclipse Welcome View, if it is open;



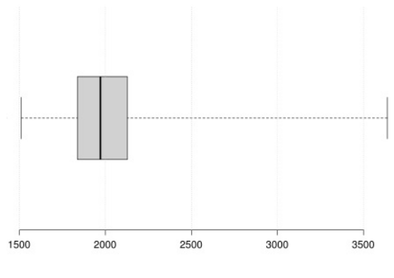
(a) `createFile`



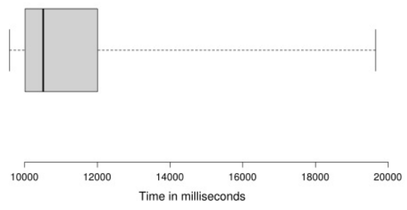
(b) `createFolder`



(c) `createJavaProject`



(d) `buildSessionConcurrently`



(e) `buildSessionSequentially`

Figure 1: Execution time of different methods

- opening the default perspective.

3. `setUpSaros` that brings Saros to an initial state before beginning the test, by:
   - disabling the *automaticReminder*;[4]
   - opening Saros views;
   - ensuring that all testers are available in each of the testers' contact list;
   - connecting the current testers among themselves.

After which each test is executed, the method `tearDownSaros` is called. This method tries to reset Saros by:

- resetting all contact names;
- disconnecting from current session;
- deleting the contents of the workspace for each tester.

For each test, the Saros session is set up anew. As mentioned before, setting up the session can be done with different methods, which rely on only two methods `buildSessionConcurrently` and `buildSessionSequentially`. By setting up the session for each test, the execution time for the whole battery of tests definitely increases.

3.3.3 Important Test Cases

For finding important test case, which other test cases derive from, I read all test cases and tried to figure out which test cases derive from which test cases and what functionalities they test. I identified different levels of derivation:

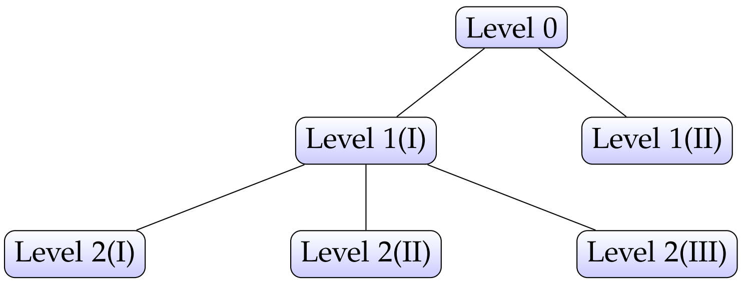


Figure 2: Derivations in test battery

1. *Level 0* represents test cases that are not derived from any other test cases. An example for this category is the test case `HandleContacts`, which checks if the removal of a contact from the contact list functions properly.
2. *Level 1(I)* represents test cases, which derive from the concurrent building of a session. An example for this category is the test case `AddMultipleFiles`. This category contains the most test cases.

[4]The *automaticReminder* disables feedback-sending via GUI

3. *Level 1(II)* represents test cases, which derive from the sequential building of a session. An example for this category is the test case `Share3UsersSequentially`.
4. *Level 2(I)* represents test cases, which derive from the `EditDuringInvitation` test. The only example for this category is the `EditDuringInvitationStressTest`.
5. *Level 2(II)* represents test cases, which derive from the `ConcurrentEditing` test. Examples for this category are `ConcurrentEditingInsert100Characters` and `ConcurrentEditingWith3UsersTest`.
6. *Level 2(III)* represents test cases, which derive from the `SimpleFollowModeII` test. All other test cases from the `follow mode` category derive from this test.

3.4 Conclusions of the Analysis

After I analyzed the Saros test cases' code, the main conclusion that could be drawn was that *the quality of the test battery could be improved* by reducing its execution time. The analysis led to two approaches for improving the quality of the tests:

1. Tests, which derive from other tests, do not need to be executed if the parent test did not pass.
2. Tests can be grouped according to their necessary environment. The environment will be set only once and torn down after the execution of the last test in the group.

4 Proposed Approaches and Their Implementations

In this chapter, two approaches for the improving of the Saros tests' quality will be presented. Firstly, the concepts that these approaches rely on will be explained, then the motivation for choosing the approaches and the implementation for each approach was realized. At the end, the status of the integration of the code in the Saros project will be presented.

4.1 Conditional Testing

4.1.1 Explanation of the Concept

The premises for conditional testing are that the software has some core functionalities, without which the software will not work properly.

By *conditional testing* it is meant identifying the main test cases first, which do no derive from other test cases and if the tests pass, running the rest of the tests that rely on the identified test cases. If the tests fail, then the rest of tests, which depend on the main tests will be skipped.

Two steps were identified for the realization of the conditional testing. These are:

1. The main test cases need to be singled out. There is usually one main test case; the other existing test cases derive from the main one.

2. There could be other test cases that derive from the main one(s). In this case, we talk about different *levels of conditional testing*. The structure of these levels can be illustrated in a tree, where the nodes represents different test cases, from which others derive from.

4.1.2 Motivation

Saros is a rather stable plugin that already went through the initial stages of development when the software contains a lot of faults. After analyzing the Saros plugin (see Important test cases from *Saros Test Battery Analysis*), it came to my attention that conditional testing could be a way of saving up time. Figure 2 from the previous chapter illustrates different levels of functionalities identified by analysing the test cases.

There are several advantages of conditional testing in Saros.

- It saves up time by not having to execute other tests if the main functionalities tests fail. For example, there is no point in running `AddMultipleFiles` test if the test for building the Saros session fails.

- It gives an organized structure of the tests' execution. When the tests are skipped, this means that an important test did not pass. In this way, it is easier for the whole Saros team to asses where the fault might me.

4.1.3 Implementation

For each level described in Figure 2 a *Test Watcher* was implemented as a *JUnit Rule* that stores in an `ArrayList` the results of test case, which the others depend on. For example, the test case `SimpleSessionConcurrently` has as JUnit rule `STFTestWatcher-LevelONEi` (see Figure 7 from *Appendix*). The implemented test watcher has the following methods:

- `succeeded` is called when the test succeeds.
- `failed` is called when the test fails.
- `checkIfAllSucceeded` checks if all tests that have this test watcher as JUnit Rule this rule succeeded.

In the `StfTestCase` there is a JUnit watchman implemented, which became deprecated. This is the reason why a new test watcher was implemented, basically with the methods from the already implemented watchman that enables logging the results of the test cases and also capturing screenshots if the tests fail. The method `skipped(final AssumptionViolatedException e, final Description description)` was overridden to be able to log the reason why the test was skipped. The difference between the two test watchers is that one stores the results for each level of tests and the other one logs the result of the test.

Depending on the different levels of the categories, combinations of the lastly mentioned methods of all STF test watchers are called in the `StfTestCase`, a class which all test cases extend from, in order to check whether a test should run or not. The methods defined in the `STFTestCase` are then called in `@BeforeClass` of each test. For example:

```
Assume.assumeTrue("EditDuringInvitation failed",
    checkIfLevelONEiandTWOiSucceeded());
```

Figure 3: @BeforeClass of the `EditDuringInvitationStressTest`

In this case the test is skipped [5]and the corresponding message is logged with the help of the test watcher implemented in the `STFTestCase`.

Example

In order to illustrate how conditional testing in Saros works, the decision tree for the execution of `FollowModeDisabledInNewSession` is presented. This test checks if follow mode is disabled after initially being enabled and then the *followee* leaving the session. The test belongs to `LevelTWOiii` category.

[5]A violated `Assumption` does not lead to failures.

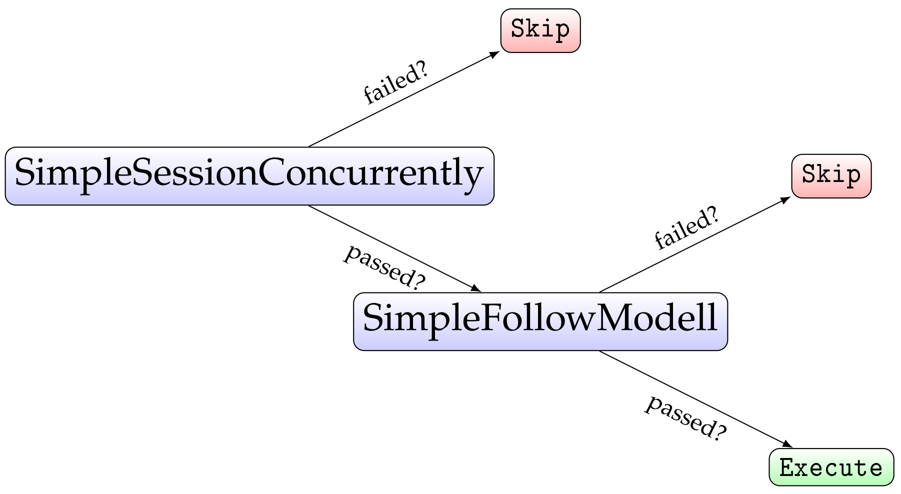


Figure 4: Decision tree for FollowModeDisabledInNewSession

Difficulties in implementation

- The main problem was that there is no direct way for dynamical conditional testing in JUnit. There is a static way to do it by putting the label `@Ignore("reason for ignoring")`. The whole implementation is a workaround for dynamical ignoring in JUnit.
- An non-existent solution for dynamical ignoring led to another problem: not being able to show the reason why the test was skipped. In order to have an overview over the whole tests' execution, it is needed to know why some of the tests were skipped, in the case that they are skipped. For this issue the `skipped` method was implemented in the `StfTestCase`.

4.2 Set-up Equivalence Classes

4.2.1 Explanation of the Concept

A *set- up equivalence class* represents a set-up environment that is suitable for more than one test case. In other words, the set-up methods are run only once for each group of test cases that need the same set-up environment. Multiple steps have been identified in the implementation of this concept:

1. Analyzing the test cases and finding all similarities among the set-up methods of all test cases.
2. Organizing the test cases in categories. One category corresponds to the execution of only one set-up method.
3. Simplifying the set-up as much as possible. For example, if there is more than one possible way of doing the same thing, choose the faster one.

One important aspect that should not be forgotten, is that the tear-down methods should also be adapted to the set-up equivalence classes. This means, they should not clean up completely after each test case. However, they should bring back the system to the initial state after which the set-up was done.

4.2.2 Motivation

After analyzing the execution time of some important test methods, (see *Execution time* from *Saros Test Battery Analysis*) and all similarities of the set-up and tear-down in test cases (see *Similarities of the set-up and tear-down methods in test cases*), the following conclusion could be drawn regarding the set-up equivalence classes:

- A Saros session is always built anew for each test case. This increases the run time of each test case with at least two seconds, depending on the method used for the session build.
- If a Java project is needed for the test case, it is created every time for each test case.
- There are four main set-up equivalence classes that could be identified by taking into account the number of the participants in the test case.
  1. **ALICE**: all test cases that need only one participant
  2. **ALICE&BOB**: all test cases that two participants
  3. **ALICE&BOB&CARL**: all test cases that three participants
  4. **ALICE&BOB&CARL&DAVE**: all test cases that four participants
- There are more possibilities to build a Saros session (`buildSessionConcurrently` and `buildSessionSequentially`). For the implementation of this approach `buildSessionConcurrently` will be chosen, since the building of the session is much faster (only for three and four participants this aspect is relevant because the *invitees* are invited one by one).
- The session can be built with an empty Java project and afterwards the needed files for the test case will be added. This way, there will be a clean starting point for each test and after its execution everything that the project contains will be deleted so that there will be clean starting point for the next test.

4.2.3 Implementation

There were multiple steps in the implementation of this approach. Firstly, I will present them and at the end I will give an example for the execution of a tests group, whose set-up represents one set-up equivalence class.

Structure of a tests group

For each set-up equivalence class the structure of the group should be clearly define. This is important because the *main set-up methods* will be run at the beginning of the whole group's execution and the *main tear-down methods* will be run after the tests' execution. By *main set-up methods* and *main tear-down methods* are meant the methods that set up the environment to be able to run the tests and the methods that tear down completely the environment. In Saros environment the *main set-up method* would be building the Saros session and the *main tear-down method*, disconnecting the session. In each tests group there are three categories of tests:

- In the *first test of the group*, the environment is set up for the rest of the group. This test needs to be run first.
- The *last test of the group* should disconnect the existing session. This test has to be executed last.
- The *other tests* should neither set up the environment nor tear it down.

Rewriting the Test Cases

After structuring the tests groups, the next step was to modify the existing set-up and tear-down methods. Table 2 offers an overview over set-up and tear-down methods each tests category needs. Mostly, it is a combination of the existing methods in the `StfTestCase`.

Only deleting all files from the existing project had to be implemented, by deleting the source folder of the project, or other folders that were used in the test.

```
public void deleteFolder(String projectName, String folderName
    )
        throws RemoteException, CoreException {
        IProject project = ResourcesPlugin.getWorkspace().
            getRoot()
             .getProject(projectName);
        IFolder src = project.getFolder(folderName);
        src.delete(true, false, null);

    }
```

Figure 5: `deleteFolder` in the `InternalImpl.java` in tear-down method

The tests themselves were rewritten afterwards according to the group they belong to. The changes are the following:

- In no test methods a Saros session is built anymore because the building of the session is done in the set-up methods.
- Files and folders are added to the source folder according to the test case, since each test has as starting point an empty Java project.
- In every *@BeforeClass* method, it is checked if the session exists, if it does not exist, then the session is built. As a result of an error or a failed test, the set-up could be destroyed. This should not influence the next test case.

However, there are test cases that need special set-up and could not be grouped together with the other test cases that needed the same number of testers.

Tests category	set-up	tear-down
first test	• initialize testers; • close all opened pop-up windows; • close all opened editors; • close welcome view, if it is open; • open default perspective; • delete all existing projects; • close unnecessary views; • set up Saros; • build Saros session concurrently.	• delete all files from the project; • close all non-workbench windows; • save and close all open editors; • reset the active perspective; • switch to the default perspective for the workbench; • reset the default perspective for the workbench.
last test	• initialize testers; • close all opened pop-up windows; • close all opened editors; • close welcome view, if it is open; • open default perspective; • close unnecessary views; • set up Saros.	• reset all contacts' names; • disconnect from current session; • delete contents of the workspace.
other tests	• initialize testers; • close all opened pop-up windows; • close all opened editors; • close welcome view, if it is open; • open default perspective; • close unnecessary views; • set up Saros.	• delete all files from the project; • close all non-workbench windows; • save and close all open editors; • reset the active perspective; • switch to the default perspective for the workbench; • reset the default perspective for the workbench.

Table 2: Set-up and tear-down methods for tests categories

Organizing the Tests in a Test Suite

The order of the test cases is very important, since the set-up of a test depends on another test's set-up. Additionally, organizing the resulted tests of this implementation did not have to intervene with the resulted tests of *conditional testing's* implementation. The resulted tests from the *Set-up equivalence classes* had definitely priority.

Since the set-up of the environment for tests is not properly tested, there was the need to write two additional test cases that checked if a Saros session works properly. One test checks if the concurrent session build between two users works properly, and the other if the sequential session build between two users functions. These two test cases represent the conditions for all test cases from *Level 1(i)* and *Level 1(ii)* explained in Important test cases from chapter 2.

Example of a Test Group

In the following table, one can see how and in which order tests from the group **ALICE&BOB** are executed. This groups represents all tests in which there were only two participants and need the same set-up. The first entry in table is the first executed test case class in this group and the last entry the last one. Apart from the following test cases, the are other test cases that need two testers *ALICE* and *BOB* but need special set-up. For example, the test case `HandleContacts` does not need a Saros session or the test case `InviteWithDifferentVersions`, as the name suggests, tests if the invitation process works if the participants have different versions of Saros.

Test name	Remarks
`SimepleSessionConcurrently`	• condition for Level 1(i) test cases; • checks if the set-up and tear-down works properly.
`SimepleSessionSequentially`	condition for all Level 1(ii) test cases
`ChatViewFunctions`	set-up for the whole group
`AddMultipleFiles`	
`ModifyFileWithoutEditor`	
`RecoveryWhileTyping`	
`ConcurrentEditing`	
`EditDifferentFiles`	
`Editing3Projects`	
`EditWithReadAccessOnly`	
`FolderOperations`	
`SimpleFollowModeII`	condition for the next two test cases
`FollowModeDisabledInNewSession`	
`FollowMode`	
`RefactorInFollowMode`	
`SimpleFollowModeI`	
`WriteAccessChangeAndImmediateWrite`	

Table 3: ALICE&BOB group

4.3 Implementation Status at the Moment of Submission

At the moment of submission of my bachelor thesis, the code I wrote was yet not integrated in the whole project on GitHub, just locally on my computer.

5 Analysis of the Implemented Approaches

This chapter will start with presenting a statistical analysis for the implemented approaches. Afterwards, disadvantages of the implementation will be presented.

5.1 Statistical Analysis

This analysis will focus only on the second implemented approach, *set-up equivalence classes*, since this approach led to the reduction of the run time of all tests. The run time of tests before the implementation and after the implementation of the *set-up equivalence classes* approach will be measured and then compared.

There are some factors to take into account before summarizing this statistical analysis. Firstly, the execution times might vary considerably when the tests are run on a different computer. Another factor that could influence greatly the run time of tests is the CPU usage. I kept the CPU usage to a minimum; this means that no unnecessary processes ran in the background while measuring the execution time. Moreover, when running the test battery, not all of tests passed every time. Each execution of the tests battery contains maximum three tests that failed or that threw errors when they were run; they represent 10% of all tests. Lastly, the number of the executions for each category is only 20; for a quantitative analysis it is a rather small number. Considering this, this analysis offers just a rough approximation of the improvement that the implementation of this approach brought.

5.1.1 Analyzed Data

Since the proposed approach focuses on the set-up for each test, not only the run time of the tests themselves will be measured but also the time for the set-up and tear-down of the environment.

The data for this analysis does not represent the entire test battery for the Saros project, but all test cases that need only two testers, *ALICE* and *BOB*. The main reason why I chose only this group is the time constraint; it takes long time to run all of the tests multiple times. The reasons why this group is representative for the purposes of this analysis are the following:

- This group represents 47% of all tests.
- The other groups use mainly the same methods as these tests. The only difference is the set-up for each test that takes longer, since there are more participants.
- This group contains also tests, whose set-up was not changed due to the fact that their set-up could not be grouped into equivalence classes. Only 45% of the tests from this group were modified. Thus, this analysis gives an overview over the whole test battery, not only over the modified tests.

5.1.2 Results' Visualization

For each category (old tests and new tests) a box plot will be presented. It provides an overview for each category, how the run time for each execution of the tests varies. Statistics for both box plots can be found in the *Appendix*, Table 5.

In Figure 6(a), one can see that the run time of the old tests varies between approximately between 793 and 1188 seconds. In 50% of the cases, the run time varies between 843 and 1038 seconds. In Figure 6(b), one can can see that the run time of the new tests varies between approximately between 477 and 540 seconds. In 50% of the cases, the run time varies between 492 and 508 seconds.

In Figure 6(c), the median time of the old tests and new tests are presented in order to get an overview over the improvement of the run time. In addition, the standard error is also represented on the chart. The standard error is 25660,811 milliseconds for old tests and 3593,871 milliseconds for the new tests. It shows how the run time fluctuates. The definitely smaller standard error for the new tests indicates the results gathered in the analysis are more accurate than in the case of the old tests. In order to get a smaller standard error for the old tests, I could have gathered more data points, but because of the previously mentioned reasons I did not.

5.1.3 Conclusions of the Statistical Analysis

The visualization of the results led to the following conclusions:

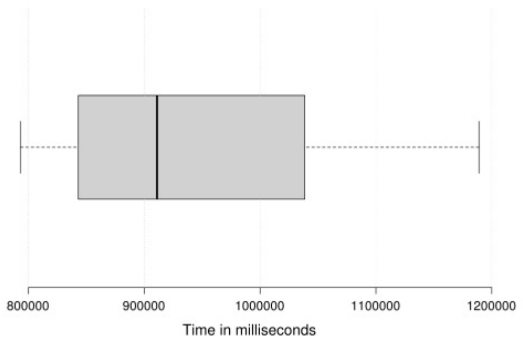
- The run time of the new tests is more stable; it fluctuates only in an interval of 62 seconds, while the run time of the old tests varies in an interval of 395 seconds.

- The run time has definitively improved, it takes approximately 45% less time to run the tests for this category.

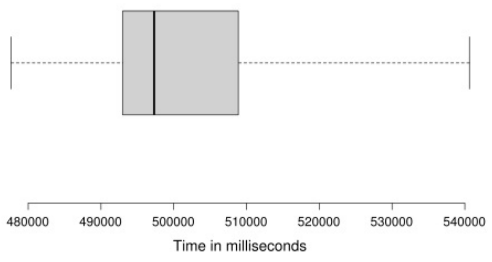
5.2 Disadvantages of the Implemented Approaches

Considering the previous analysis, the main advantage of the implemented approaches is that the overall run time of the test battery has improved. A compromise was made between the independence of the test cases and the execution time. The test cases themselves do not depend on each other, but the set-up of one test case depends on another one's. Considering this, there are at least two issues that I could identify:

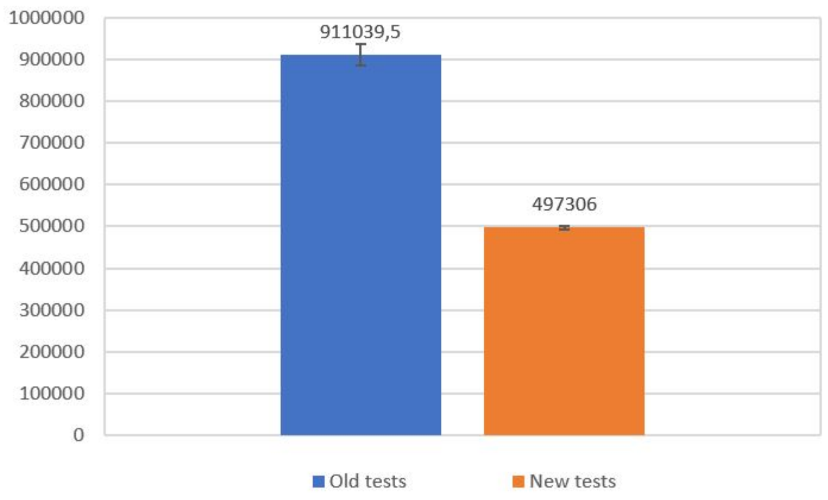
- If just a part of the tests need to be executed, then at least the test that tears down the environment has to be executed as well. At the moment of submission in the bachelor thesis, before the execution of each test it is checked, whether the Saros session exists, if not, it is rebuilt. Therefore, the only problem would be that after executing the wanted tests, one should either change the tear-down method of the last test executed (`tearDownSaros` to `tearDownSarosLast`) or execute extra the last test from the category the executed tests belong to.



(a) Run time of old tests



(b) Run time of new tests



(c) Median run time of old tests and new tests in milliseconds

Figure 6: Statistical Analysis

- Since the tests should be executed in a precise order, when reorganizing the tests battery, one should pay attention to the order in which he or she reorganizes it.

6 Conclusion

This bachelor thesis presents two of the ways the quality of the Saros tests could be improved. It starts with some theoretical aspects that are important for analyzing the tests themselves: what software testing implies, testing concepts and the types of testing activities and at the end it presents some of the characteristics for a good test case and some principles I found important for conducting the analysis.

It continues with the Saros test battery analysis, the starting point for the whole thesis. With the help of this analysis, I found what could be improved and got insights in how I could improve the Saros tests. The execution time of the most used methods was measured and analyzed. Afterwards, the dependencies among the tests were investigated and displayed in an *derivations tree*.

Chapter 4 focuses on the two proposed approaches for improving the quality of the tests. The first one is *conditional testing* that reduces the execution time in case the most representative test for a core functionality fails. The other tests that test different feature of the core functionality are not run anymore. It is described how I implemented this approach and an example for this is given. The second part of this chapter focuses on the implementation of the *set-up equivalence classes*. Firstly, it is explained what this concept means and then how it was implemented in the Saros project.
The last chapter presents different statistics on how the run time of the tests has been improved.

Bibliography

[1] IEEE Std 610.12-1990IEEE Standard Glossary of Software Engineering Terminology, 1990.

[2] Frederick P. Brooks Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] Bernd Bruegge and Allen A. Dutoit. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[4] Michael S. Deutsch. *Software Verification and Validation*. Prentice-Hall Inc., Englewood Cliffs, 1982.

[5] Cem Kaner. What is a good test case? *Software Testing Analysis & Review Conference (STAR) East USA*, 2003.

[6] Cem Kaner and Hung Quog Nguyen Jack Falk. *Testing Computer Software*. Van Nostrand Reinhold, New York, 1993.

[7] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

[8] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[9] Sandor Szücs. Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung. Diploma Thesis, Freie Universität Berlin, 2010.

Appendix

```
public class STFTestWatcherLevelONEi extends TestWatcher {

    public static ArrayList<String> list = new ArrayList<String>();

    @Override
    public void succeeded(Description description) {

        list.add("Succeeded");

    }

    @Override
    public void failed(Throwable e, Description description) {
        list.add("Failed");

    }

    public static ArrayList<String> getList() {
        return list;
    }

    public static boolean checkIfAllSucceeded() {
        for (int i = 0; i < list.size(); i++) {
            if (list.get(i).equals("Failed")) {
                return false;
            }
        }
        return true;
    }

}
```

Figure 7: STFTestWatcherLevelONEi

Graph statistics	**Value**
Maximum/Upper whisker	20 ms
3rd quartile	7 ms
Median	6 ms
1st quartile	5 ms
Minimum/Lower whisker	1 ms
Number of data points	2353

(a) `createFile`

Graph statistics	**Value**
Maximum/Upper whisker	20 ms
3rd quartile	7 ms
Median	3 ms
1st quartile	1 ms
Minimum/Lower whisker	1 ms
Number of data points	197

(b) `createFolder`

Graph statistics	**Value**
Maximum/Upper whisker	157 ms
3rd quartile	120.5 ms
Median	75 ms
1st quartile	65.5 ms
Minimum/Lower whisker	40 ms
Number of data points	79

(c) `createJavaProject`

Graph statistics	**Value**
Maximum/Upper whisker	3638 ms
3rd quartile	2128 ms
Median	1971 ms
1st quartile	1837 ms
Minimum/Lower whisker	1511 ms
Number of data points	169

(d) `buildSessionConcurrently`

Graph statistics	**Value**
Maximum/Upper whisker	19651 ms
3rd quartile	12003 ms
Median	10509 ms
1st quartile	10010 ms
Minimum/Lower whisker	9583 ms
Number of data points	27

(e) `buildSessionSequentially`

Table 4: Box plots statistics

Graph statistics	**Value**
Maximum/Upper whisker	1188914 ms
3rd quartile	1038598 ms
Median	911039 ms
1st quartile	843103 ms
Minimum/Lower whisker	793209 ms
Number of data points	20

(a) Old tests

Graph statistics	**Value**
Maximum/Upper whisker	540632 ms
3rd quartile	508871 ms
Median	497306 ms
1st quartile	492975 ms
Minimum/Lower whisker	477639 ms
Number of data points	20

(b) New tests

Table 5: Box plots statistics for old tests and new tests

List of Figures

List of Tables