MTAT.03.159-Software Testing

Lab. #5 - Using Static Analysis Tools to Find Bugs

Instructor: Dietmar Pfahl (dietmar.pfahl@ut.ee)
Teaching Assistant: Svetlana Omelkova (svetlana.omelkova@ut.ee)
Institute Of Computer Science
University of Tartu

TABLE OF CONTENTS

1 INTRODUCTION

In this lab, students will run a static code analysis tool [1] on an existing codebase and analyze the results. Static code analysis tools are designed to conduct automated inspection on a given codebase and detect potential defects. Static code analysis is an advanced bug-finding technique in the industry and there are open-source and commercial tools for this purpose. Students will execute an analysis tool on an existing project and interpret its output.

1.1 OBJECTIVES

The main objective of this lab is to familiarize students with static code analysis tools; how they are used and how they help to find faults. Students will gain an understanding of how these tools can find bugs which would otherwise likely be missed by manual peer review/inspection or testing activities.

1.2 TESTING TOOLS

The tool used for analysis in this lab is FindBugs version 2.0 [2]. We will be using FindBugs' Eclipse [3] plug-in but it is also available for other development environments such as NetBeans [4] and the command line. FindBugs analyzes a Java project on demand and provides a detailed report of the potential defects found and what they may mean to the developer.

1.3 SYSTEM UNDER TEST

The SUT for this lab is JFreeChart 1.0.14 [5]. JFreeChart is an open source Java framework for chart calculation, creation and display.

To get started with the JFreeChart system, download the "JFreeChart.zip" file from course page and extract the entire archive to a known location. More information on how to get started with these files will be provided in the familiarization stage. Note that the version of JFreeChart distributed for this lab is an actual release of JFreeChart.

2 FAMILIARIZATION

2.1 TOOL SETUP

2.1.1 Install FindBugs

1. Install Eclipse for Java Developers if you do not have a working installation
2. Launch Eclipse
3. Install the FindBugs' Eclipse plugin by navigating to "Help > Install New Software…" and enter: http://findbugs.cs.umd.edu/eclipse in the "Work with" field.
4. Check "FindBugs" and click "Next"

2.1.2 Add JFreeChart to Your Workspace

5. Unzip "JFreeChart.zip" to a known location on your hard drive
6. In Eclipse, navigate to "File > New > Project"
7. Select "Java Project from Existing Ant Buildfile" and click "Next"
8. Click "Browse…" and navigate to your JFreeChart folder

9. Navigate to the “ant” directory, select “build.xml” and click “Open”
10. Click Finish and rename the project if there is a naming conflict

2.2 RUNNING FINDBUGS

To become familiar with FindBugs, we will simply ask it to analyze JFreeChart.

11. Right click on the JFreeChart project in the package explorer
12. Select “Find Bugs > Find Bugs” from the context menu as below

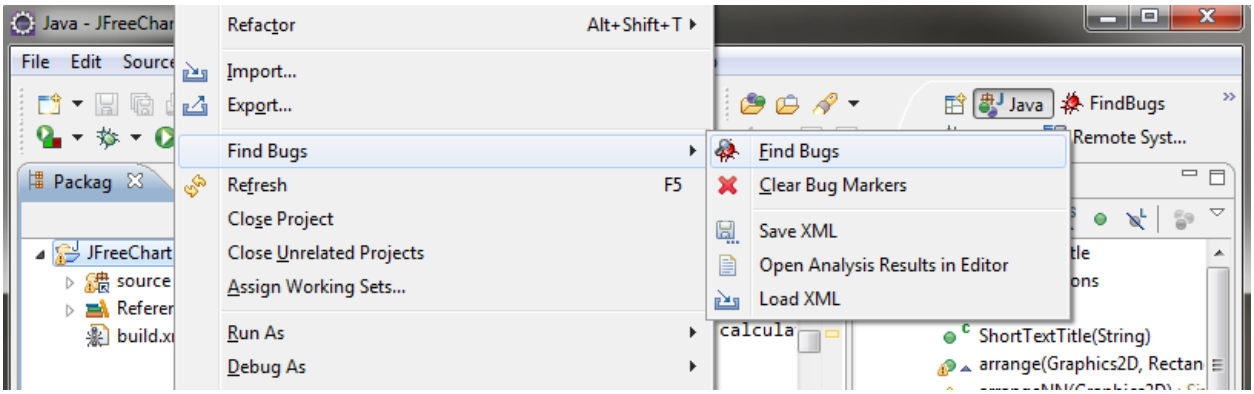

Figure 1 – The Find Bugs Menu

13. Wait for FindBugs to complete
14. Once FindBugs has completed, it will tell you the number of (potential) bugs it has found and ask you if you wish to switch to the FindBugs perspective, select “Yes” (you may go between your previous perspective and FindBugs at any time by selecting “Java” or “FindBugs” from the perspective choices in the top right of Eclipse’s window)

Figure 2 – FindBugs Completed Window

15. The FindBugs perspective provides you with several useful windows to work with as seen in Figure 3

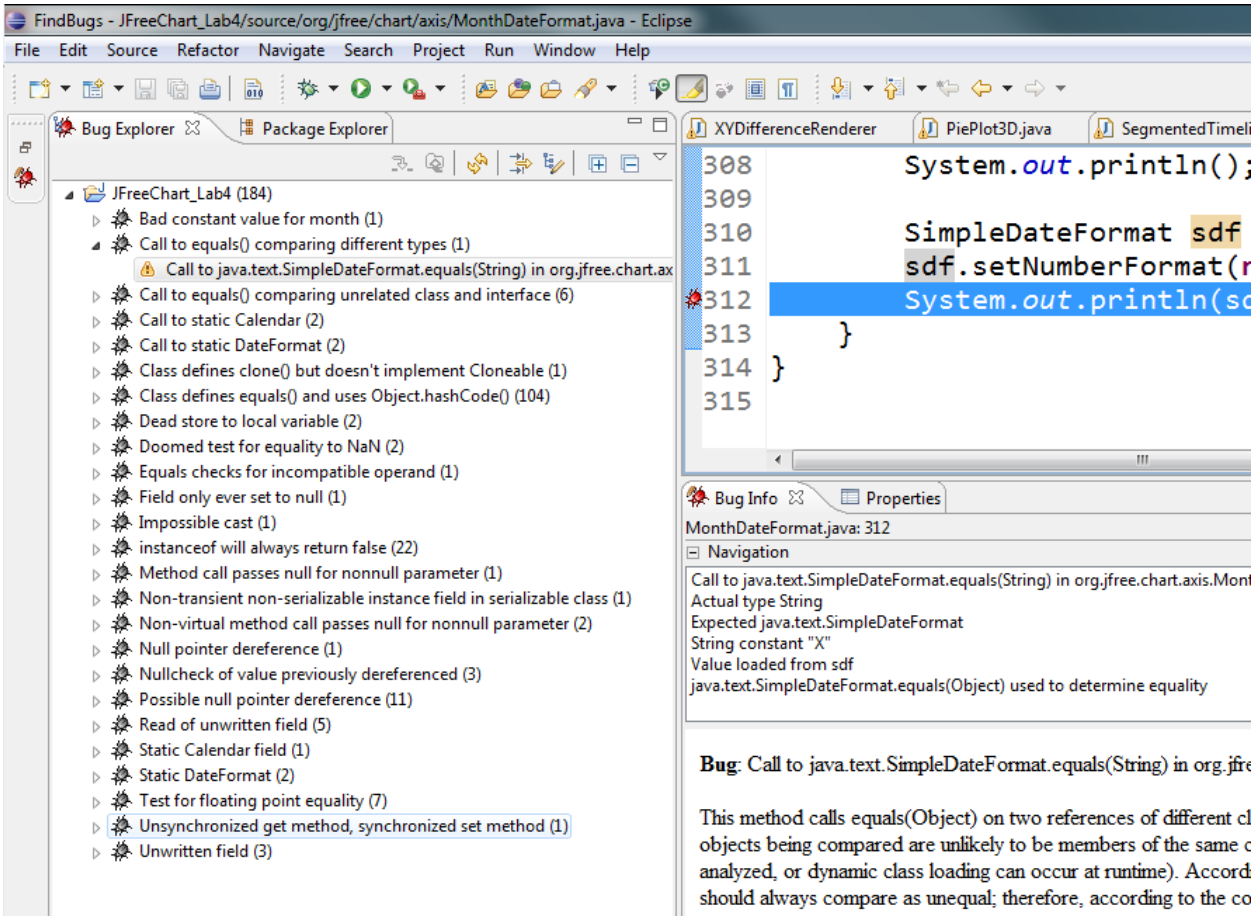

Figure 3 – The FindBugs Perspective

16. First take a look at the "Bug Explorer" view. You will notice that FindBugs organizes the (potential) bugs into several categories. Depending on the type of bug detectors you have chosen, you may see different list of bug types as it is shown in Figure 3 . 25 bug types are shown in Figure 3 . We will discuss the different bug detectors in Section 2.4.
17. Select and expand the bug category: "*Call to equals() comparing different types*". Clicking on each potential bug will load information about the bug into the "Bug Info" view incl*uding a short description of what the bug* means and how it is potentially a threat.
18. Double clicking on this occurrence will show you the code area including the potential fault which will also be denoted in the code by a small bug icon ( ) in the left margin

2.3 Analyzing an Issue

Let us now examine the issue "*Call to equals() comparing different types*"

19. Inspect the issue by clicking on the occurrence or by viewing the description on the FindBugs website: http://findbugs.sourceforge.net/bugDescriptions.html#EC_UNRELATED_TYPES.
20. FindBugs tells us that:

> *This method calls equals(Object) on two references of different class types with no common subclasses. Therefore, the objects being compared are unlikely to be members of the same class at runtime (unless some application classes were not analyzed, or dynamic class loading can occur at*

runtime). According to the contract of equals(), objects of different classes should always compare as unequal; therefore, according to the contract defined by java.lang.Object.equals(Object), the result of this comparison will always be false at runtime.

21. Examine the code to check that FindBugs correctly evaluated this bug. When you double click on the bug, the suspicious code will be shown as:

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy");
 sdf.setNumberFormat(null);
 System.out.println(sdf.equals("X"));
```

22. It is easy to see that FindBugs is correct in indicating that this code looks suspicious, however before confirming that this is indeed a bug, we should examine all possibilities.

23. The first thing that we must consider is that FindBugs may be incorrect in understanding the requirements of this method. We should consider that there is a possibility that this could be a correct use of `SimpleDateFormat::equals()` which perhaps deviates from the norm of most `Object::equals()` methods. We could imagine a situation where code such as the following could result in the same issue yet perhaps be a correct implementation:

```
public class MonetaryValue {

    protected int dollars;
    protected int cents;

    public MonetaryValue(int dollars, int cents)
    {
        this.dollars = dollars;
        this.cents = cents;
    }

    @Override
    public boolean equals(Object object) {
        if (object instanceof MonetaryValue)
        {
            MonetaryValue value = (MonetaryValue) object;
            return this.dollars == value.dollars && this.cents == value.cents;
        }
        if (object instanceof Double)
        {
            Double value = (Double) object;
            return value.equals(new Double(this.dollars + (this.cents / 100.0)));
        }
        return false;
    }

}
```

24. The above example may break some Java coding conventions but we should consider that such a case could be possible for the `SimpleDateFormat` object. Fortunately `SimpleDateFormat` is a native Java class so we can inspect the documentation easily at:

    http://docs.oracle.com/javase/1.4.2/docs/api/java/text/SimpleDateFormat.html#equals%28java.lang.Object%29

    The documentation does not mention any special cases for receiving a string as input, so we can be fairly confident that FindBugs is correct in identifying this as a misuse of the method `SimpleDateFormat::equals()`.

25. The second thing that we must consider is context: is there any possibility that this misuse is the intended implementation? On another quick examination of the above code, we can see that the suspect code is part of a method `main()`. Above the method declaration, on line 257, there is a code documentation simply stating: *"Some test code"* which clearly indicates to use that this

method exists as a test. If the intention is to test the `SimpleDateFormat::equals()` method then this could be the intended implementation, if not then this may again indicate an error.

26. In this particular example, we could see that this code may in fact be erroneous or may be the intended implementation. This then could lead us to believe that making the determination ourselves would not be possible without contacting the developers, which would be impracticable for this example. Choosing one of these three possibilities is up to you. However, we can see that using tools such as FindBugs in your own development projects may be useful.

2.4 FINDBUGS OPTIONS

27. FindBugs has a variety of options that you can customize and then re-run it. To view the FindBugs' options windows, right click on the project and choose Properties. Then choose FindBugs in the right-hand-side list, as shown in Figure 4.

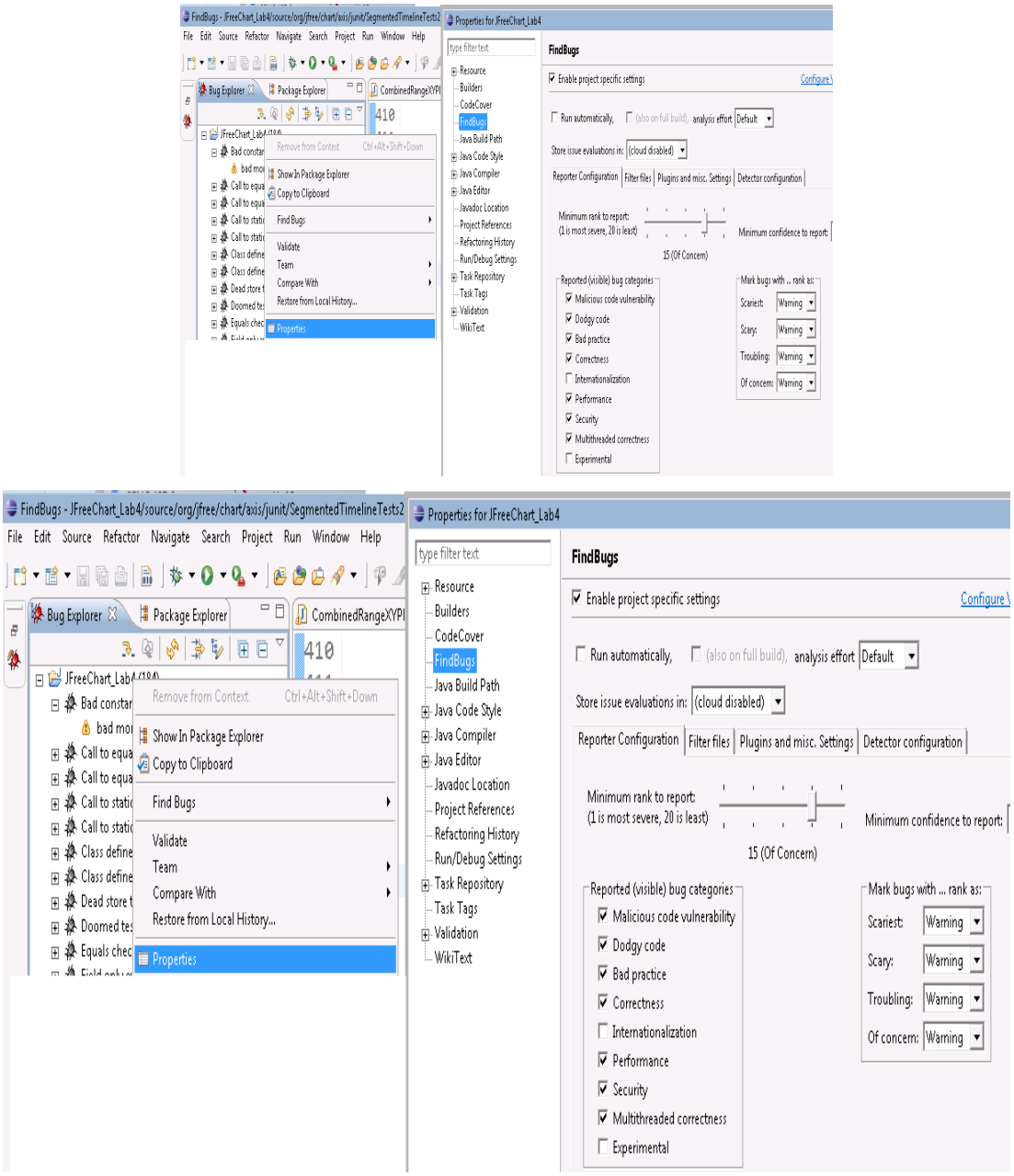

Figure 4– FindBugs' Options

28. The options window lists a variety of choices. To change settings from their default values, you need to choose "Enable project-specific settings". Of special importance is the list of reported (visible) bug categories. The popular bug categories include: bad practices, code vulnerability, and correctness. Enable all of the bug categories and then go to the tab "Detector Configuration" (as shown in Figure 5).

29. For example, let us choose the Detector *FindRefComparison* and read its details. As we can see, the pattern *EC_UNRELATED_TYPES* (finding bugs of type "*Call to equals() comparing different types*") is supported by this particular detector.

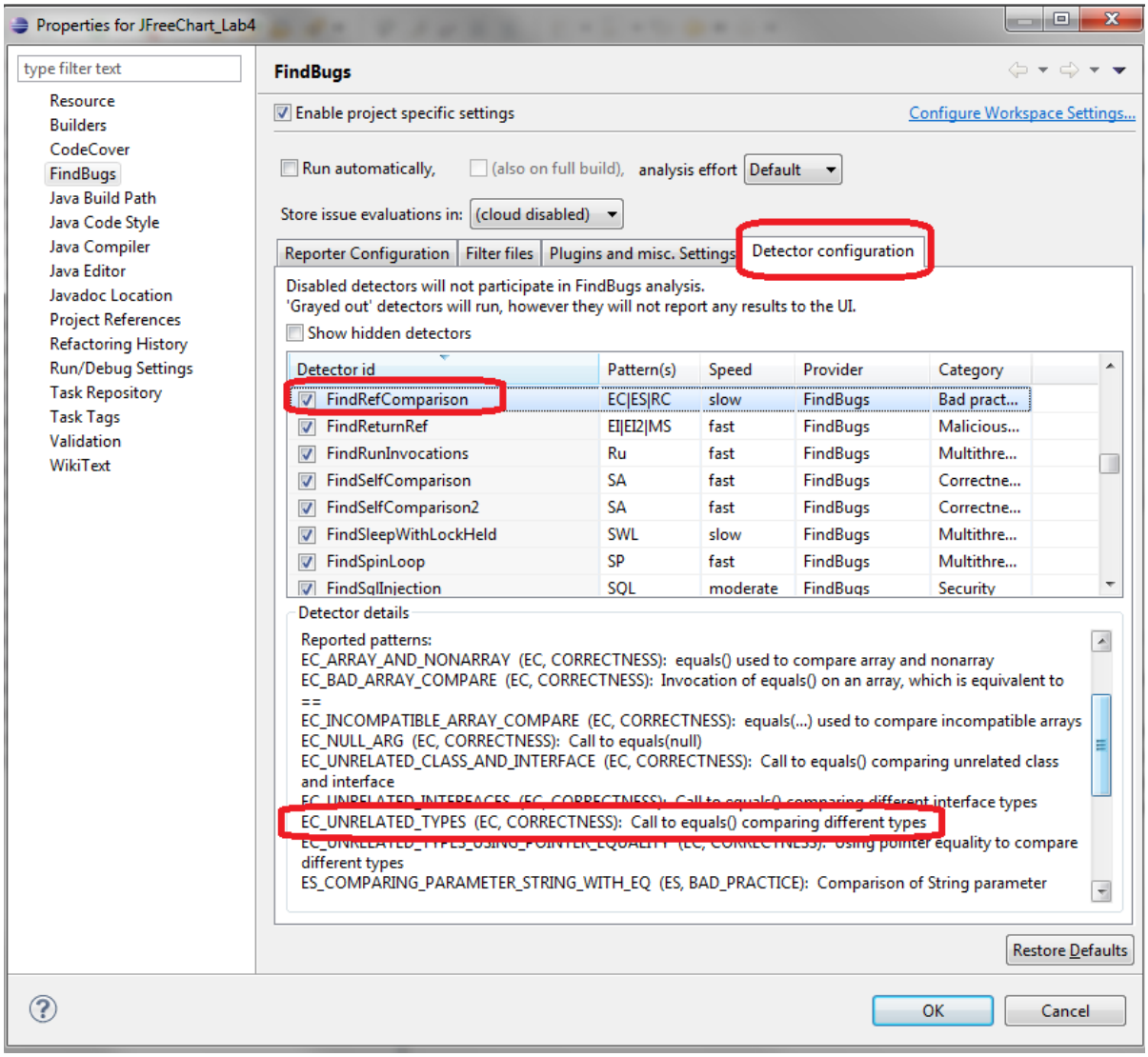

Figure 5 – FindBugs' Options-> Detector Configuration

30. Enable all of the bug categories in both the reported (visible) bug categories list and also in the detector ID list. Re-run FindBugs and review the list to find any new bugs found in the second run. Review, analyze and discuss a few of them with your lab partner.

2.5 RESOURCES

FindBugs has an on-going tutorials page at: http://code.google.com/p/findbugs-tutorials/ which you may find useful.

Also, you can find the descriptions of the bugs found by FindBugs in this URL:
http://findbugs.sourceforge.net/bugDescriptions.html

3 INSTRUCTIONS

31. To make your workload manageable, we would like you to analyze one potential issue from 8 different bug types reported by FindBugs, in Sections 3.1 and 3.2.

3.1 ANALYZE 7 BUGS UNDER TYPES OTHER THAN "POSSIBLE NULL-POINTER DEREFERENCE"

32. Choose 1 possible bug from each of 7 randomly chosen different bug types (except "Possible Null Pointer Dereference").
33. The 8th issue must be from the "Possible Null-Pointer Dereference" type (you will be required to expand on this in 3.2).
34. For these 7 bugs, analyze them by stepping through the code and give an overview of which issues you think are:
    a. Actual bugs:

       If you believe that the issue will cause a failure in runtime and should be fixed.

    b. Intended implementations:

       If you believe that the issue will not cause a failure and should not be fixed. This may come as a result if you believe that the potential threat that the issue raises

       - is so remote that it is not worth fixing,
       - could never be realized, or
       - is the intended implementation (throws an exception which will later be caught for example).

    c. Impossible or impractical to determine:

       It is possible that the bug which you are examining may take too much effort or may be otherwise impossible to determine if it will cause a failure or not. This may be a result of not having complete knowledge of the program's requirements or if you have spent enough effort stepping through the code to believe that making a final decision may not be worth the effort.
35. Explain the steps you took to reach you decision.

3.2 ANALYZE BUGS UNDER TYPES "POSSIBLE NULL-POINTER DEREFERENCE"

36. Choose random bug under bug types "Possible Null Pointer Dereference", as shown in Figure 6, and inspect them

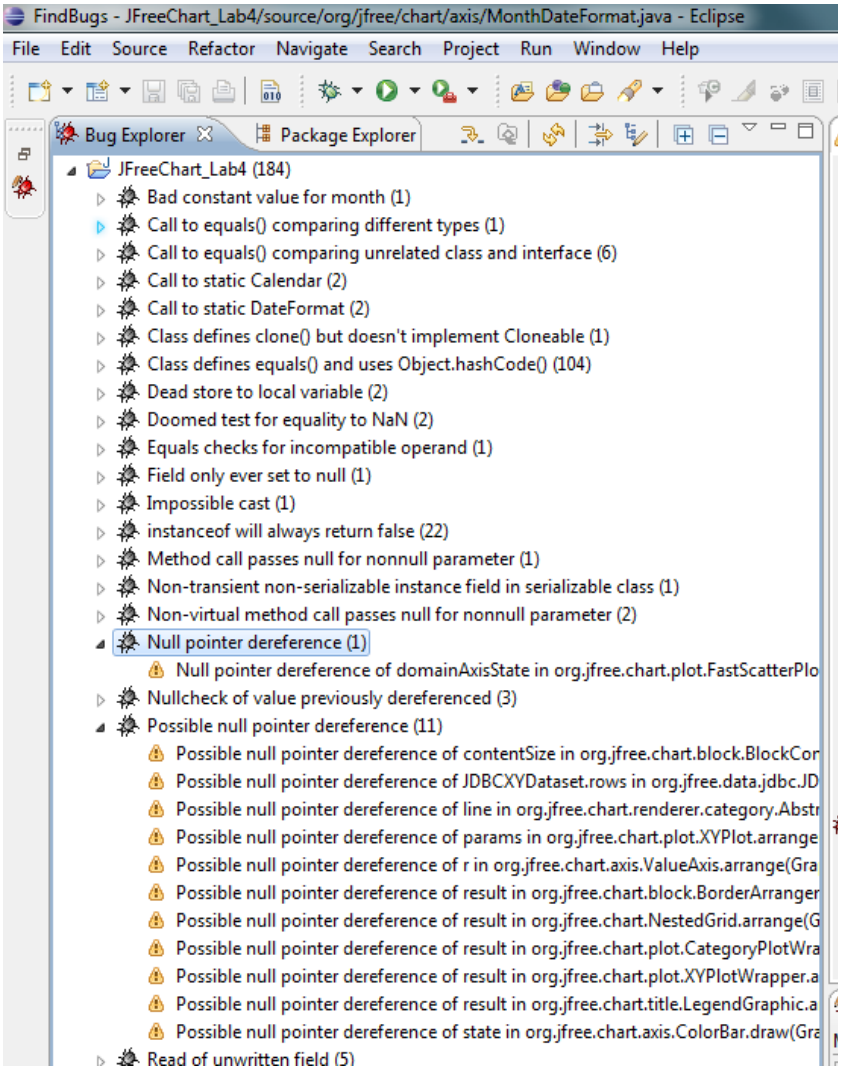

Figure 6 – Bug types "Possible Null Pointer Dereference" and "Null Pointer Dereference"

37. Explain what the potential fault is that FindBugs is trying to find, for that bug.
38. For that bug, explain how you would modify the code to fix bug warning or why you believe that you should not.

4 SUMMARY

After completion of this lab, students should have a general understanding of the concepts of static code analysis tools; how they can be beneficial and what their limitations are. Students should gain a general understanding of how these types of tools can play a role in the software verification/validation and that testing and static code analysis complement each other in achieving high-quality software.

5 DELIVERABLES AND GRADING

5.1 IN-LAB DEMONSTRATION OF RESULTS (10%)

You will be required to demonstrate your progress to the TA or the instructor during lab time.

Note: Both students of each group should be present in the demo times.

5.2 LAB REPORT (90%)

Students will be required to submit a report on their work in the lab as a group. In the report should be included:

Analysis of all 7 randomly-chosen Bugs (Section 3.1)	50%
Discussion and analysis of randomly-chosen bug of type "Possible null pointer dereference" (Section 3.2)	15%
Comparison of the benefits of static code analysis versus manual code inspection and testing. Could you for example find all the bugs by manual code inspection? What could be the benefit of manual code inspection over automated static code analysis? Give concrete (code) examples if needed	20%
Any difficulties encountered, challenges overcome, and lessons learned from performing the lab	5%
Comments/feedback on the lab itself. (Was it easy to follow? Too much/too little time? etc.) Please try to keep comments and feedback constructive.	5%

6 ACKNOWLEDGEMENTS

This lab instructions was originally performed by prof. Vahid Garousi from University of Calgary.

This lab is part of a repository of an open software-testing laboratory courseware available under a Creative Commons license for other testing educators at:

http://www.softqual.ucalgary.ca/projects/testing_labs

This courseware has been a team effort among Vahid Garousi (2008-), Michael Godwin (2011-12), Yuri Shewchuk (2008), Negar Koochakzadeh (2009), Christian Wiederseiner (2010-11) and Riley Kotchorek (2010). We would like to thank students of the courses SENG 437 and 521 in the last few years for their careful reviews and feedbacks on this set of lab manuals.

REFERENCES

[1] Wikipedia. Static program analysis. [Online]. http://en.wikipedia.org/wiki/Static_program_analysis

[2] FindBugs™ - Find Bugs in Java Programs. [Online]. http://findbugs.sourceforge.net/

[3] The Eclipse Foundation. [Online]. http://www.eclipse.org/

[4] NetBeans. [Online]. http://netbeans.org/

[5] JFreeChart. [Online]. http://www.jfree.org/jfreechart/