UNIVERSITY OF TARTU

Institute of Computer Science

Computer Science Curriculum

Kristine Leetberg

Lab Package: Static Code Analysis

Bachelor's thesis (9 ECTS)

Supervisor: Dietmar Pfahl

Tartu 2016

Lab Package: Static Code Analysis

Abstract:

The main goal of this thesis is to enhance the lab materials about static code analysis used in the course "Software Testing (MTAT.03.159)" in the University of Tartu. The motivation for the changes is explained and the new materials are introduced in this work. The materials were applied in the course and received positive feedback. Students' feedback given after the execution of the lab is analyzed with suggestions for future improvements given.

Keywords:

Software testing, static code analysis, lab package

CERCS: P170

Praktikumimaterjal: staatiline koodianalüüs

Lühikokkuvõte:

Antud bakalaureusetöö eesmärgiks on luua uus versioon staatilist koodianalüüsi tutvustavast praktikumimaterjalist, mida kasutatakse Tartu Ülikoolis aines "Tarkvara Testimine (MTAT.03.159)". Töös kirjeldatakse nii peamiseid põhjuseid muutusteks kui ka töö käigus valminud uuenenud materjale. Loodud materjale rakendati eelnimetatud aines ning neile antud tagasiside oli positiivne. Töö lõpeb tudengite antud tagasiside analüüsiga ning lisatud on ka soovitusi edasisteks parandusteks.

Võtmesõnad:

Tarkvara testimine, staatiline koodianalüüs, praktikumimaterjal

CERCS: P170

Contents

1 Introduction

Testing is an essential part of the software development lifecycle. In order to ensure high software quality, different types of testing must be used. The course "Software Testing (MTAT.03.159)" taught at the University of Tartu introduces many types of testing through lectures and lab sessions. The starting point for this thesis is the thought that by ensuring the materials used in the course are as relevant and appropriate as possible, it will help the students improve the quality of their upcoming projects and work as IT specialists.

Static code analysis is a convenient yet highly effective testing technique for discovering potential bugs, especially in cases where dynamic testing might fail to find them [1]. Since it is important to introduce the method to students, a lab focusing on static code analysis has been in place in the "Software Testing" course. However, the feedback received from the students about the materials in the previous years has not been the best and therefore the objective of this work is to enhance the lab materials on static code analysis. The thesis gives an overview of the enhanced and newly created lab materials, how they differ from the existing materials, how they were used in practice, and how students perceived them.

The course "Software Testing (MTAT.03.159)" consists of six lab sessions, each introducing a different testing technique. Every lab introduces a new tool or technique which is then used to find faults in a given piece of software or software documentation. The feedback given to the lab about static code analysis was fairly negative with students mostly claiming that the system under test was too large to be comprehended in the given timeframe, the workload was too big and the materials were not explanatory enough. Therefore, the enhancements done within this thesis were targeted to eliminate those problems and help students get familiar with static code analysis more easily.

In order to improve the materials used in the previous years, several changes were made. To begin with, a completely new system under test was written specifically for this lab with carefully chosen bugs seeded into it. Also, a new motivational task was added to the beginning of the lab to raise the awareness of the need for and use of static code analysis tools. Furthermore, the homework tasks and the grading scheme were adjusted to fit with the defined workload of the lab (360 min per student).

The thesis consists of several parts. The second Chapter gives an overview of materials previously used in the "Software Testing MTAT.03.159" course, focusing on the lab about static code analysis. Also, principles followed when constructing the lab package for the materials are explained. The third Chapter focuses on the finished lab materials, explaining the use of each individual task and pointing out how they were chosen and composed. The fourth Chapter consists of the analysis of the feedback about the lab given by students who completed it within the university course. Furthermore, some recommendations for future improvements are suggested. The thesis ends with a summary where the main results about usage of the developed materials are stated.

2 Background and Existing Materials

This Section gives an overview of the course the lab is part of, the topic of the lab and the materials that were used in the previous years.

2.1 Course at the University of Tartu "Software Testing (MTAT.03.159)"

"Software Testing" is a 3 ECTS course taught every spring semester at the University of Tartu. The course explains systematic methods of software testing. According to the official course page, "Software Testing" aims at introducing the most important aspects of software quality control as well as various techniques used in testing [2]. The course held in the spring semester of 2015/2016 consisted of 7 lectures and 6 labs, each lab introducing a new testing technique in the following order:

1. Issue Reporting
2. Black-Box Testing
3. White-Box Testing
4. Document Inspection and Defect Prediction
5. Static Code Analysis
6. Automated GUI Testing (Regression Testing)

The goal of this thesis is to improve the materials for lab 5 (Static Code Analysis).

2.2 Static Code Analysis

Static analysis [3] is a testing method which involves looking for known error patterns or unconventional code in order to find faults without ever actually executing the code. The main advantages of static analysis are [1, 4]:

- It helps to find faults in places that are rarely executed in code;
- It helps to find faults in their exact location in code;
- It helps to maintain records on the faults found since formal documents are usually created;
- It helps to find code which might cause maintenance issues;
- It helps to detect quality issues earlier in the product development when they are cheaper to fix;
- It helps to detect security issues.

This means static analysis holds a lot of good qualities. One of the most widely used techniques for static code analysis is code review [3] in which pieces of code are inspected by other developers who then report all found faults. Code review has proven to be highly effective in detecting weaknesses already for a long time [5], and has been recently adapted in several companies [6]. However, code review can demand a lot of time and money when working on bigger projects, which has led to the need for automated tools.

In the 1970s Stephen Johnson created a tool called Lint [7], which examined C source programs and tried to find known error patterns, without ever executing the code. Lint proved to find many faults that would have normally be revealed during code review much faster and cheaper.

Since then, many new automatic code review tools have emerged and they have proven to be quite effective [8].

2.3 Last Year's Lab

In the „Software Testing" course, a lab originally created and performed by Vahid Garousi at the University of Calgary, Canada, introducing static code analysis has been in place for many years without major changes. The lab introduces a static code analysis tool for Java called FindBugs[1] and uses it on an open-source project called JFreeChart[2]. Students work in pairs to complete tasks.

2.3.1 Structure

The lab materials used in the previous years consisted of an instructions document and a zip-file containing the system under test. The instructions document describes the following main tasks:

- Familiarizing with running FindBugs and reading on analyzing found warnings by looking at an example *"Call to equals() comparing different types"* warning and deciding whether it is a false-positive or not;
- Analyzing seven randomly chosen bugs other than *"Possible null-pointer dereference"* and one *"Possible null-pointer dereference"* bug;
- Comparing the benefits of static code analysis versus manual code inspection;
- Writing a short feedback about the lab.

The lab gives a total of 10 points, which are distributed as follows:

- 5 points for the analysis of seven randomly chosen bugs other than "Possible null-pointer dereference";
- 1 point for the analysis of one "Possible null-pointer dereference" bug;
- 2 points for the comparison of the benefits of static code analysis versus manual code inspection;
- 1 point for the feedback.

[1] http://findbugs.sourceforge.net/

[2] http://www.jfree.org/jfreechart/

[3] https://eclipse.org/

2.3.2 Feedback Analysis

In every "Software Testing" lab students are asked to write a short feedback explaining what they liked or did not like about the lab. The exact question asked is the following:

"General comments and conclusions on performing the lab. Did you find it a useful practice? Was it easy to follow? Did you spend too little/too much time on it? Should some parts be dropped or explained better? Is something missing? Etc. Please try to be constructive."

After analyzing the feedback for lab 5 "Using Static Analysis Tools to Find Bugs" of the "Software Testing" course taught in 2014/2015 the following results can be reported.

Negative comments mentioned in more than five feedbacks:

- JFreeChart was too extensive to be fully understood in the given timeframe. A lot of background information was needed to determine the purpose of a single method;
- Too little information was given on how to analyze the issues;
- The workload was too big since determining whether an issue was a false positive or not was difficult.

Other negative comments mentioned:

- An outdated version of FindBugs was mentioned in the materials;
- Students would like to have an opportunity to choose between static code analysis tools;
- Problems occurred with using Eclipse and FindBugs;
- Installation and setting up the tool took away most of the time;
- Many of the bugs were very language-specific;
- The system under test did not support different integrated development environments.

Positive comments mentioned:

- FindBugs plugin was easy to use.

Consequently, the system under test, explanations on analyzing found issues and the workload had to be reconsidered in order to improve the existing materials. Even though it was mentioned that FindBugs was easy to use, the idea of changing the tool was taken into consideration. Other open-source tools were such as CheckStyle[4], PMD[5] and Lint4J[6] were considered but FindBugs seemed to have the best range of warnings and the easiest setup process to help with reaching the goal of the lab. Students also mentioned that they would like to freely choose static code analysis tools and integrated development environments instead of being restricted to FindBugs and Eclipse. However, this aspect is not relevant for the goal of this lab and therefore was left unchanged to minimize problems such as different options for preferences which may occur during setup.

[4] http://checkstyle.sourceforge.net/
[5] https://pmd.github.io/
[6] http://www.jutils.com/

3 Lab Design

This section gives an overview of the tasks and schedule of the revised lab and introduces the materials given to the students and lab instructors.

3.1 Tasks and Schedule

The tasks can be divided into two parts, i.e., preparation tasks and homework tasks. Students are expected to finish at least most of the preparation tasks in the lab with the help of the lab instructor and complete the homework tasks later.

3.1.1 Preparation Tasks

There are three main tasks which need to be completed before moving on to the homework tasks:

- Manual Code Inspection Task;
- Tool Setup and Familiarization with the System Under Test;
- Familiarization with the Process of Analyzing an Issue.

Manual Code Inspection Task

The goal of the manual code inspection task is to raise the awareness of the need and usefulness of static code analysis tools by having students inspect a piece of code manually to try to spot faults. For this task, the faults are chosen deliberately to be known to the students from previous courses. The idea is that even very talented students, who might think that they would never write faulty code, should not be able to find all faults in a reasonable time that a tool would spot in seconds.

When in the lab session, lab instructors hand out the piece of code found in appendix “First Task” on paper. Students are given five minutes to find as many faults as they can. Since this task is meant only for raising students’ awareness, it is not graded in any way and therefore there is no demand for students to report found faults in writing. After the five minutes have passed, the lab instructor asks how many and what kind of faults students found, which are then briefly orally discussed in class. Following the discussion, the lab instructor presents and explains how all of these and even more faults can be found in just a second with FindBugs.

The piece of code represents a single Java class with 6 variables, a constructor, two getters and one longer method. The code contains 14 seeded bugs, for example in the following section:

```
if (name == "Harry"){
        name.replace('r', 'p');
}
```

One can notice two faults:

1. In the condition of the if-statement two strings are compared using "==", which leads to reference equality comparison. It is highly probable that in the case here, the *equals(Object)* method should be used instead;
2. Inside the if-statement, the return value of the *String.replace(char, char)* method is ignored. Since strings are immutable in Java, the return value of the method should also be assigned to the variable itself.

Students do not run FindBugs on this piece of code, instead the lab instructors present the output on the screen. FindBugs only detects true positives in this code. Therefore, students should be able to find all of them by manual inspection. For example, the first fault noted in the previous example can be seen in the FindBugs perspective as shown in Figure 1.

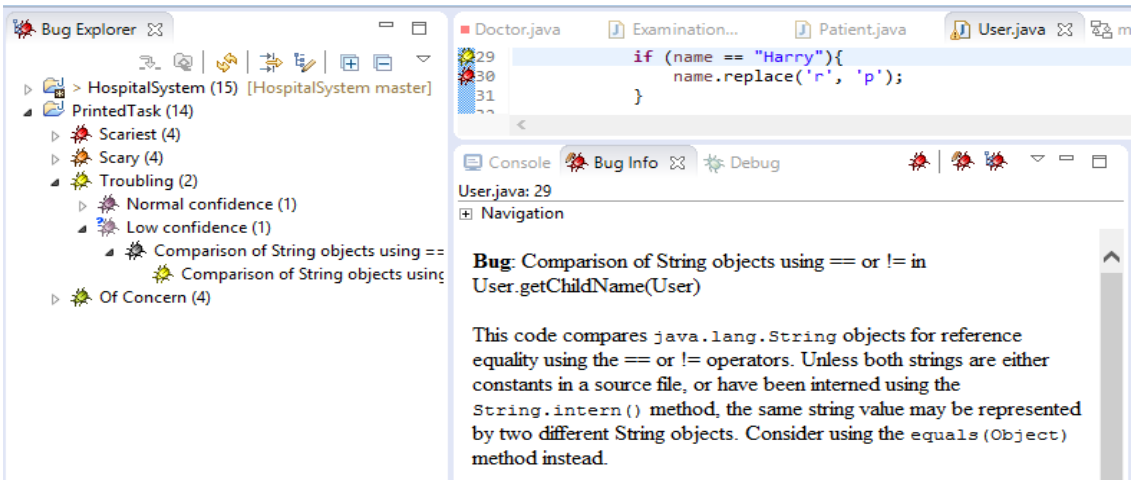


Figure 1. FindBugs perspective.

As seen from Figure 1, FindBugs reports 14 faults in the piece of code. The faults were seeded in the code on purpose, considering aspects talked about in the course "Object-Oriented Programming"[7], faults noted to be common in research [9] and the complexity of the faults. Some faults, such as self-assignment of fields in constructors where they are not given as arguments or apparent self-invoking methods which might cause infinite recursive loops are expected to be noticed easily by most students when examining the code. It is also probable that many students will notice that some fields are left unused but might not note this down as a fault even though it should be. Some other faults, even though they are repeatedly mentioned in the course "Object-Oriented Programming" will most likely stay unnoticed. Examples of such faults are ignoring the return value of a method, for example a method for modifying strings and null-pointer dereference in an or-statement, which are easy to be overlooked. Nevertheless, when the faults are mentioned to the students, they will most probably remember having heard about these problems before and consequently realize the importance of static code analysis tools.

[7] https://courses.cs.ut.ee/2015/OOP/spring

This is a completely new task added to the lab package and therefore a big change in the lab materials and work flow. In the previous years, the comparison of automatic versus manual static analysis was addressed in one of the home tasks, where students were asked to compare the two methods. This year the comparison task was changed since the separate manual code inspection task is done during the lab.

Tool Setup and Familiarization with the System Under Test

After completing the manual code inspection task, students move on to setting up the necessary tools and the system under test. There are three main things to set up: Eclipse, the system under test, which is HospitalSystem in this lab, and FindBugs. HospitalSystem is added to this thesis as appendix "HospitalSystem". Before moving on to the technical setup, students get a brief overview of what the system used in this lab is like and what are its main functionalities.

HospitalSystem is a basic Java application which was created only as a sample project and should therefore not be thought of as a complete system. The project consists of seven classes and only allows the user to do some simple actions. The class diagram of the system is shown in Figure 2. The user interface of the system is shown in Figure 3.

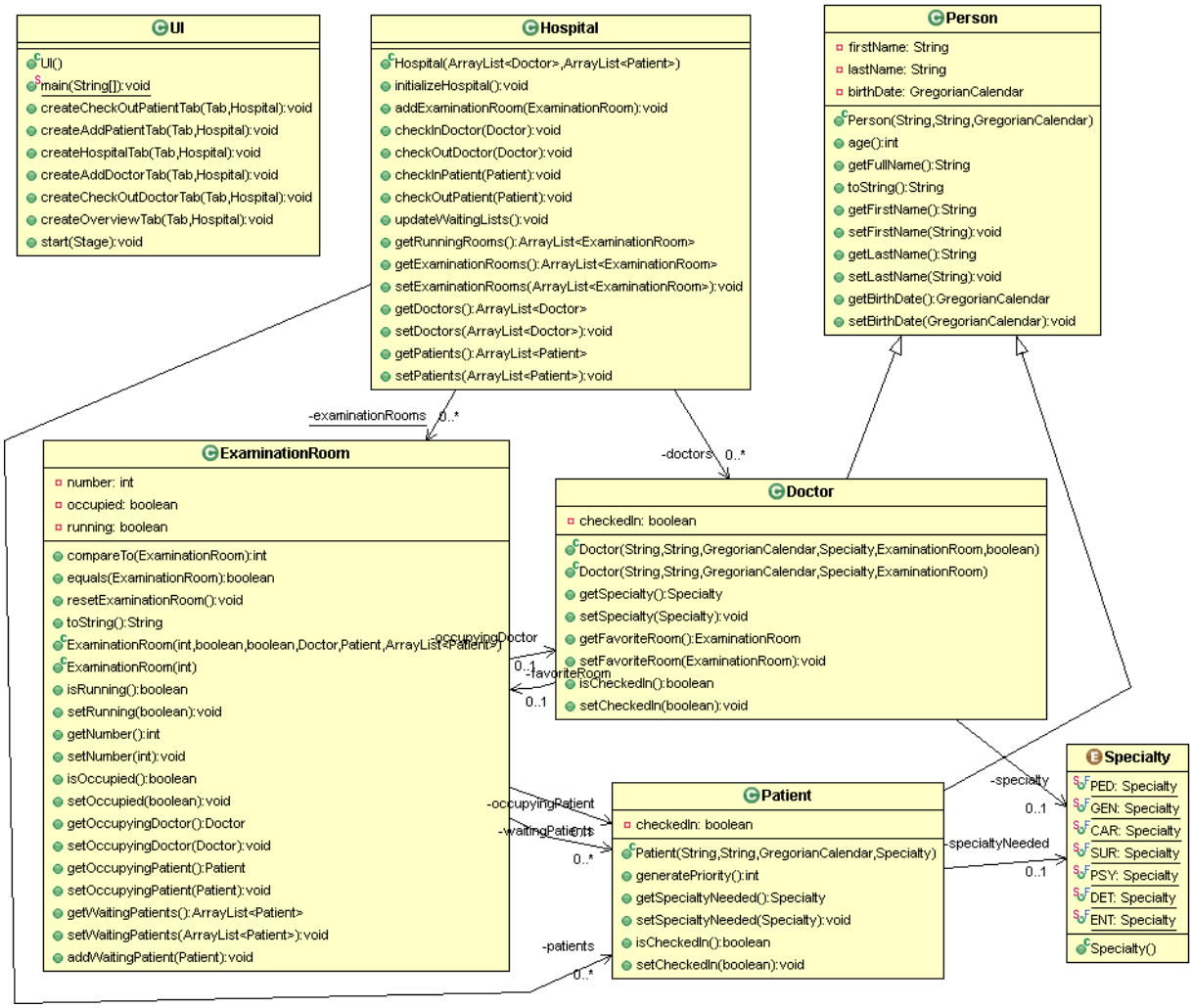


Figure 2. Class diagram of HospitalSystem.

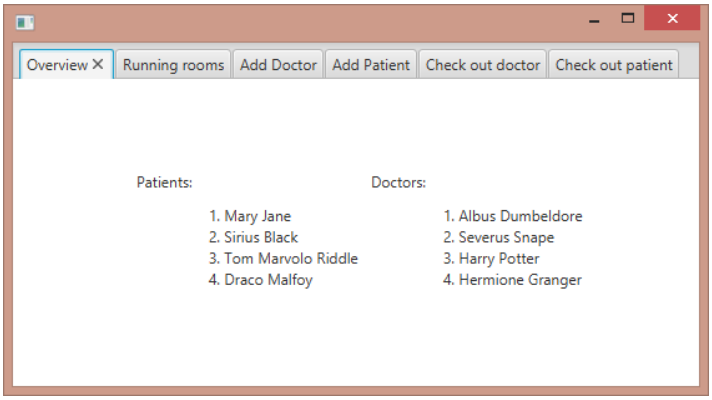


Figure 3. User interface of HospitalSystem.

The main functionalities of the system include checking in and out patients and doctors and seeing the overview of the current status of the hospital.

The tool setup instructions were left mostly unchanged compared to the ones in last year's materials, with the exception of updating software version numbers and screenshots. Since students complained about the system under test used last year and as several other labs in the "Software Testing" course use sample projects which are specifically adjusted to the learning goals of a lab, a specific project was created for the static code analysis lab as well. In the previous years, the system under test was pretty extensive and FindBugs reported 184 warnings in the project, out of which students were asked to choose 8 to analyze, deciding whether they are false positives or not. Even though the fact that FindBugs finds nearly 200 potential faults is a good example of how the tools might work on real-life projects, having less warnings makes the task easier to comprehend for the students. It reduces the workload of the lab instructors, since the warnings chosen by different student pairs greatly overlap. This gives the lab instructors more time for detailed feedback to the students.

Familiarization with the Process of Analyzing an Issue

A big problem in static code analysis tools is the fact that they tend to produce many false positives. These types of warnings are the most difficult to deal with and the concept might be difficult to understand for the students at first. One of the tasks that students receive later on is to determine themselves whether a fault reported by FindBugs is a true or false positive. In order to help the students realize how to analyze warnings reported by FindBugs, an example analysis is given as part of the lab materials.

The document can be found in appendix "Analyze an Issue". It shows an example of a warning reported by FindBugs and explains how to determine whether the given warning is a false positive or not by giving two possible implementations of the class where the potential fault was found. One possible implementation shows the case in which the warning would be a true positive and the other one shows the case in which the warning would be a false positive. The students will therefore learn that in order to determine whether a given warning is relevant or not, they might need to look deeper into the program.

Comparing to last year's lab, little was changed in this section. The description was made a bit clearer and two possible implementations were brought out instead of one. It is also important to make sure that students would be familiar with the fault brought as an example, because otherwise it would be very difficult to quickly realize the difference between the implementations. The example fault given in the document refers to a possible fault of using an *equals()* method to compare objects from different types, which is very similar to the one used last year. The type of the fault was not changed since faults of this kind are introduced in the course "Object-oriented programming" which is a preliminary course for "Software Testing" and therefore makes it very likely that most of the students are familiar with faults of this kind.

3.1.2 Homework Tasks

There are two tasks that need to be handed in:

- Analyzing bugs;
- Comparing methods.

Analyzing Bugs

In order to reach the goals of this lab, students are asked to run FindBugs on the system under test and analyze a number of the reported warnings. Each warning needs to be analyzed separately. To do that, students are asked to fill out a table, where each row corresponds to one warning. A table with the first row filled in is given to the students as an example and the task is to fill out the rest of it. Table 1 shows the table with the example given to the students.

Table 1. Bug analysis table with an example.

Bug description and location	What seems to be the problem?	Is this a false or a true positive? Should this bug be fixed? If yes, then how, otherwise why not?
Exception created and dropped rather than thrown Person.java:15	This code creates an exception (or error) object, but doesn't do anything with it.	This is a true positive and should be fixed. Besides simply creating the instance of the exception, it should also be thrown. The fixed line would be: *throw new Exception("Birthyear cannot be in the future!");*

The system under test for this lab is HospitalSystem, which was built specially for this lab and bugs seeded into this system were chosen deliberately to be common and understandable. The new system consists of a total of 731 lines of code, divided between 7 classes. FindBugs finds potential faults from all different ranks and confidences from the code. There are 15 warnings in total, which are distributed across classes and should all be understandable taking under consideration the mandatory courses that the students must have taken before enrolling in the "Software Testing" course. A class diagram that displays the amount and distribution of warnings in the code can be seen in Figure 4.

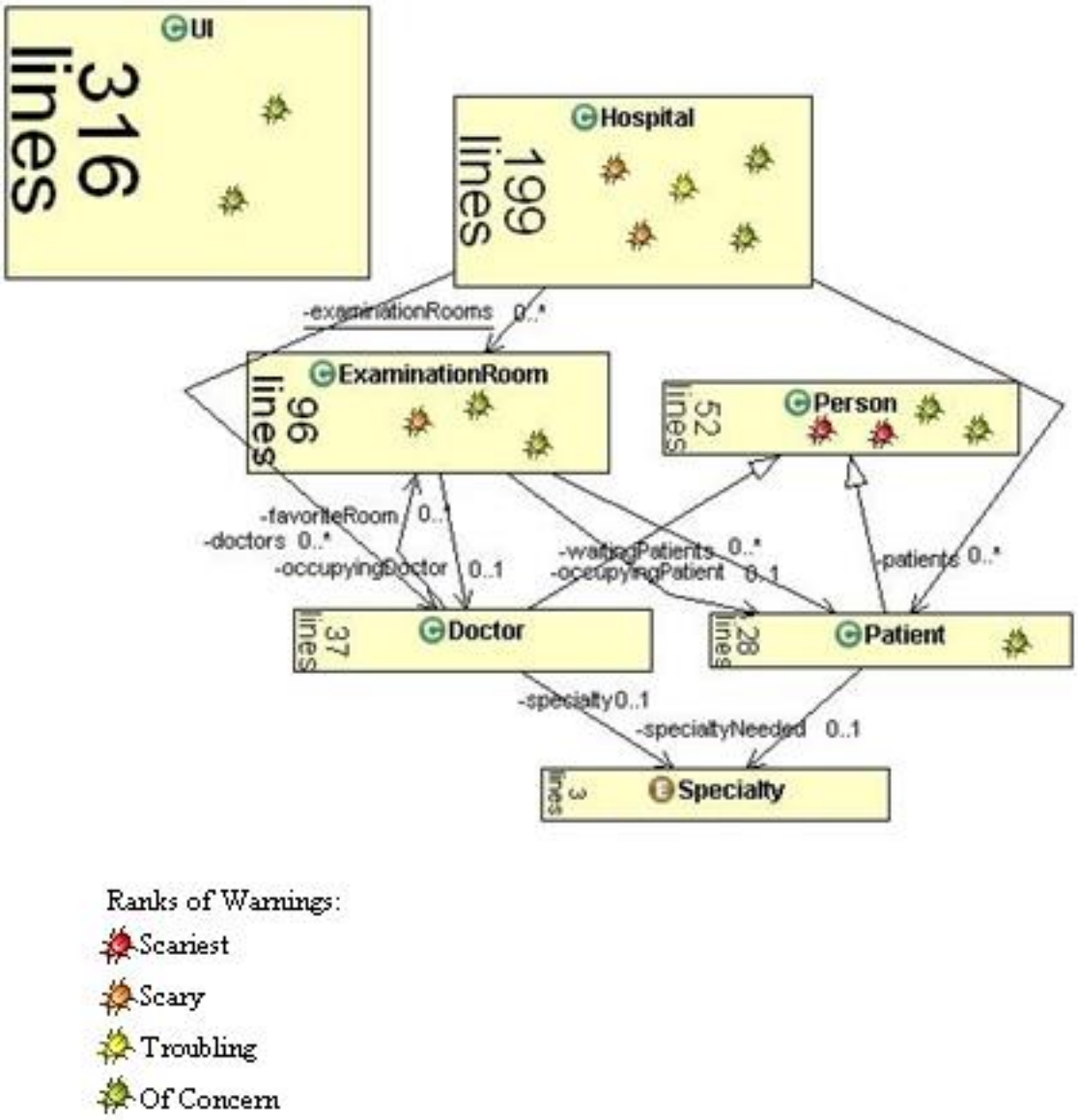


Figure 4. Warning distribution in HospitalSystem

Students are asked to analyze 10 warnings, choosing at least one from each rank (Of Concern, Troubling, Scary, Scariest) and confidence (Low, Normal, High). When running FindBugs on the target system, all found warnings are reported in the Bug Explorer view. When selecting a certain warning, students see the location and a short description of the found warning as seen in Figure 5.

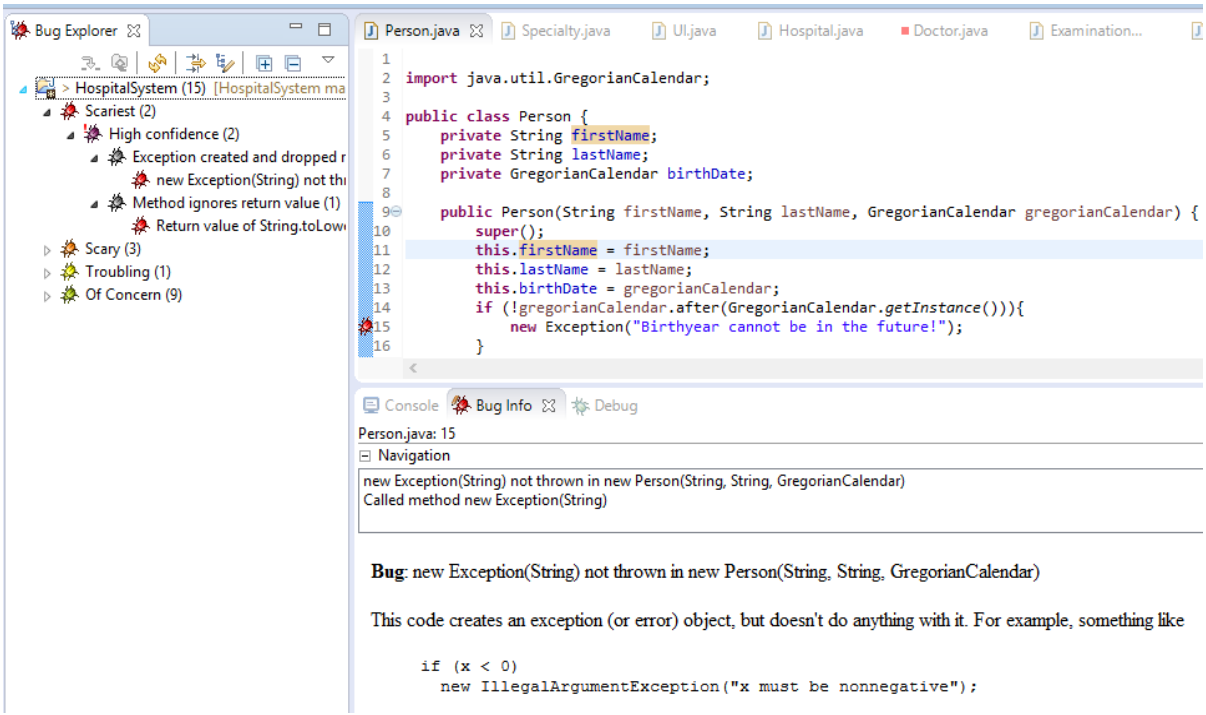


Figure 5. HospitalSystem in FindBugs perspective

Having read what FindBugs says about the warning, the task for the students is to understand why the tool counts this as a fault and mainly, whether the warning is a false positive or not.

While the example warning seen in Table 1 is a clear true positive, students might also encounter some false positives in the code. For example, FindBugs reports a null pointer dereference warning as seen in Figure 6.

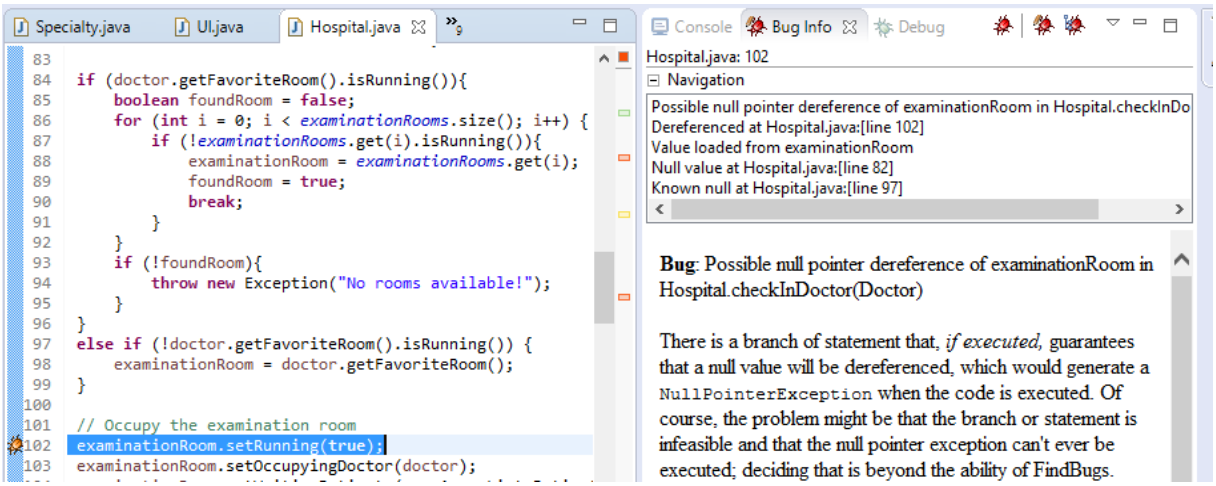


Figure 6. Possible null pointer dereference as seen in FindBugs perspective.

When looking closer at the code, one can see that in fact, this is a false positive, since one of the *if* and *else if* statements is always true and therefore the variable *examinationRoom* will always have a value set to it in line 102. However, this is not a good style for coding and therefore it is also correct for the students to note that the second *else if* statement could be changed to an *else* statement. Students are expected to write a similar analysis as shown in Table 2.

Table 2. Expected analysis of bug shown on Figure 6 [A1.8].

Bug description and location	What seems to be the problem?	Is this a false or a true positive? Should this bug be fixed? If yes, then how, otherwise why not?
Possible null pointer dereference Hospital.java:102	The variable examinationRoom is initialized as null and later given values in if-else if statements. Since there is no else statement, FindBugs is not sure whether the variable is always assigned a value before it is dereferenced.	This is a false positive, since the if and else if statement cover all possible situations. However, this coding style should be discouraged and the else if statement should be replaced by an else clause.

Even though the principles of this task are the same as in the materials for last year, this time the analysis is asked to be reported in the form of a table, which helps make the task clearer for the students and also facilitate the work of the lab instructors. Furthermore, the scale of the system and the number of the faults to be analyzed was changed to fit it better into the given timeframe.

Comparing Methods

In order to raise the awareness of the use of static code analysis even more, students are asked to briefly compare the benefits of static testing vs dynamic testing by bringing up at least one concrete benefit of each method. The expected result is for the students to mention that when static code analysis is useful for finding faults in areas of the code which are rarely executed, dynamic testing can find out whether all required functionalities have been implemented and work as expected.

Last year, students were asked to compare automatic static code analysis versus manual static code analysis. As this comparison is now covered by the manual code inspection task added to the beginning of this lab, this task was changed to let the students also think over the benefits of static code analysis over dynamic testing and vice versa.

3.1.3 Schedule

"Software Testing" is a 3 ECTS course where each lab is estimated to take about 8 student hours per person to complete. These 8 student hours or 360 minutes include the 2 hours or 90 minutes in the lab session and are expected to be spent as follows:

- 45 min – Lab introduction and manual code inspection task;
- 45 min – Tool and system setup;
- 45 min – Getting familiar with the tasks and materials, inspecting the example on analyzing an issue;
- 150 min – Completing the task on analyzing bugs;
- 45 min – Completing the task on comparing methods;
- 30 min – Feedback and submission.

In order to make sure that the actual workload of an average student corresponds to the expected values, two experiments were carried out before the lab session to the test some of the tasks. Experiment 1 was carried out to test and set the time limit for the manual code inspection task and Experiment 2 to set the deliverables for the task on analyzing bugs.

Experiment 1

In the interest of finding out how many faults an especially talented student will find in a given timeframe and therefore decide whether the prepared task is suitable or not and how long the time given to the students should be, four third-year Bachelor computer science students who have already passed the "Software Testing" course with exceptionally high results were asked to do the manual code inspection task. The students were asked to go through the given code while an examiner noted down all of the faults that the students spotted. The results are shown in Table 3. Faults spotted within the first 5 minutes are listed in the second column and additional faults found in the next 5 minutes are listed in the third column.

Table 3. Results of Experiment 1.

Student	Faults found in 5 mins	Additional faults found in next 5 mins
Student 1	1	2
Student 2	3	1
Student 3	6	1
Student 4	3	0

As seen from Table 3, most of the faults were found in the first five minutes and therefore this seems to be a reasonable time limit, regarding that the additional time did not result in a significantly higher number of found faults. It should also be noted that the students participating in the experiment only found a maximum of seven faults in 10 minutes. Therefore,

it can be expected that average students will not find all 14 bugs during the given timeframe and hence the task will fulfil its purpose.

Experiment 2

Last year, students were asked to analyze 8 bugs in total. Since this year a much smaller system is put under test and the bugs to be examined were intentionally seeded to the code, taking under consideration the students' skills acquired in predecessor courses, an experiment was conducted to determine how many bugs should be given to the students to analyze.

In the interest of finding out how long it takes for students to analyze a fault and therefore later decide how many faults a pair of students should analyze, 5 third-year Computer Science bachelor students who have already passed the "Software Testing" course with varying results were asked to set up the system under test and complete the task on analyzing bugs individually. In addition, they were asked to note down the time it took them to analyze each warning. The subjects completed the task by themselves and sent back their results in writing. It is understandable that time is very valuable and therefore each student analyzed exactly as many warnings as they had time for. The results are shown in Table 4.

Table 4. Results of Experiment 2.

Student	Total time spent	Faults analyzed	Average time spent on a fault
Student 1	39.2 minutes	7	5.6 minutes
Student 2	54.6 minutes	13	4.2 minutes
Student 3	56.5 minutes	5	11.3 minutes
Student 4	15 minutes	1	15 minutes
Student 5	20 minutes	1	20 minutes

As seen from the previous table, the results seem to vary a lot. A few key points to keep in mind when looking at the results are that students 1 and 2 completed the "Software Testing" course last year with very high results, 3 and 4 with average and 5 with a rather low result. Nevertheless, even though student 2 analyzed the most bugs and achieved the lowest average time, the analysis was often very poor and even incorrect. Furthermore, even though students 4 and 5 only analyzed 1 fault, they noted that the task was rather complicated and when looking at the other faults, it seemed that each one would take about 15-20 minutes to analyze.

The results indicate that an average student from the test group would spend around 11 minutes to analyze a bug individually. Even though working in pairs might reduce that time a little, it should be considered that the students from the test group had already done a similar task in last year's course and were therefore more familiar with the analysis process than the students who will be taking the lab this year. All in all, based on the experiment, it is expected that an average student pair will spend an average of 10 minutes to analyze a bug in the code while a weaker pair will spend around 15 minutes. Considering these metrics and the time estimated to be spent on this task, students are asked to analyze 10 bugs in total.

3.1.4 Point distribution

In the "Software Testing" course, each lab gives a maximum of ten points. Eight points out of those ten are given for the tasks mentioned above. Two points are given for active participation in the lab session and for writing a constructive feedback about the lab. Considering those matters, the ten points for lab 5 are distributed as follows:

1. In-Lab Activity – 1 point;
2. Analyze bugs – 7 points;
3. Compare methods – 1 point;
4. Feedback – 1 point.

3.2 Lab Materials

In order to make the tasks as clear as possible and to make the materials manageable, several documents were provided both to the students and to the lab instructors. A scheme of all the documents associated with the lab package can be seen in Figure 7.

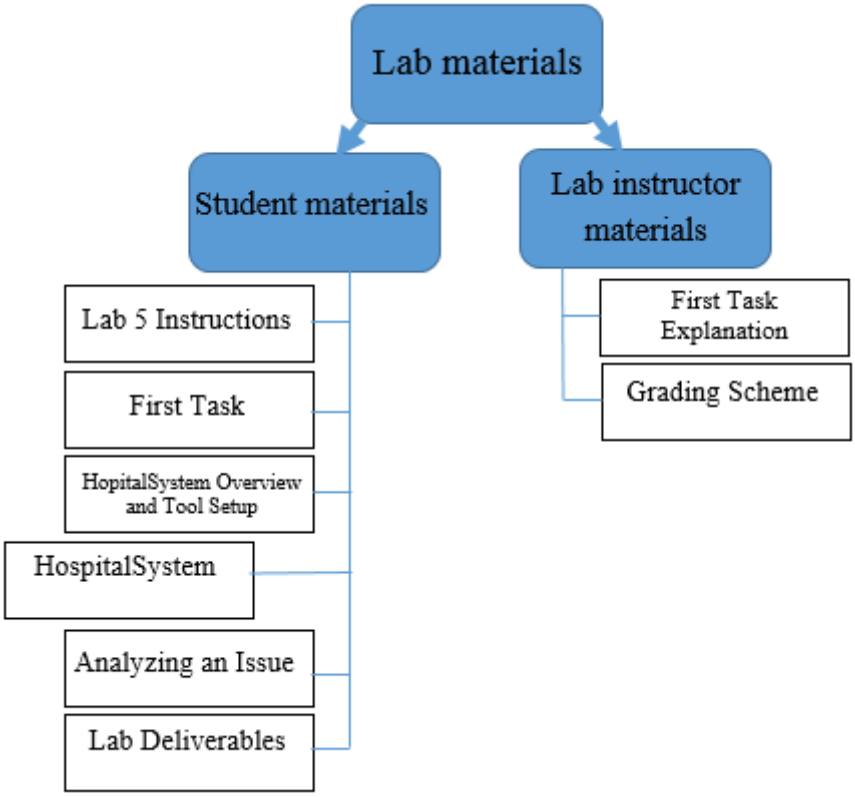


Figure 7. Structure of the lab materials.

All of the materials seen in Figure 7 can be found in Appendix 1 under the same titles. All of the student materials are also available on the "Software Testing" course wiki page. The content and purpose of each document is explained in the following paragraphs. The student materials are found in Sections 3.2.1-3.2.6 and the lab instructors' materials on Sections 3.2.7-3.2.8.

3.2.1 Lab 5 Instructions

The first document introduced to the students is a short document meant to be used as a guideline for the lab. It starts with a general introduction to the topic, tool to be used and system under test. After the introduction, it presents the order in which the tasks in this lab should be done. The grading and deliverables are also briefly mentioned at the end. Each Section refers to the specific document on the wiki page which has detailed information about the given task or topic.

The instructions document was added to the lab package in order to make the big amount of information systematized and hence easier to comprehend. Since the lab consists of several independent tasks, scrolling through a long document can get overwhelming and confusing. In order to improve this, the material was divided into smaller parts following the example used in the thesis of Rasmus Sõõru [10] which facilitates navigation between the separate tasks. Furthermore, this document can be used by the lab instructors to give an overview about the materials in the beginning of the lab session.

3.2.2 First Task

This is a pdf-document containing the Java code used in the manual code inspection task.

3.2.3 HospitalSystem Overview and Tool Setup

This is a pdf-document containing a short description of the system under test and detailed instructions on setting up the tool and the system.

3.2.4 HospitalSystem

This is a zip-file of the system under test which is meant to be imported to Eclipse as an existing project.

3.2.5 Analyzing an Issue

This is a pdf-document that shows an example of a fault reported by FindBugs and explains how to determine whether the given fault is a false positive or not by giving two possible implementations of the class where the fault was found.

3.2.6 Lab Deliverables

This is a pdf-document containing details about the point distribution, instructions for the task on analyzing bugs and comparing methods and also the in-lab activity and general feedback tasks.

3.2.7 First Task Explanation

This is a pdf-document where all the faults that FindBugs finds in the code given for the manual code inspection task are listed and each fault is also explained a little. This document is meant to help the lab instructors in showing and explaining the actual faults that appeared in the code to the students, after they have finished their task.

3.2.8 Grading scheme

This is a pdf-document that represents the same "Lab Deliverables" document that the students receive, filled out with expected answers for the two main tasks in this lab: analyzing bugs and comparing methods. This document is meant as a guideline to help the lab instructors with grading.

4 Lab Execution

The "Software Testing" course held in spring 2016 consisted of 6 labs. The materials created within this thesis were used in lab 5, which took place in seven lab groups between 11th of April and 14th of April 2016. Each lab group had approximately 20 students and one lab instructor to help and guide them. There were three lab instructors in total in this course and in order to make sure they are familiar with the materials, a brief introductory session was held with them before the actual lab sessions.

Before the lab, students already have some knowledge about the topic. Lab 4, which takes place two weeks before lab 5, introduces manual inspection of design documents and therefore already gives students some idea of static analysis. Furthermore, static code analysis is also mentioned and briefly explained in the lectures taking place before the lab session. No other specific preliminary work is required for the lab. In the lab itself, lab instructors start by explaining briefly the topic and the workflow of the lab with the help of the "Instructions" document. After this introduction, they will hand out "First Task" on paper and discuss the results afterwards. When the discussion is over, students can look up all of the materials from the "Software Testing" course wiki page and continue with the individual tasks.

In the labs everything seemed to go according to plan. The preparation tasks went smoothly. As the manual code inspection was a completely new task, the results were examined in one lab group and are shown in Table 5. As can be concluded from Table 5 and from examinations done by the lab instructors in the lab sessions, the task seemed to fulfil its purpose since all students found some, but not all faults.

Table 5. Results of the manual code inspection task in one lab group (8 students present).

Number of Faults	Students with such result
1 or less	0
2	4
3	1
4	3
5 or more	0

The setup of the tool and system did not cause any bigger problems as well. This was expected since Eclipse with its plugins and existing projects packed into a zip-file are also used in other labs in the "Software Testing" course.

As seen from the marks received for the homework tasks, many students completed their tasks well and received full marks. Since one of the aims of the "Software Testing" course is to ensure that all students would be able to complete all tasks given in the labs, this kind of a result should be thought of as a success. Nevertheless, sometimes students also lost marks for several reasons. Even though the reasons varied greatly, two mistakes occurred more often:

- Misinterpretation of the false positive warning seen in Figure 6. This fault was sometimes interpreted as a true positive which indicates that students have not understood the concept of false positives or have not examined the code deeply enough and therefore solved the task too superficially;
- Misinterpretation of the true positive warning found in line 25 in the class *ExaminationRoom* in the system under test with the following description given by FindBugs: "*ExaminationRoom* defines *equals(ExaminationRoom)* method and uses *Object.equals(Object)*". This fault was sometimes interpreted as a false positive which indicates to the lack of knowledge on the *equals()* method in Java or superficial solution to the task.

5 Feedback

In every lab in the "Software Testing" course students are asked to give constructive qualitative feedback. This task gives one out of the ten maximum points given for a lab. In addition to this qualitative feedback, an online questionnaire about lab 5 was given to the students in order to determine the value of the enhanced lab materials and decide what kind of improvements should be made in the future. A bonus point was given to students who completed the questionnaire. 67 students participated in total in the lab, out of whom 63 gave qualitative feedback and 61 filled in the online questionnaire. The details of the collection and results are presented in the following paragraphs.

5.1 Feedback Collection

The feedback was collected via two different sources. The analysis and suggestions for future improvements can be found in Sections 5.2 and 5.3.

Qualitative Feedback

Qualitative feedback was collected from the students as part of the lab deliverables. The question was the following:

"General comments and conclusions on performing the lab. Did you find it a useful practice? Was it easy to follow? Did you spend too little/too much time on it? Should some parts be dropped or explained better? Is something missing? Etc. Please try to be constructive and concrete."

Online Questionnaire

Additional feedback was collected using a SurveyMonkey[8] questionnaire. For evaluation of the questionnaire a Likert Scale [11] was used, which is a rating scale allowing a person to express how much they agree or disagree with a statement. The statements given were the following:

1. "The goals of the lab were clearly defined and communicated";
2. "The tasks of the lab were clearly defined and communicated";
3. "The materials of the lab were appropriate and useful";
4. "The FindBugs" tool was interesting to learn";
5. "If I have the choice, I will work with FindBugs again";
6. "The support received from the lab instructors was appropriate";
7. "The grading scheme was transparent and appropriate";
8. "Compared to the previous labs, the difficulty/complexity of the lab was";
9. "Overall the lab was useful in the context of this course".

For statements 1-7 and 9 there were five options to choose from: "Agree Entirely, Somewhat Agree, So-So, Somewhat Disagree, Disagree entirely". The exact form of the options was chosen based on the ones used in the official student feedback system implemented by the University of Tartu. Since statement 8 was slightly different, students could choose between: "Much Lower, Somewhat Lower, The Same, Somewhat Higher, Much Higher".

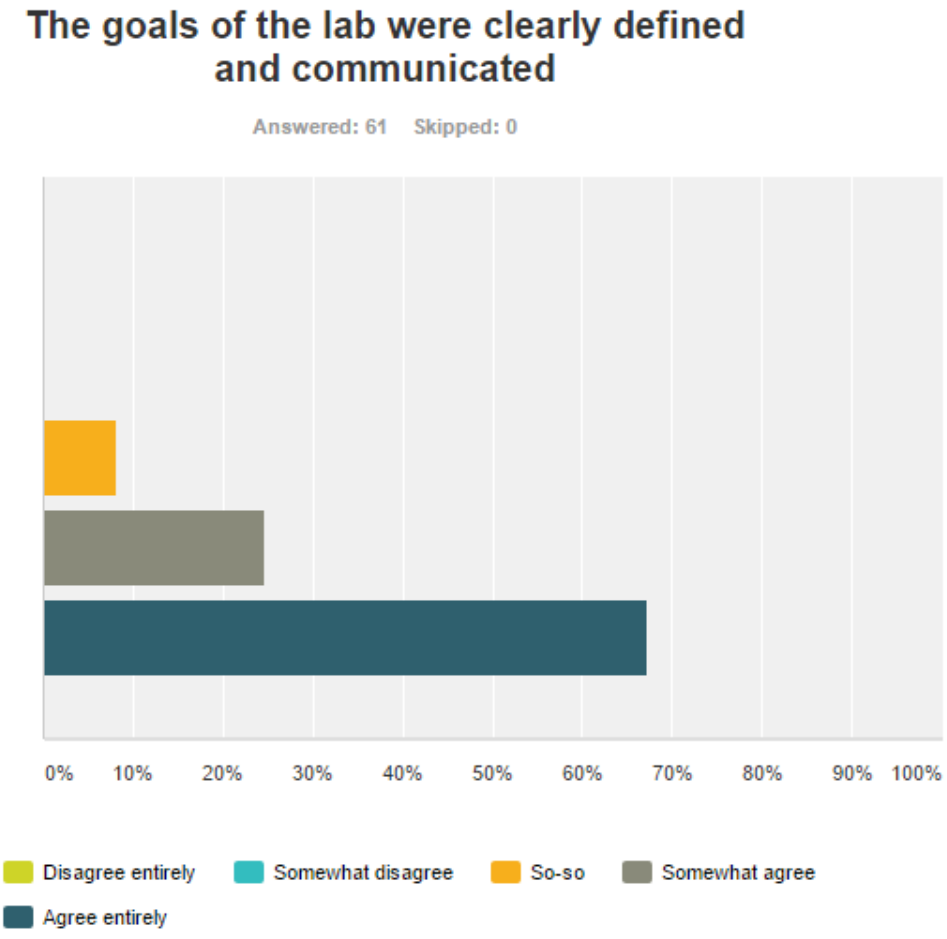
[8] https://courses.cs.ut.ee/2015/OOP/spring

5.2 Feedback Analysis

Most of the feedback received from the students was positive. Nevertheless, some negative aspects as well as improvement suggestions were also mentioned. In the following Section, both the positive and negative aspects are analyzed. Results from the online questionnaire can be found in Appendix 2.

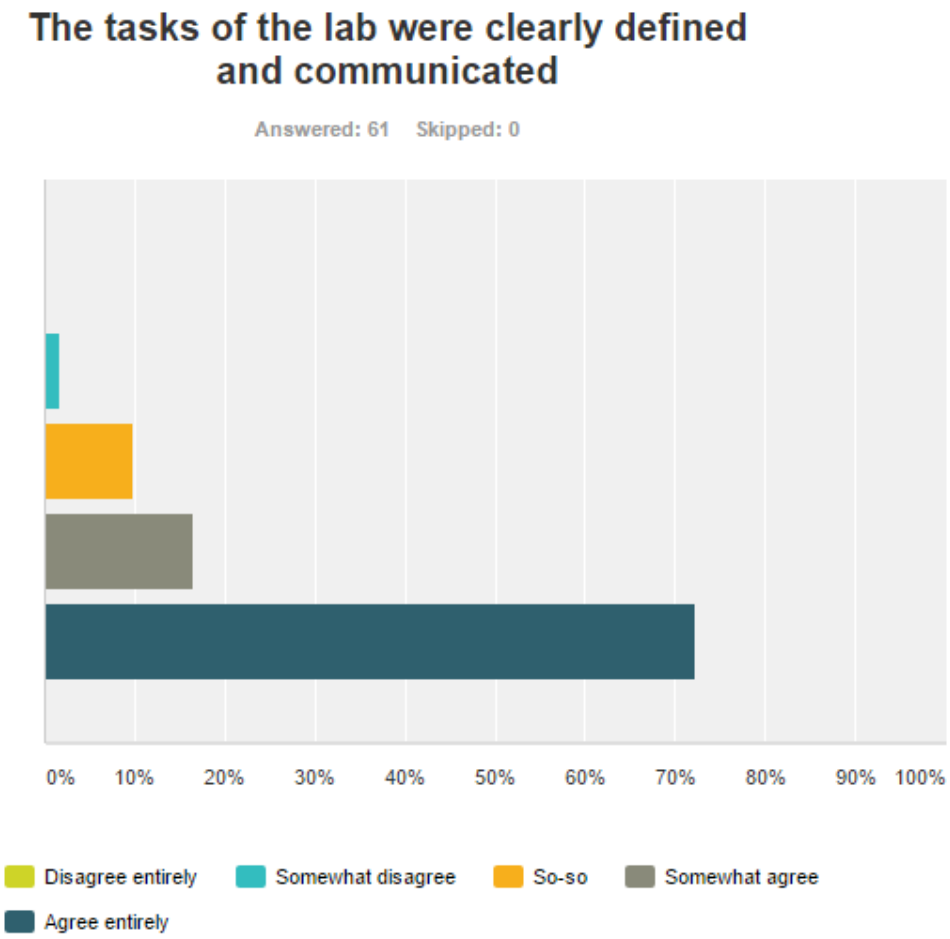
Positive aspects

In the qualitative feedback it was widely mentioned that the lab materials were well-written and easy to follow, the FindBugs tool was easy to use and the examples given in the materials were very helpful. Similar tendencies can also be seen from the online questionnaire results. Three of the questions received exceptionally high results, which are shown in Figures 8, 9 and 10. The results have been conducted into horizontal bar charts, where each one represents the percentage of students who chose the corresponding option.



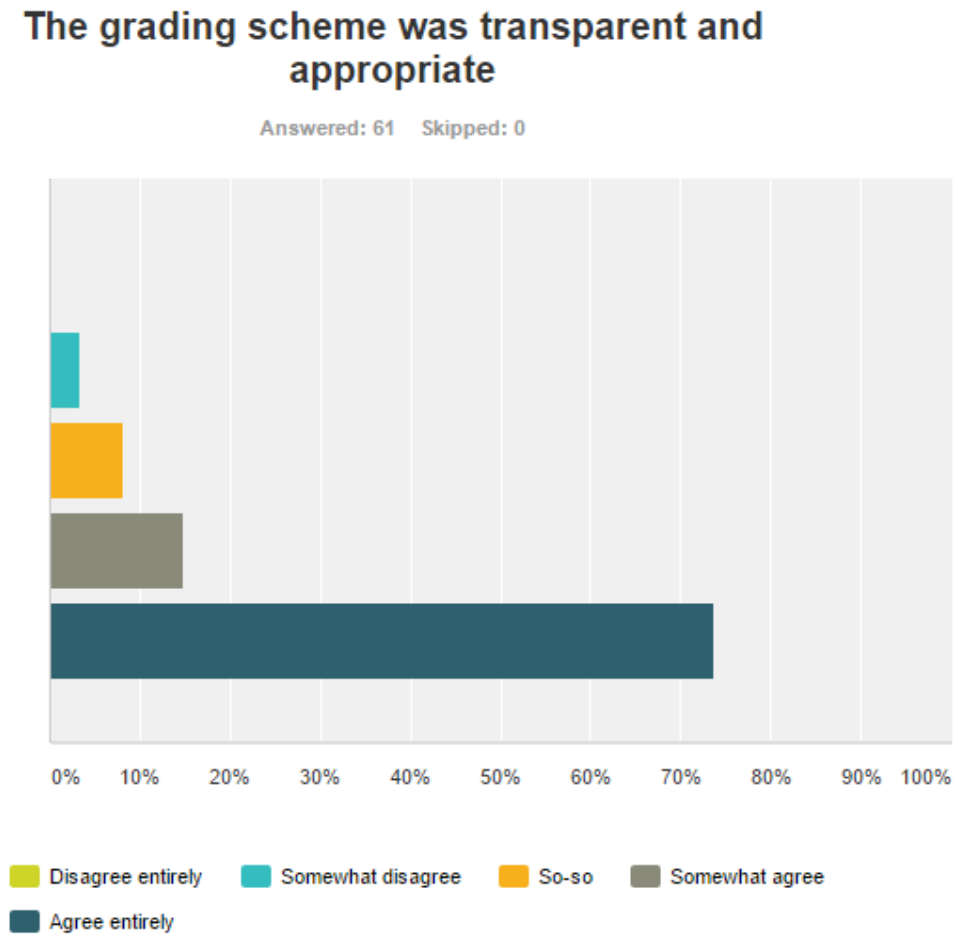
	Disagree entirely	Somewhat disagree	So-so	Somewhat agree	Agree entirely	Total	Weighted Average
	0.00% 0	0.00% 0	8.20% 5	24.59% 15	67.21% 41	61	4.59

Figure 8. Results for the statement "The goals of the lab were clearly defined and communicated"



	Disagree entirely	Somewhat disagree	So-so	Somewhat agree	Agree entirely	Total	Weighted Average
	0.00% 0	1.64% 1	9.84% 6	16.39% 10	72.13% 44	61	4.59

Figure 9. Results for the statement "The tasks of the lab were clearly defined and communicated"



	Disagree entirely	Somewhat disagree	So-so	Somewhat agree	Agree entirely	Total	Weighted Average
	0.00% 0	3.28% 2	8.20% 5	14.75% 9	73.77% 45	61	4.59

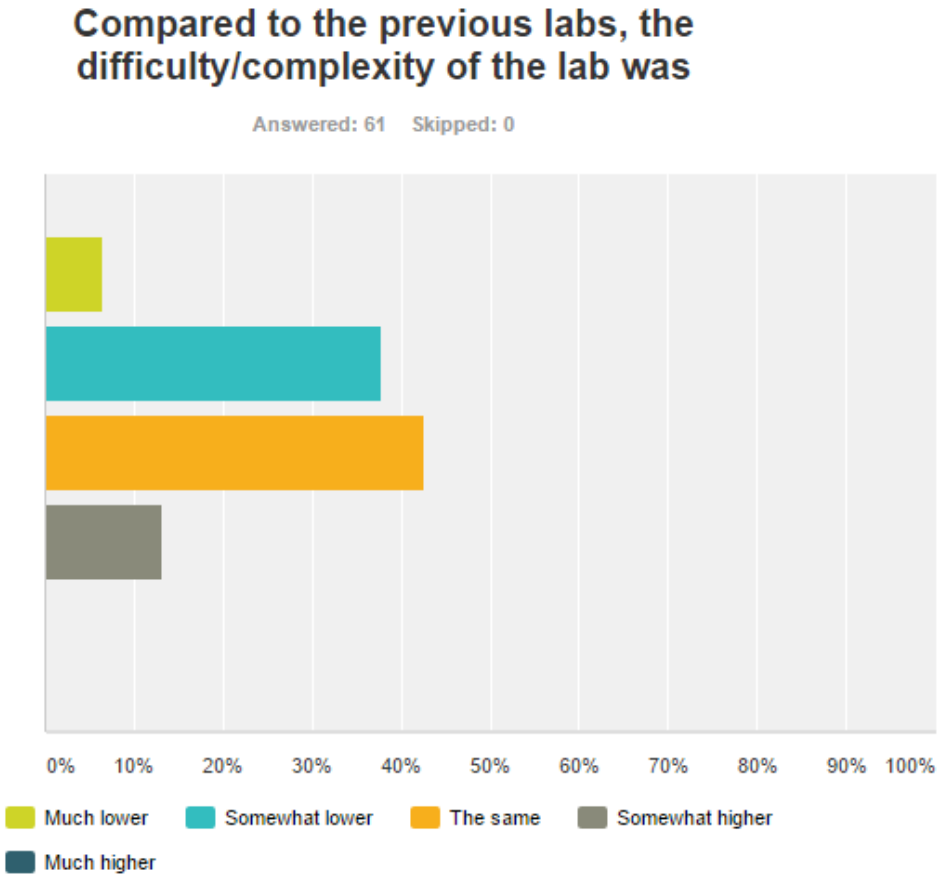
Figure 10. Results for the statement "The grading scheme was transparent and appropriate"

Negative aspects

Most negative matters mentioned in the qualitative feedback complained about technical aspects. Students requested that other Integrated Development Environments, especially IntelliJ[9] would be supported to be used in the lab. As mentioned in Section 2.3.2, this problem is not relevant for the goals of this lab. Furthermore, it was stated in a few feedbacks that FindBugs works a bit different in some other environments. Also, some students mentioned that they would like to either use more than one static analysis tool or have the chance to choose between several ones. Although this problem was not fixed for this year, a possible solution for next year is adding an extra task in which students are asked to run another tool on the system and compare the results.

[9] https://www.jetbrains.com/idea/

Another problem which did not occur from the qualitative feedback but showed clearly in the online questionnaire results, is the workload of the lab. As seen in Figure 11, the majority of the students said the workload of the lab was lower than expected. Since the results shown in Chapter 4 indicate that students usually completed all the tasks very well, one can presume that the simplification of the lab compared to last year was taken too far. On the other hand, the new lab materials have created room for more lab-specific tasks. For example, students could be asked to analyze the warnings more deeply or to compare the FindBugs output with that of other static analysis tools, as suggested in the previous paragraph.



	Much lower	Somewhat lower	The same	Somewhat higher	Much higher	Total	Weighted Average
	6.56% 4	37.70% 23	42.62% 26	13.11% 8	0.00% 0	61	2.62

Figure 11. Results for the statement "Compared to the previous labs, the difficulty/complexity of the lab was"

5.3 Future Improvements

As appeared from Section 5.2, the complexity of the lab should be increased. Some new, more difficult faults should be added into the system and students should be asked to analyze warnings more deeply. For example, in order to give students the chance to use several tools, a task could be added where students run another tool on the system and compare the outputs.

This task can also only be added as a bonus task for students already familiar with other static analysis tools.

Some other improvements were also suggested by students in their feedback and by the lab instructors. For example, it was proposed to ask students to install all the necessary tools at home to increase the productivity of the lab. Also, students would like to have the code for the manual code inspection task uploaded to the wiki page so that they could easily run FindBugs on it themselves. To add, it was said that the system under test should be examined manually before running FindBugs on it. This would allow the students to realize again the use of static code analysis tools and would force them to get acquainted with the system under test.

6 Conclusions

It is essential to assure that students have good materials to work with in order to maximize the profit of a whole course or a single lab. In this thesis project, a new version of lab materials for the lab about static code analysis was created. The new materials were used in the "Software Testing" course. Feedback was collected from the participating students and the result turned out to be very positive. It was mentioned that the workflow was clear and appropriate and the materials well put together. Furthermore, students liked the tasks and their descriptions. This thesis showed that static code analysis is an important and interesting topic which should be introduced to students. Consequently, in the future, a similar lab should be kept in place in the "Software Testing" course.

In order to assure that the lab fulfils its purpose and to gain high student satisfaction, a few improvements could be made to the lab. Firstly, the workload should be adjusted by changing the requirements for the task about analyzing bugs and/or by adding additional tasks. Since some students asked for a chance to try out other tools as well, an idea of adding a task where students are asked to compare the output of several tools was suggested within this thesis. Overall, since the scope of this was limited, not all possible opportunities were examined. The next step would be to consider different teaching methods, task types and other aspects which were not talked about in this thesis, in order to raise the quality of the materials even more.

References

[1] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in 35th International Conference on Software Engineering (ICSE), IEEE/ACM, 2013, pp. 672-681.

[2] Course Outline (2015). [Online]. https://courses.cs.ut.ee/MTAT.03.159/2015_spring/uploads/Main/swt2015-outline-v1.pdf (Accessed: 10.05.2016)

[3] I. Sommerville, Software Engineering. Pearson, 2011.

[4] A. Vetro', M. Morisio, and M. Torchiano, "An empirical validation of FindBugs issues related to defects," in 15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011), 2011, pp. 144–153.

[5]M. Fagan, "A history of software inspections," Software Pioneers: Contributions to Software Engineering, Springer, 2002, pp. 562–573.

[6] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German, "Contemporary peer review in action: Lessons from open source development," IEEE Software, vol. 29, no. 6, Nov 2012, pp. 56–61.

[7] S. Johnson, Lint: A C Program Checker, tech. report 65, Bell Laboratories, Dec. 1977

[8] S. Panichella, V. Arnaoudova, M. Di Penta, G. Antoniol, "Would Static Analysis Tools Help Developers with Code Reviews?" in 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 161-170.

[9] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh, "Using Static Analysis to Find Bugs", IEEE Software, vol. 25, no. 5, Sept-Oct 2008, pp. 22-29.

[10] R. Sõõru (2015). "Lab Package: Automated GUI Regression Testing", Bachelor's thesis, the University of Tartu.

[11] S. McLeod (2008), "Likert Scale". [Online]. http://www.simplypsychology.org/likert-scale.html (Accessed: 10.05.2016)

Appendix

I New Lab Materials

Student Materials

A1.1 – “Lab 5 Instructions”, pdf-document –

https://courses.cs.ut.ee/2016/SWT2016/spring/uploads/Main/SWT2016-lab5-Instructions-rev1.pdf

A1.2 – “First Task”, pdf-document –

https://courses.cs.ut.ee/2016/SWT2016/spring/uploads/Main/SWT2016-lab5-FirstTask-rev1.pdf

A1.3 – “HospitalSystem Overview and Tool Setup”, pdf-document –

https://courses.cs.ut.ee/2016/SWT2016/spring/uploads/Main/SWT2016-lab5-HSovToolSetup-rev1.pdf

A1.4 – “HospitalSystem”, zip-file –

https://courses.cs.ut.ee/2016/SWT2016/spring/uploads/Main/SWT2016-lab5-HospitalSystem.zip

A1.5 – “Analyzing an Issue”, pdf-document –

https://courses.cs.ut.ee/2016/SWT2016/spring/uploads/Main/SWT2016-lab5-Analyzinganissue-rev1.pdf

A1.6 – “Lab Deliverables”, pdf-document –

https://courses.cs.ut.ee/2016/SWT2016/spring/uploads/Main/SWT2016-lab5-LabDeliverables-rev1.pdf

Lab Instructor Materials

A1.7 – “First Task Explanation”

A1.8 – “Grading Scheme”

For confidentiality reasons lab instructor materials are not made available in the thesis but will be made available on request.

II Questionnaire feedback

Statement	Disagree entirely	Somewhat Disagree	So-So	Somewhat Agree	Agree Entirely	Weighted Average
The goals of the lab were clearly defined and communicated	0.00%	0.00%	8.20%	24.59%	67.21%	4.59
The tasks of the lab were clearly defined and communicated	0.00%	1.64%	9.84%	16.39%	72.13%	4.59
The materials of the lab were appropriate and useful	0.00%	1.64%	6.56%	31.15%	60.66%	4.51
The FindBugs tool was interesting to learn	0.00%	1.64%	18.03%	34.43%	45.90%	4.25
If I have the choice, I will work with FindBugs again	4.92%	4.92%	16.39%	45.90%	27.87%	3.87
The support received from the lab instructors was appropriate	0.00%	1.64%	9.84%	26.23%	62.30%	4.49
The grading was transparent and appropriate	0.00%	3.28%	8.20%	14.75%	73.77%	4.59
Overall the lab was useful in the context of this course	0.00%	1.67%	3.33%	30.00%	65.00%	4.58

III License

Non-exclusive licence to reproduce thesis and make thesis public

I, Forename Surname,

Kristine Leetberg

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Title,

"Lab Package: Static Code Analysis"

supervised by ,

Dietmar Pfahl

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **10.05.2016**