2009

Pidgin
Carlos Simões
Higino Silva
João Carlos Almeida
Miguel Graça Oliveira

[ANALYSIS ASSIGNMENT 10]

INTRODUCTION

The purpose of this project is to evaluate a testing tool chosen by the team and provide feedback about its usage, its advantages and its disadvantages.

THE TOOL

For this project, we are evaluating the tool FindBugs, version 1.3.8. The tool has its source code, as well as precompiled binaries, available at the FindBugs website [1].

EVALUATION PLAN

In order to evaluate the tool we will be run it through both previous work done by the team and a third party, open source, project. We will evaluate the results obtained and, through that evaluation, assess how accurate is the tool and how beneficial would be its usage on a software project.

TOOL DESCRIPTION

OVERALL VIEW

FindBugs is bug finder for Java programs. It uses static analysis to inspect Java byte code (compiled class files). This means that you do not even need the program's source code to analyze it, nor execute the program in order to detect bugs.

It is based on the concept of bug patterns. A bug pattern is a code idiom that is often an error. Bug patterns arise for a variety of reasons:

- Difficult language features
- Misunderstood API methods
- Misunderstood invariants when code is modified during maintenance
  Garden-variety mistakes: typos, use of the wrong Boolean operator

Thus, the tool inspects Java byte code for occurrences of bug patterns.

FindBugs is free software, available under the terms of the Lesser GNU Public License. It is implemented using Java, and can run on any virtual machine compatible with Sun's JDK 1.5. It can analyze programs written for any version of Java.

Nowadays, the analysis engine reports nearly 300 different bug patterns. FindBugs also supports a plug-in architecture allowing anyone to add new bug detectors. Therefore, with new bug detectors the number of bug patterns found by the tool can increase even more.

FindBugs uses the Byte Code Engineering Library (BCEL) to analyze Java byte code. The tool also supports bug detectors written using the ASM byte code framework. FindBugs uses dom4j for XML manipulation.

The tool's detected bugs are grouped into a number of categories (e.g. correctness, performance) as well as attributed priorities (high, medium or low) to each of the bug reports. These priorities are determined by heuristics within each detector/pattern and are not necessarily comparable across bug patterns.

FindBugs can be run from the command line, Ant or Maven, or in a GUI, Eclipse or NetBeans. The analysis results can be saved in XML, which can then be further filtered, transformed, or imported into a database.

HOW TO INSTALL

The installation of the tool is very simple no matter the flavor. In the context of this project, we are using the Swing interface version.

In order to install the tool, just follow these steps:

1. Go to the tool download page (http://findbugs.sourceforge.net/downloads.html);
2. Select the swing interface version you want to install in your machine;
3. After download it, double-click on the file you just download and it is ready to use.

HOW TO USE IT

To begin using the tool, you need to create a project. To do that you can go to the option menu "File\New Project", or simply type CTRL+N.

A new screen will open requiring a project name and some others information. Figure 1 shows the screen to be fulfilled.

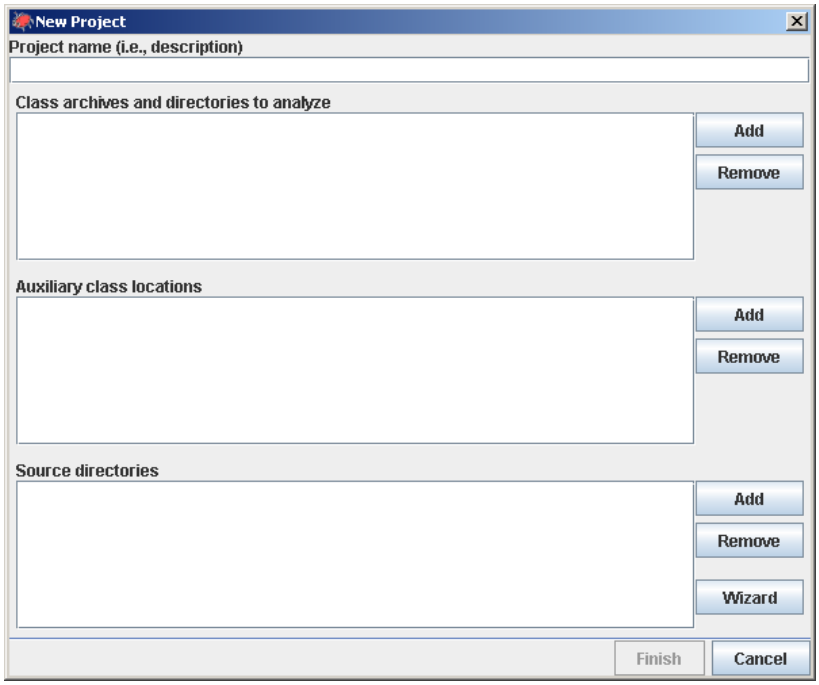

FIGURE 1 - SCREEN TO CREATE A NEW PROJECT ON THE TOOL.

On the project name field, you are able to input a small description that identifies to project your analyzing. Under the "Class archives and directories to analyze", you should provide the directory containing the java byte code to be analyzed (usually the "/bin" directory on the project). To do that, click on the Add button and inform the directory containing the application byte code.

In case you also have the source code and you would like to see it on the tool while browsing throughout detected bugs, you can inform where is located the application source code. To do that just click on the Add button under "Source directories" and inform the directory where the source code is located.

In addition to this, please note that we are able to add external libraries (in the Auxiliary class locations section) and add multiple directories for each of the project directory categories.

After fulfilling this information, click on "Finish". At this point, the tool will start analyzing the application provided. The screen now should be similar to the one shown on Figure 2.

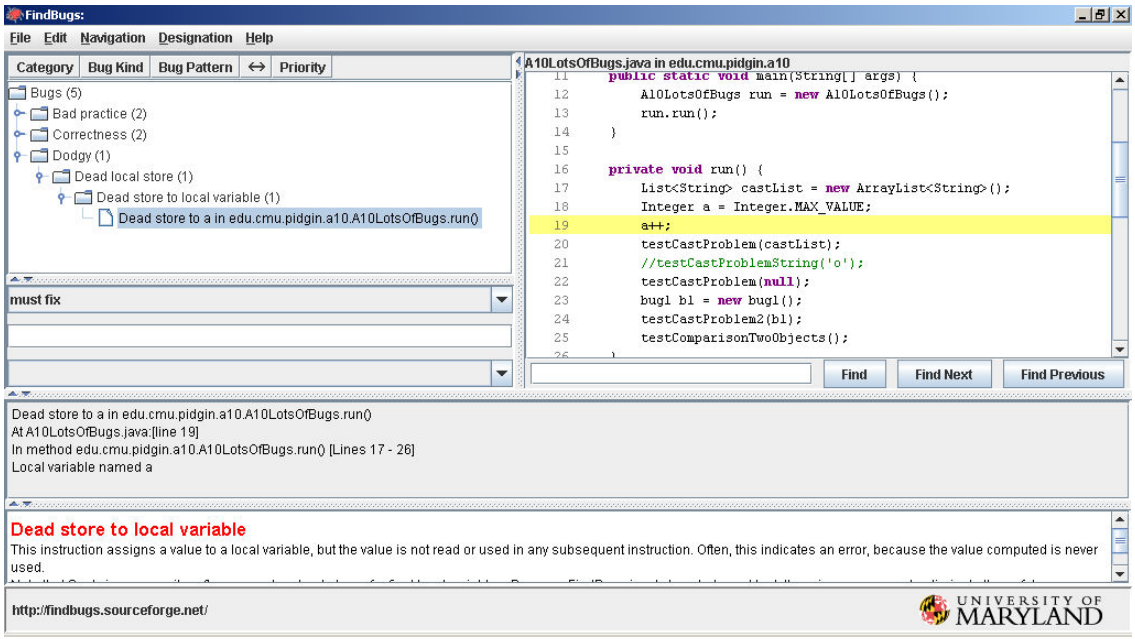

FIGURE 2 - MAIN SCREEN AFTER FINDBUGS ANALYSIS ON APPLICATION BYTECODE.

This screen has three major sections that are provided to support main tool operations. The first one contains a list of detected bugs by the tool and an option to categorize this bus according to the options provided by the tool. Figure 3 shows this screen section.

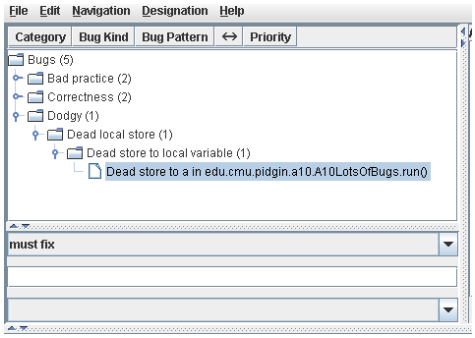

FIGURE 3 - BUG LIST SECTION FROM THE TOOL MAIN SCREEN

The right section on the main screen shows the source code where a bug was detected. When a bug is selected from the list provided on the bug list (shown on figure 3), the tool opens the corresponding source code on this screen. Figure 4 show this code section.

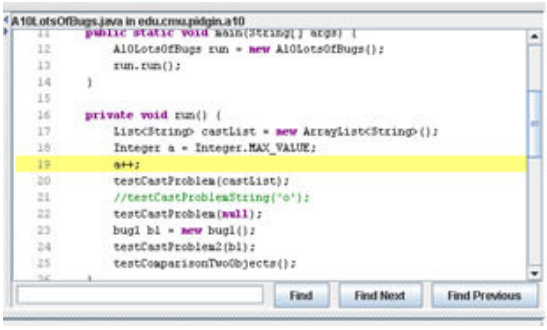

FIGURE 4 - CODE SECTION FROM THE TOOL MAIN SCREEN.

The lower section on the main screen presents the details about the bug selected from the bug list. It also shows the line within the source coude were this bug was found. Figure 5 shows this bug information section.

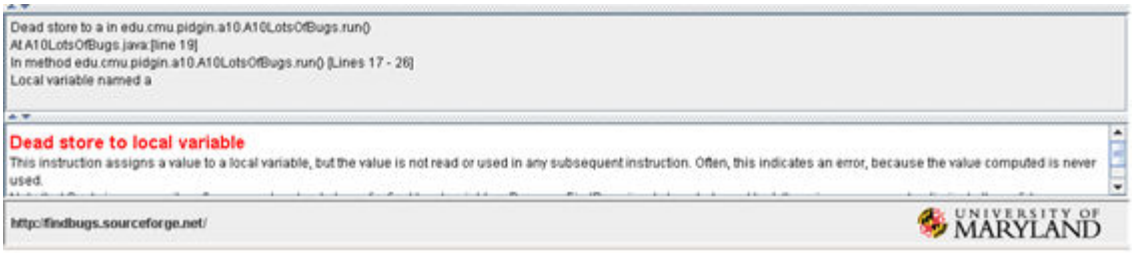

FIGURE 5 - BUG DETAIL SECTION FROM THE TOOL MAIN SCREEN.

After the tool completing the report with the all the found bugs, we are able to assess these results, categorize the bugs found and also export these results to further analyzes and reports.

For more information about how to use the tool, you can go to the on-line user manual for the tool available at http://findbugs.sourceforge.net/manual/index.html [1].

TOOL EXPERIMENTS

In order to follow the evaluation plan we chose two different projects to run the tool on and analyze its results. The two selected projects in this work were the **Plural project** and the **Game checkers project** (this project includes both GUIs, the Framework and the game plugin, which was developed by our team).

The assessment done on the experiments will follow the following process:

1. Once with the projects source code available, we start tool analyzes on the code;
2. The list of bugs produced by the tool shall be assessed to looking for: false positives, true positives irrelevant to the project and true positives relevant to the project.
3. Log the results of this assessment.

RESULTS

After using the tool and assessing the bugs found we came up with the results presented on Table 1.

TABLE 1 - DEFECT BREAKDOWN

	Plural	Checkers game
Total defects	68	45
False positives	2	1
Defects/KLOC	2.44	12.86
Defects not relevant	8	10

Defects found for each project are grouped by category as described on the following figures. Figure 6 shows the defects found on Plural Project

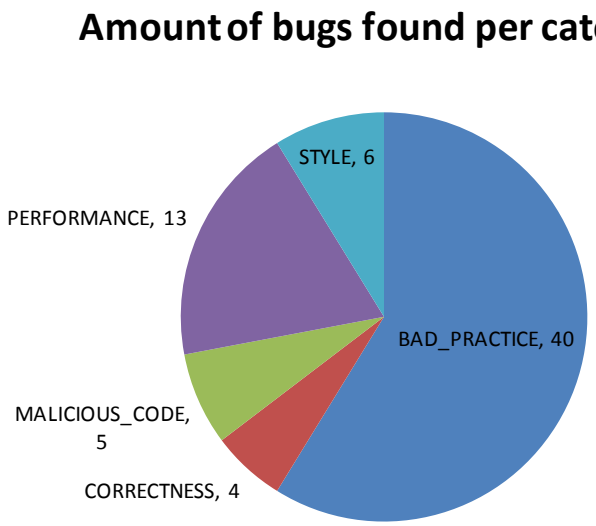

FIGURE 6 - AMOUNT OF BUGS FOUND PER CATEGORY

We can see from this graph that most of the defects found o Plural was due to Bad Practices (corresponding to 59% of the defects found).

Figure 7 shows the defects found for Checkers game.

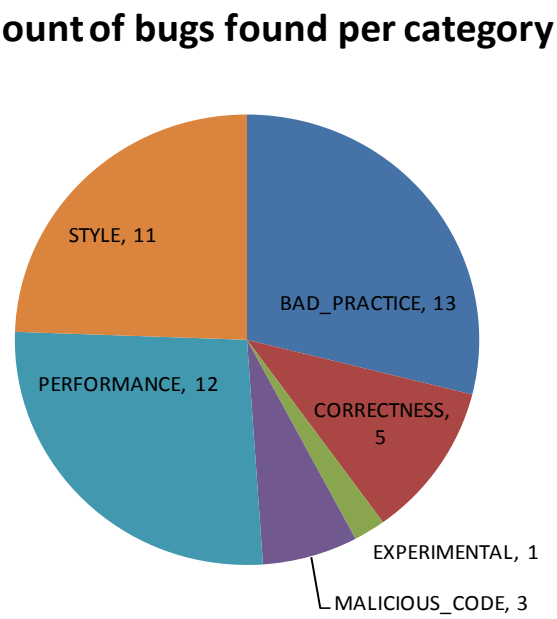

FIGURE 7 - AMOUNT OF BUGS FOUND PER CATEGORY

On the Checkers game project we can see that style, bad practices and performance bugs are responsible to most of the defects found by the tool.

TRUE POSITIVES RELEVANT TO THE PROJECT

The true positives releavant to the project are the total bugs found by the tool minus false positives and irrelevant true positivies. With these in mind, the true positives relevant to each project are seen on Table 2.

TABLE 2 - AMOUNT OF DEFECTS BY PROJECT

Project	Amount of defects
Plural	58
Checkers	34

In the following sections, we present some examples that provide a sample for the defects found by the tool.

PLURAL

DEFECT: METHOD WITH BOOLEAN RETURN TYPE RETURNS EXPLICIT NULL

This code has some methods that return Boolean objects as results. This can be seen on figure below, where we have for instance the method: public Boolean visit (BinaryExpAP binaryExpr).

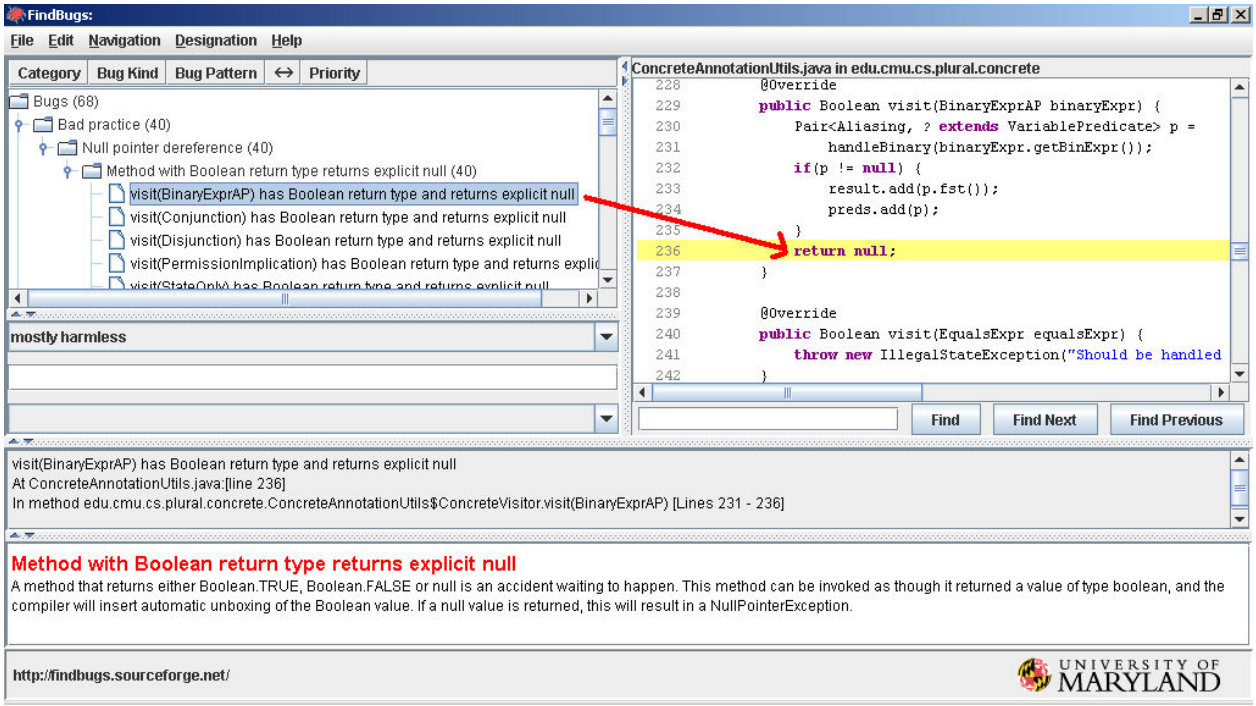

FIGURE 8 - RETURN EXPLICIT NULL

This is a true positive because, as exposed by the tool, because this method can be invoked as though it retuned a value of type Boolean, and the compiler will insert automatic unboxing. Thus, return null can cause a NullPointerException.

DEFECT: METHOD USE THE SAME CODE FOR TWO BRANCHES

This example code has a method in which perform some value verification using the syntax: <Condition> : <do this if Condition is true>, <do that if Condition is false>.

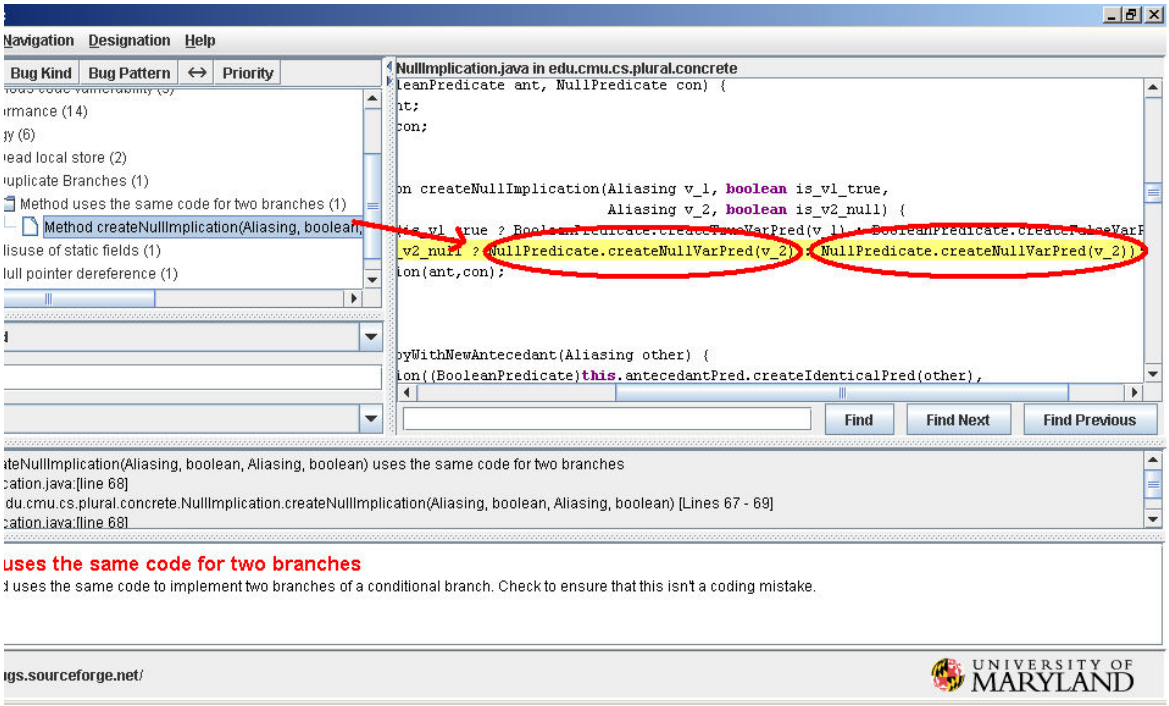

FIGURE 9 - WINDOW WITH THE BUG IDENTIFIED

We are able to confirm, from Figure 9 that this is a true positive since both actions on the then-clause and the else-clause are the same.

CHECKERS

DEFECT: COMPARISON OF STRINGS USING ==

This bug has an if-statement where the comparison is between two objects using the operator ==. This means that we are not comparing the content of the objects but references, which, of course, is a bug in the context of the program.

We are able to confirm, from Figure 10, that this is a true positive.

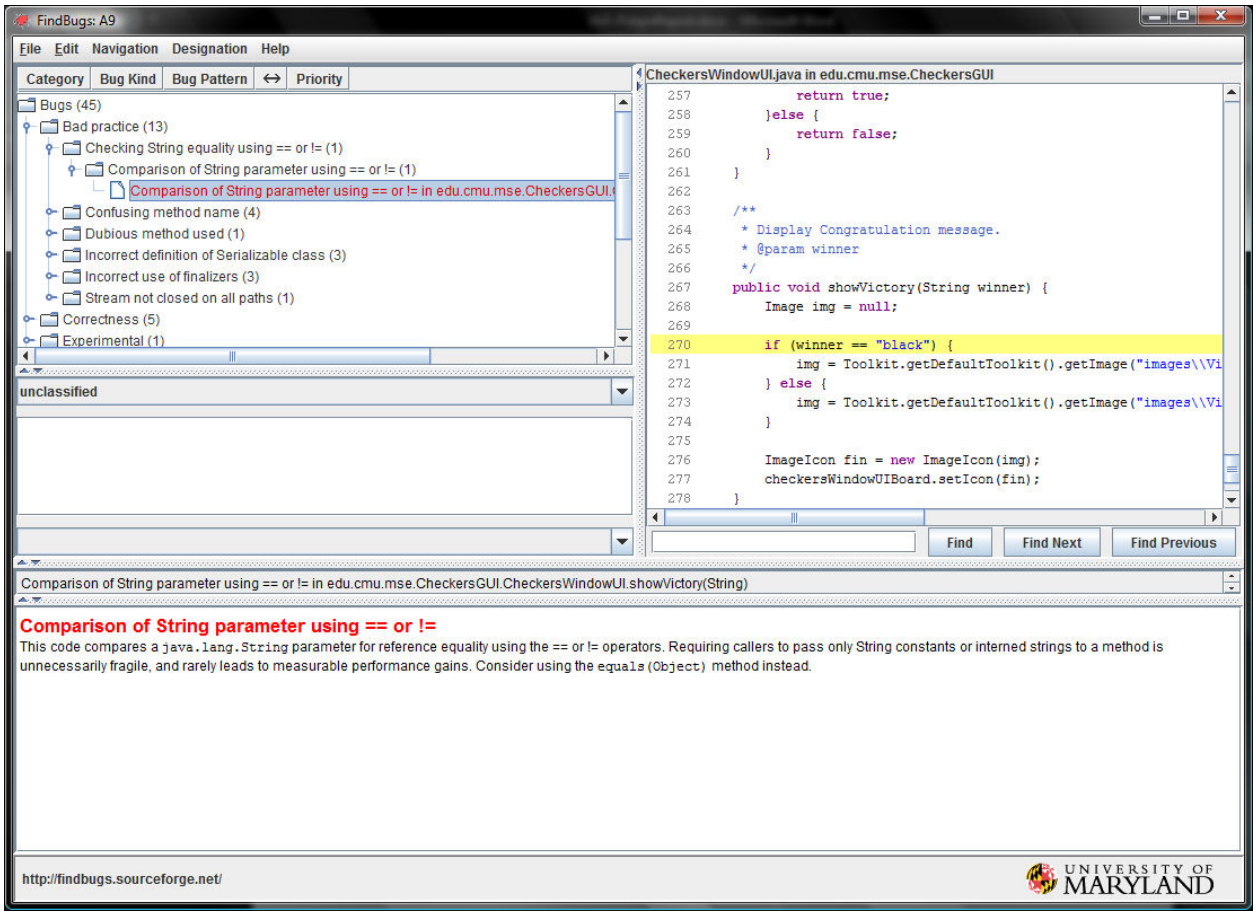

FIGURE 10 - COMPARISON OF STRING PARAMETER USING ==

DEFECT: CLASS DEFINES FIELD THAT MASKS A SUPERCLASS FIELD

This class defines a field with the same name as a visible instance field in a superclass. This is confusing, and may indicate an error if methods update or access one of the fields when they wanted the other.

In this case, this is a bug because the framework is defining the attribute game, which has certain properties, and, when redefined on the instantiation of the class will lead to errors as some methods use the parent object and others use the local object.

We are able to see the game redefinition on Figure 11.

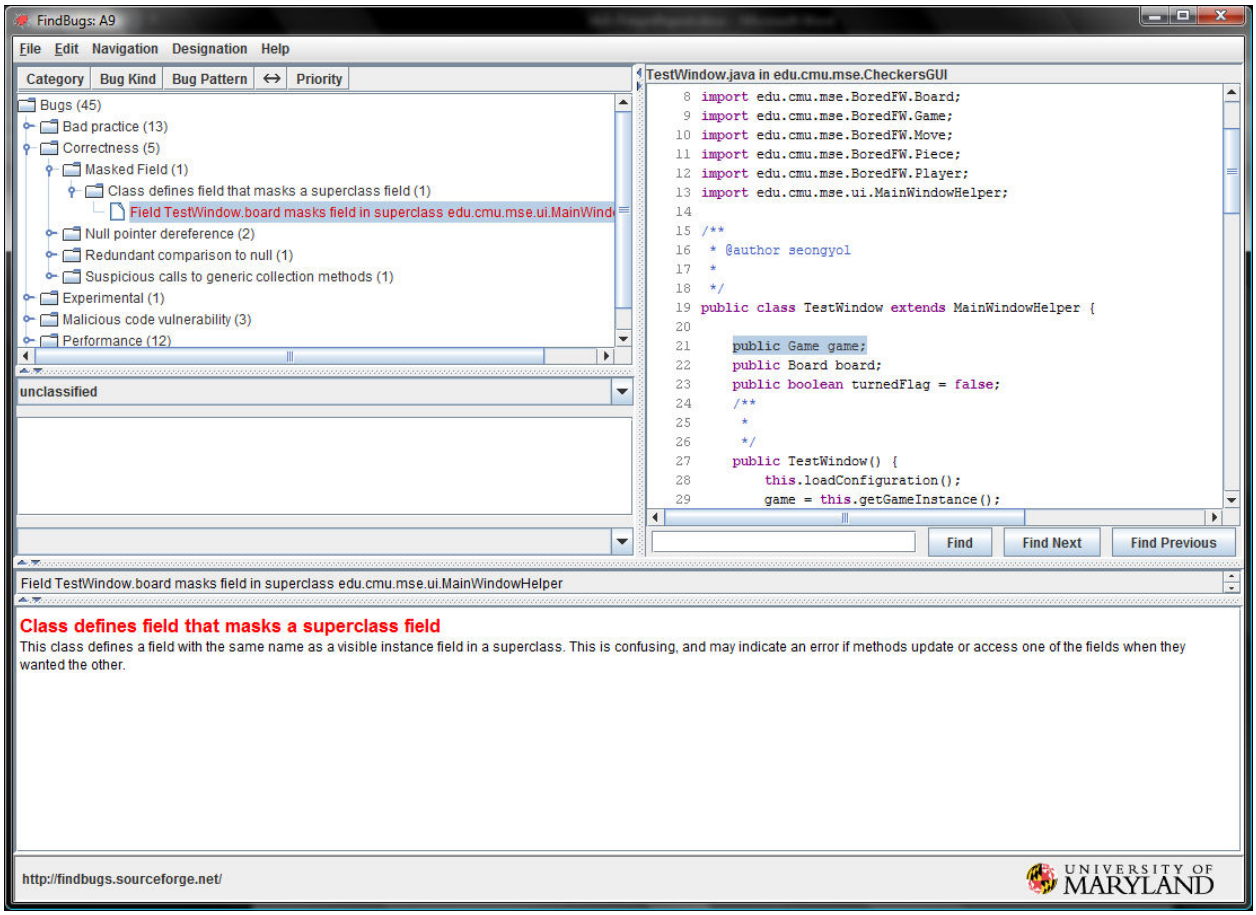

FIGURE 11 – CLASS DEFINES FIELD THAT MASKS A SUPERCLASS FIELD

TRUE POSITIVES NOT RELEVANT TO THE PROJECT

Assessing the results provided by the tool we realize that some code, while true positives where irrelevant to the project. From the Plural project, for instance, we found that many defects on code tagged as “deprecated” was counted by the tool. This should not be relevant to the project since these lines of code are no longer important. The following picture shows this example taken from the tool.

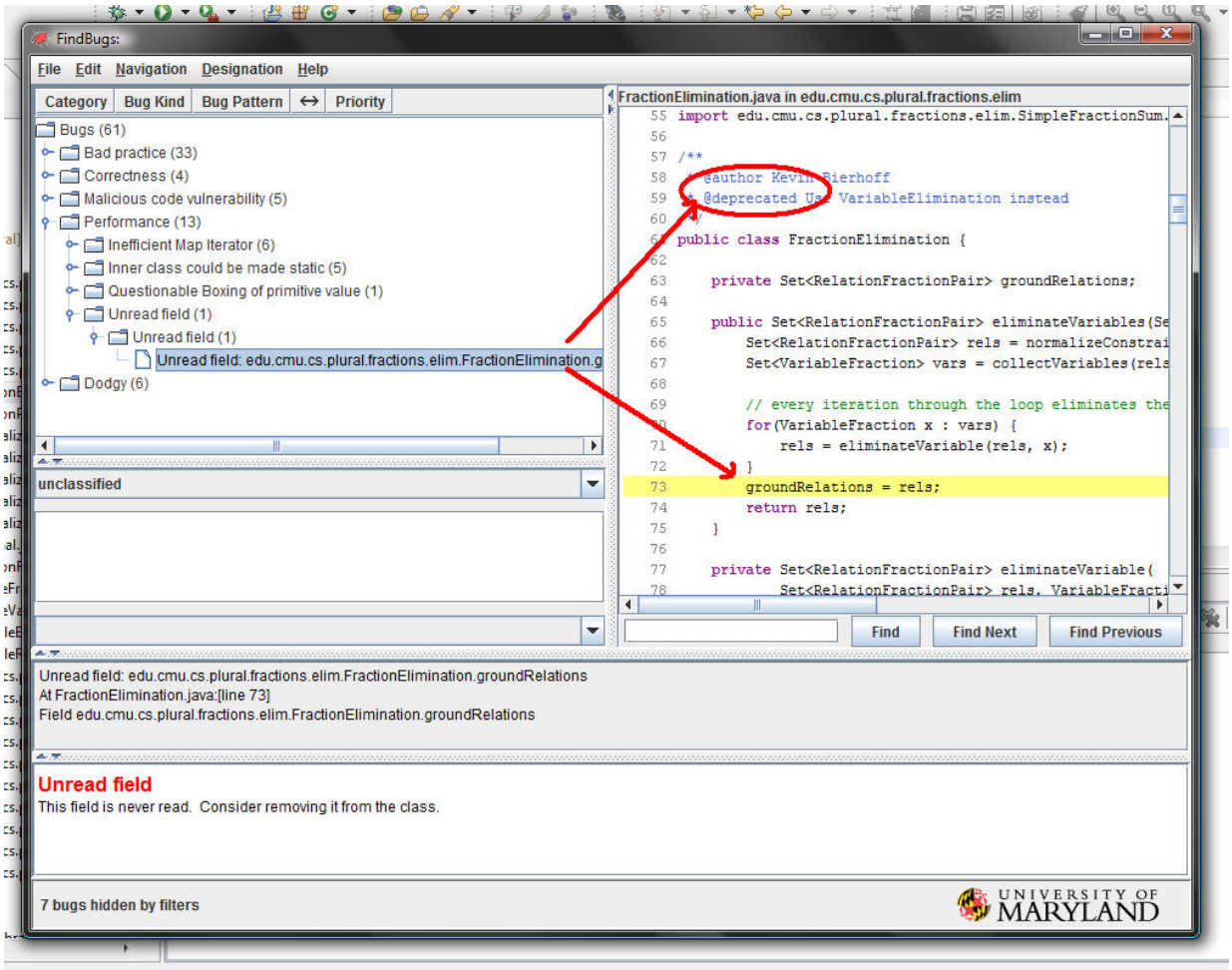

FIGURE 12 - UNREAD FIELD

FALSE POSITIVES

Regarding false positives, we could conclude that only few false positives were found on the projects analyzed by the tool. Here are some examples on the false positives found by the team.

PLURAL

DEFECT: DEAD STORE TO LOCAL VARIABLE

This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used. Note that Sun's javac compiler often generates dead stores for final local variables. Because FindBugs is a byte code-based tool, there is no easy way to eliminate these false positives. Figure 13 shows the defect found by the tool.

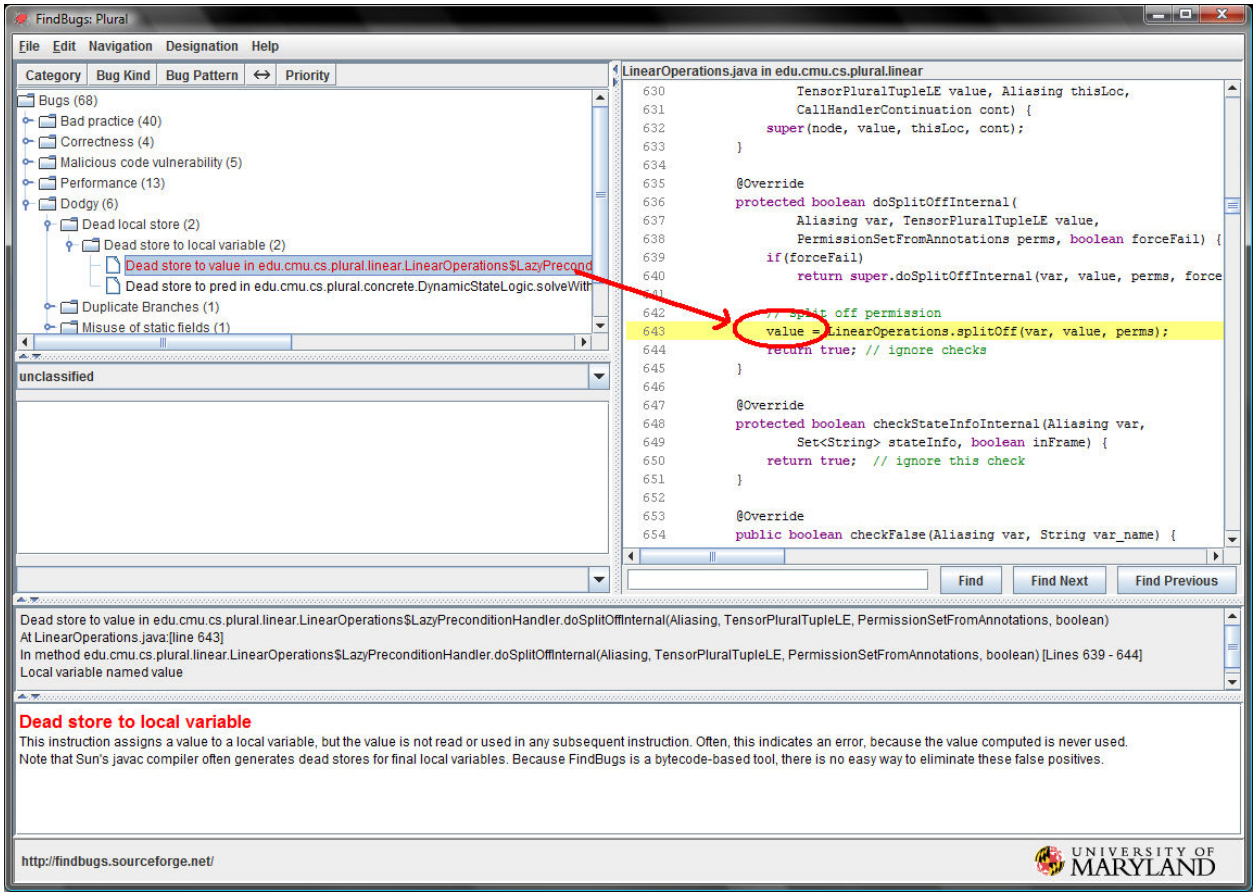

FIGURE 13 - DEAD STORE

This error is a false positive because the variable being identified as a dead store - value - is in fact a parameter of the method. This, of course, means that any operation, on the scope of the method, performed on that variable, are changes made to the variable that was passed as a parameter on the scope of the parent method.

CHECKERS

DEFECT: NULL VALUE IS GUARANTEED TO BE DEREFERENCED

A statement or branch if executed guarantees that a value is null at this point, and that value that is guaranteed to be dereferenced (except on forward paths involving runtime exceptions). This defect can be seen on the Figure 14.

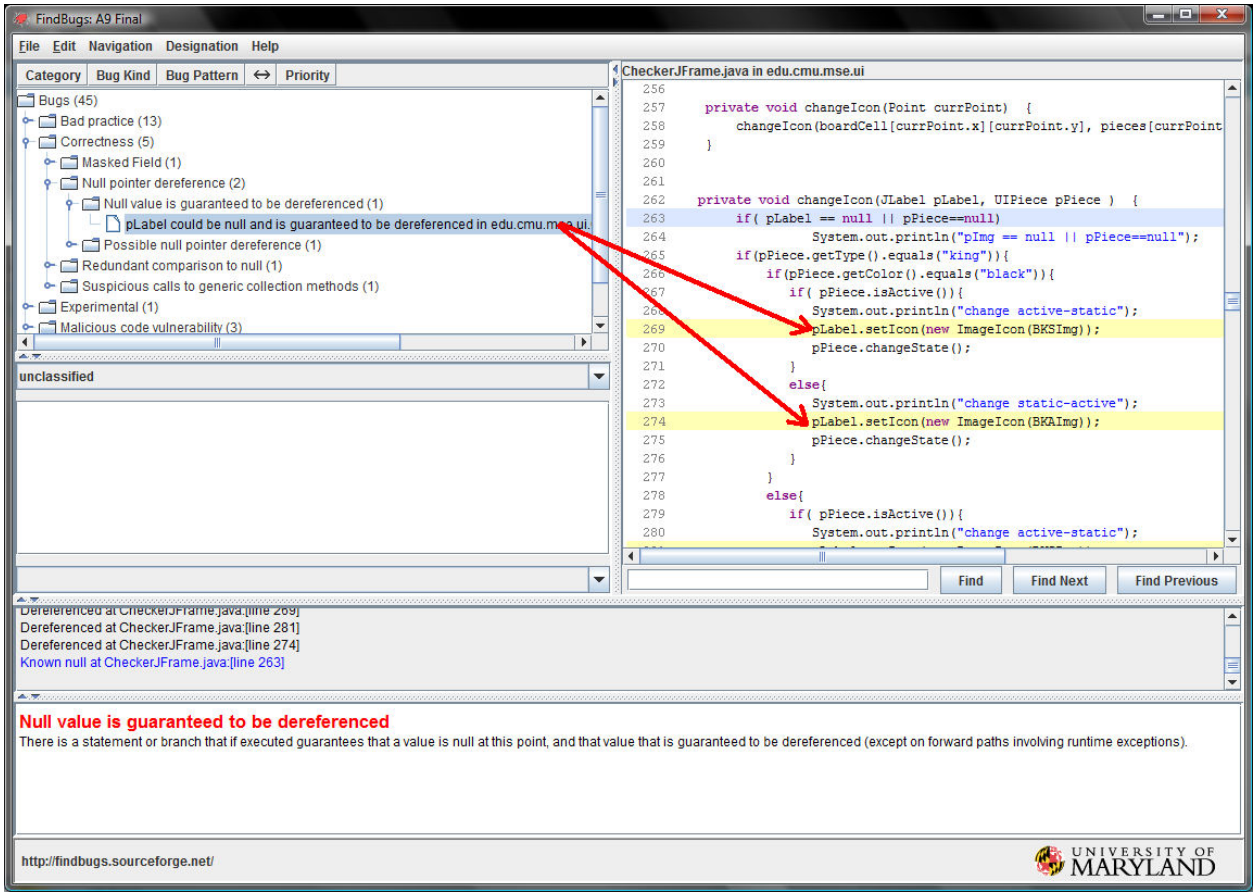

FIGURE 14 - GUARANTEED NULL DEREFERENCE

For this project, FindBugs discovered what the tool says is a guaranteed null dereference. This, however, is a false positive. Even though the variable is possibly null (this, in fact, is identified as a different bug), we do not have the guarantee that it is null at any point of the method, except on the body of the if-clause in question (line 263).

CONCLUSIONS

The tool is very easy to use. After installing the program in your machine, you should be able to use it to look for bugs in a Java code.

When using the tool to detect bug on code we achieve and ratio of defect/hour much higher from our ratio obtained on previous tests work (average of 2 defects per hour) . Considering the size of the LOC on the project being tested, the number of defects found and the time spent by the tool, it was certanlly worthy using the tool.

Nevertheless, manual inspection by humans is not limited to predefined algortihm or patterns. During code inspections process different bug patterns can arise, therefore finding different types of errors that an automated tool could not find. Although with less bug/hour ratio than an automated tool, code inspections and other kind of tests are still necessary in order to improve software quality.

We intend to use FindBugs in our studio project before doing our unit testing. We expected that this approach can help us improve software quality and improve our defect/hour ratio, reducing the number of defects during unit testing.

The FindBugs tool allows the user to filter the bugs reported using several customizable filter.

It is possible to customize the following filters:

- Version
- Priority
    - High
    - Medium
- Class
- Package
- Package prefix
- Category
    - Bad practice
    - Correctness
    - Malicious code vulnerability
    - Performance
    - Dodgy
- Designation
    - needs further study
    - not a bug
    - mostly harmless
    - should fix
    - must fix
    - bad analysis
    - unclassified
    - obsolete/deprecated/unused code; will not fix
- Bug kind
    - Bad casts of object references
    - Dead local store
    - Dubious method invocation
    - Duplicate Branches
    - Inefficient Map Iterator
    - Inner class could be made static
    - Method returning array may expose internal representation
    - Misuse of static fields
    - Mutable static field
    - Null pointer dereference
    - Questionable Boxing of primitive value
    - Suspicious calls to generic collection methods
    - Switch case falls through
    - Unread field
- Bug pattern
    - Impossible cast
    - Method uses the same code for two branches

- Dead store to local variable
- Invocation of toString on an array
- Method invokes inefficient Number constructor; use static valueOf instead
- May expose internal representation by returning reference to mutable object
- No relationship between generic parameter and method argument
- Field isn't final and can't be protected from malicious code
- Field should be package protected
- Field isn't final but should be
- Method with Boolean return type returns explicit null
- Load of known null value
- Switch statement found where one case falls through to the next case
- Should be a static inner class
- Write to static field from instance method
- Unread field
- Inefficient use of keySet iterator instead of entrySet iterator

All these filters can be combined so that the user will only see the bugs he finds important.

A filter configuration can be saved to a file to be reused. For example, a filter configuration file can be created by the enterprise Quality department and then spread across the development teams to enforce an enterprise wide quality plan.

The filter configuration can also be very handy to teach Java best practices to people that is starting to program in Java.

For example, the bug category “Bad practice” reports bugs related with code conventions, calling dangerous methods, badly written finalizers, and so on. These bug warnings are great to enforce good practices in two members of our team, which are learning or reminding the language. While they can be proficient in the language (because they have knowledge of C#), there are some common language glitches that require a certain amount of practice to learn. Alternatively, require a mentorship by some Java expert.

By using FindBugs, these developers can quickly learn to avoid certain bad practices.

On the other hand, seasoned Java developers might already know these bad practices, but might need to make use of some of them for some reason, like for example to tweak performance. Using the FindBugs filters, they can turn off that category of bugs.

Other bugs might not be considered errors by experienced Java users. These can also be turned off.

All the other category of warnings are very useful, and should not be turned off.

REFERENCES

[1] , FindBugs. *FindBugs.* [Online] [Cited: April 7, 2009.] http://findbugs.sourceforge.net/.