

# Evaluation of FindBugs

---

The static analysis tool that finds bugs

**Square Root**

**4/8/2009**

## Overview of FindBugs

FindBugs is an open source program which employs static analysis to indentify a multitude of potential errors in Java programs. The unique nature of this tool is that performs its analysis on byte code, rather than source code. Its installation and use will be explored in subsequent sections of this paper.

FindBugs can detect the bug patterns shown in the following table.

| Description   |
|---|
| AM: Creates an empty jar file entry   |
| AM: Creates an empty zip file entry   |
| BC: Equals method should not assume anything about the type of its argument   |
| BC: Random object created and used only once                                  |
| BIT: Check for sign of bitwise operation                                      |
| CN: Class implements Cloneable but does not define or use clone method        |
| CN: clone method does not call super.clone()                                  |
| CN: Class defines clone() but doesn't implement Cloneable                     |
| Co: Abstract class defines covariant compareTo() method                       |
| Co: Covariant compareTo() method defined                                      |
| DE: Method might drop exception   |
| DE: Method might ignore exception   |
| DP: Classloaders should only be created inside doPrivileged block             |
| DP: Method invoked that should be only be invoked inside a doPrivileged block |
| Dm: Method invokes System.exit(...)   |

## Application of FindBugs

FindBugs is a tool that is available for utilization in two flavors:

- Standalone application
- Eclipse plug-in

Since anything but the simplest projects can quickly overwhelm developers of an organization, and because Eclipse was also being utilized for the development of our Studio project, it was quite apparent that our proclivity would be towards exploration of the Eclipse plug-in rather than use of the tool as a standalone application.

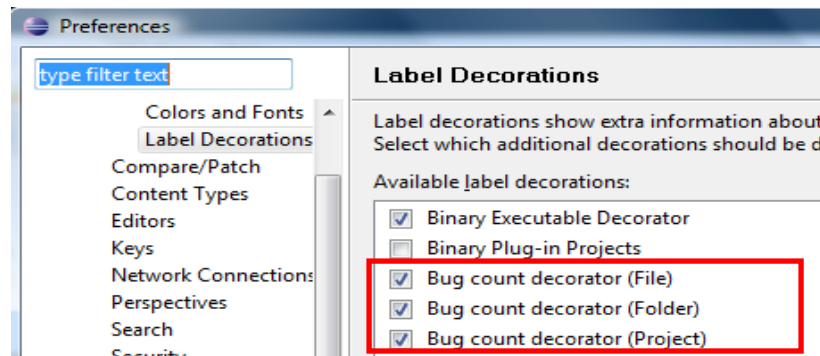
Following is an exposition of the steps required for the installation of the tool:

- The plug-in was available from the site: <http://findbugs.cs.umd.edu/eclipse>. Installation of the tool requires the following sequential operations:
  - Select the Help menu
  - Click on Software Updates

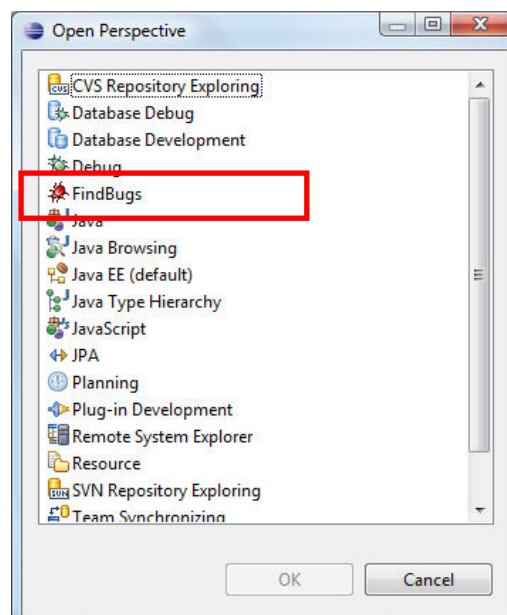
- Click on the tab labeled ‘Available Software’
- Upload the link for the aforementioned site
- Click Install, the process of which will necessitate a restart of the application

Following is an exposition of the steps that were necessitated for the execution of the tool:

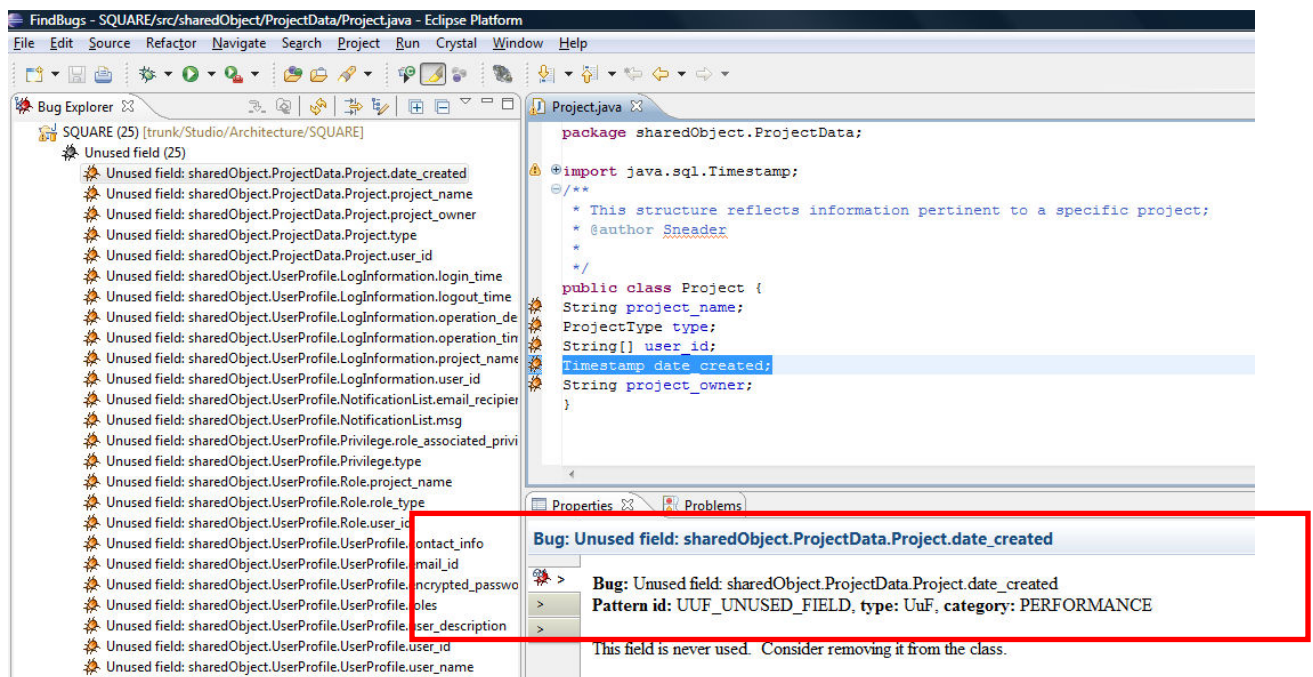
- Commencement of static analysis by the tool can be achieved by right clicking on either the:
  - The Java project
  - The Java package
  - The Java class
- Once the tool is executed the results can be viewed. A prerequisite for observing the results is the enabling of **label decorations** for the project/package/file. To carry out this operation:
  - Go to **Window->Preferences->General-Appearance->Label Decorations**
  - Enable the following Check Boxes



- The results can be viewed by opening the **FindBugs** perspective. To do this select **Window->Open Perspective->FindBugs**



- Following is an example of the results that were obtained for a specific project:

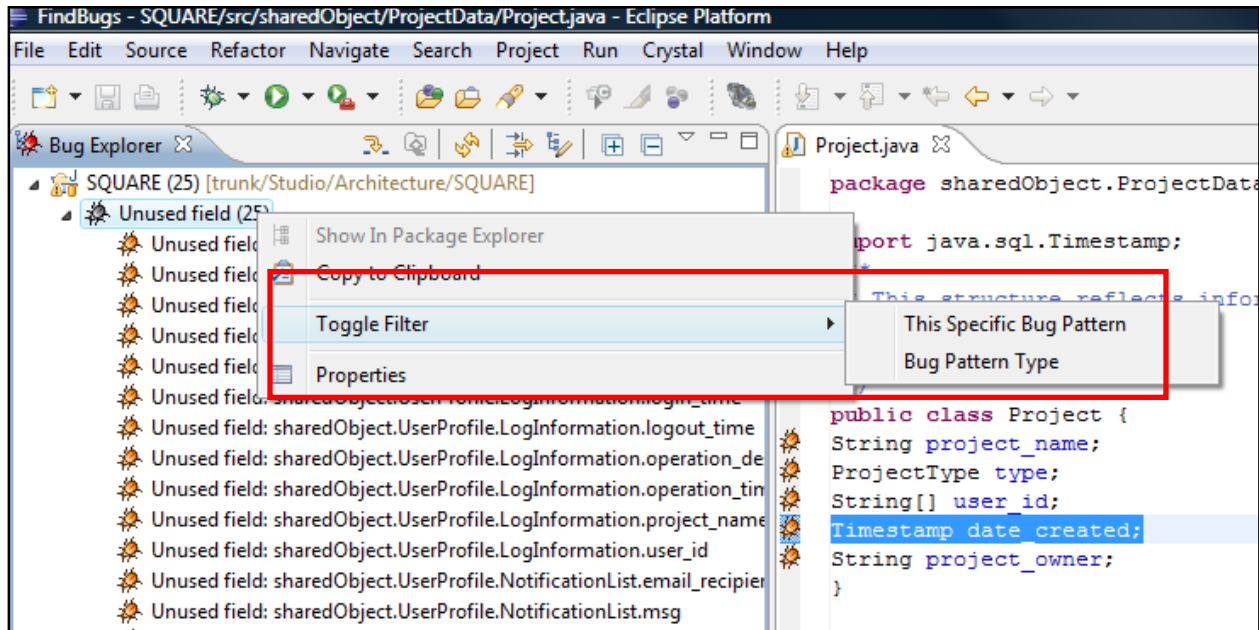


By default, the tool performs a large variety of analyses before yielding results. Consequently, it isn't uncommon for the tool to throw up a large number of potential bugs, the magnitude of which can quickly impede the analysis of the results. To overcome these problems, users of the tool are faced with two options:

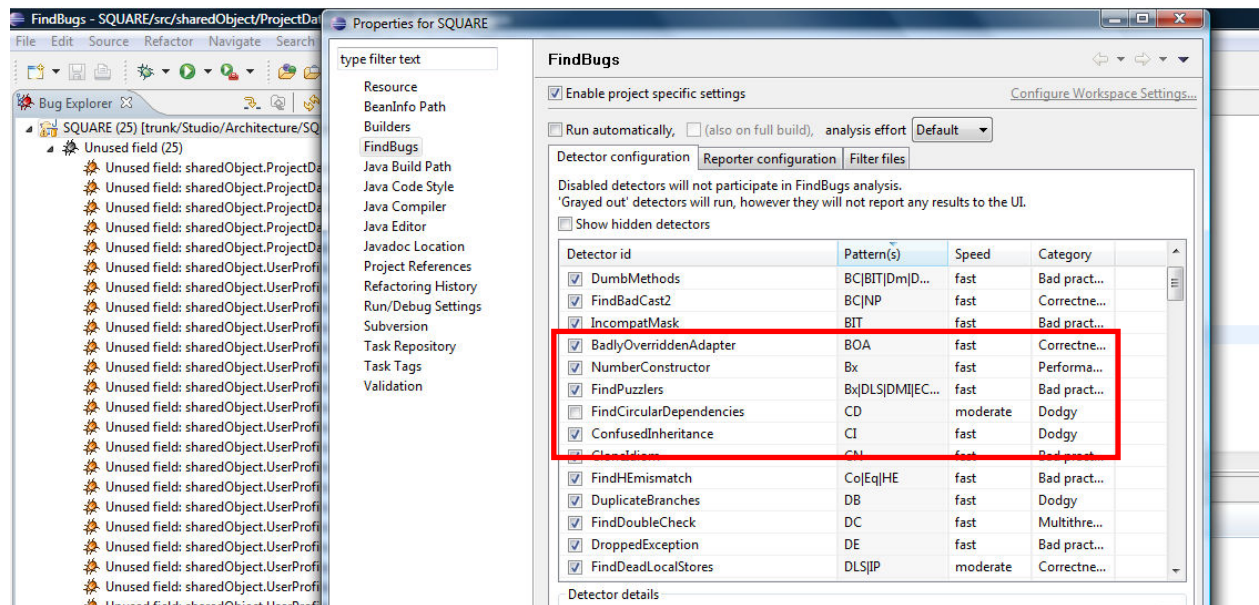
- Filter the results of the tool in order to analyze specific bug patterns
- Configure the tool so that it is restricted to finding only certain bugs that are of interest to the user.

To filter the results to view certain bug patterns:

- Select the **FindBugs** perspective.
- Select the bug pattern that isn't of relevance
- Right click on it to show the "Toggle Filter" menu.
- Select one of two toggle actions:
  - "This Specific Bug Pattern" action will toggle on/off only one, specific bug pattern
  - "Bug Pattern Type" action will toggle the whole group of patterns on/off



To modify the configuration of the tool, view the project's properties, and select **FindBugs**:



FindBugs preference page has three tabs which control different aspects of FindBugs plug-in behavior. The point of interest for us has been customization of the filters in the first tab.

Given the large number of projects at our disposal, we were faced with a quandary about the selection of the projects that would yield maximum exposure of the tool's capabilities. After much deliberation, we chose three projects (our Studio project, the Connect Four game, and an Architecture project which required a database connection) that we hoped would yield a diverse array of results.

## Customization

- We used the FindBugs eclipse plug-in instead of the FindBugs standalone tool. We did this because we wanted to find out if we could use this in an ongoing basis as part of our studio development environment. We also wanted easy traceability into the source code.
- We analyzed the project source files instead of the class files using FindBugs. This was done to easily trace to the buggy source code.
- We opted to use one of our own projects (from Architecture) instead of an open source project. Since we knew the project objectives and source code already, we could easily understand what kinds of errors FindBugs would detect for us. If we had chosen an open-source project, the intent of the source code would not be immediately clear to us.
- We saw the following FindBugs categories in FindBugs:

| Category                     | Description  | Example   |
|------------------------------|--|---|
| Bad Practice                 | Practices that violate recommended coding practices. For example using operator “==” equals instead of object equals, serializability problems, and missing finally clause for closing connections | Using “==” to compare string objects.                                       |
| Dodgy                        | Code that is confusing, anomalous, and error-prone. For example null-dereference, and catch-all exceptions.  | Dereferencing null without a prior null-check.                              |
| Performance                  | Inefficient memory usage/buffer allocation, usage of non-static classes.   | Creating a new String(String) constructor.                                  |
| Internationalization         | Use of non-localized methods   | Use of non-localized String.toUpperCase                                     |
| Malicious code vulnerability | Variables or fields exposed to classes that should not be using them.  | Returning a reference to mutable object may expose internal representation. |
| Bogus random noise           | Bug data mining related. Not useful in bug-finding.  | --  |
| Correctness                  | Apparent coding mistakes.  | --  |
| Multithreaded correctness    | Thread synchronization issues.   | --  |
| Security                     | Similar to malicious code vulnerability.   | Passing a dynamically created string to a SQL statement.                    |



- We excluded “Malicious code vulnerability” as a bug category. The bugs we found from this category were not useful to us. Security was important to this architecture project and to our studio project, but most high-priority security bugs were caught in the “security” category.
- We also excluded “Bogus random noise”. It was strictly data-mining related.
- Internationalization was not relevant to A2.
- We decided to exclude the “low-priority” bugs. They did not provide us helpful information about bugs (more false-positives). Example low-priority bugs were:
  - 1) Field names should start with lower-case
  - 2) Method may fail to close stream on exception (redundant).

## Applicability

### Finding Bugs

FindBugs can be applied very easily to java projects. It integrates well with Eclipse, and therefore can be applied on any java project being developed in Eclipse. Its high usability makes it appropriate for developers to check for common errors.

### Evaluating libraries

It can be used to evaluate library or framework code correctness. Nowadays, a lot of open source projects are in use. Running find bugs on them can provide a quick evaluation of how buggy the particular library or framework is. We found 8000 bugs after running it on the Google Web Toolkit, which was surprising, considering Google’s legendary code quality measures.

### FindBugs vs. Inspection vs. Dynamic Testing

We ran FindBugs against our analysis assignment 2 code to compare it with inspection and dynamic testing. In analysis assignment 2, we conducted an inspection and 3 kinds of dynamic tests: black box, coverage, and random.

Since both inspection and FindBugs check the static code, we expected the tool to find the same bugs as inspection. Unfortunately, it only found 9 bugs on the Assignment 2 code. None of these overlapped with the 26 bugs we found in inspection.

- In code inspection, we found a lot of bugs related to the method contracts. FindBugs was not able to find these because it does not check method contracts.
- Code inspection also located errors with the game rules implementation. Those were not found with FindBugs.
- FindBugs on the other hand, found bad practices such as invoking `System.exit()`, and null-dereference related issues. These were not found in inspection.
- Coverage testing found some of the null pointer related issues that FindBugs located. However, coverage tests were also based on game rules and domain knowledge that FindBugs could not use. Therefore FindBugs could not find those errors.

- Black box test is wholly dependent upon interface contracts, and therefore the 13 bugs found in Black box were completely different from what FindBugs located. The same is true for Random tests.

These were the number of bugs found and time spent on each type of tests:

| Strategy        | Person Hours Spent | Bug Count | Yield (Defects/hour) |
|-----------------|--------------------|-----------|----------------------|
| Black Box Test  | 20                 | 13        | 0.65                 |
| Code Inspection | 19                 | 26        | 1.37                 |
| Coverage Test   | 8                  | 6         | 0.75                 |
| Random Test     | 22                 | 12        | 0.55                 |
| FindBugs        | ~                  | 9         | 9                    |

From the above analysis, we conclude:

- FindBugs is the cheapest method of testing in terms of raw bug numbers.
- It cannot find a lot of bugs that the other kinds of tests find. Therefore it cannot substitute any of these.
- It finds bugs that other kinds of tests may not find. Therefore it can complement the other testing activities.
- The bugs found by FindBugs are tedious to detect manually, but easy to detect mechanically. FindBugs can be used before starting other kinds of tests or along with other kinds of tests (perhaps by adding it to a continuous integration tool).

## Analyzed Projects

We ran FindBugs on four projects. The overall strategy was to use different types of projects to try to cover a broader scope regarding defect categories, so that we can see the accuracy of the tool.

- **Project A - Connect 4 game UI:** The size is 1959 Lines of Code. This is the implementation is a web based application based g the Google Web Toolkit.
- **Project B - House and Health Insurance System (Architecture A2):** The size is 2609 Lines of Code. We created this project in the Architecture course. This is a three layer system that uses JDBC and RMI classes. The rational was to assess the tool over database constructs and RMI constructs.
- **Project C- Hnefatafl (Version with 5 seeded bugs):** The size is 1192 Lines of Code. The rational for selecting this project was to compare the results of the inspections and testing techniques on project at the beginning of the semester against FindBugs results.
- **Project D – GWT Libraries** – The size of this project was approximately 16,000 classes. The rational is to proof that the tool works only with the byte codes, the other

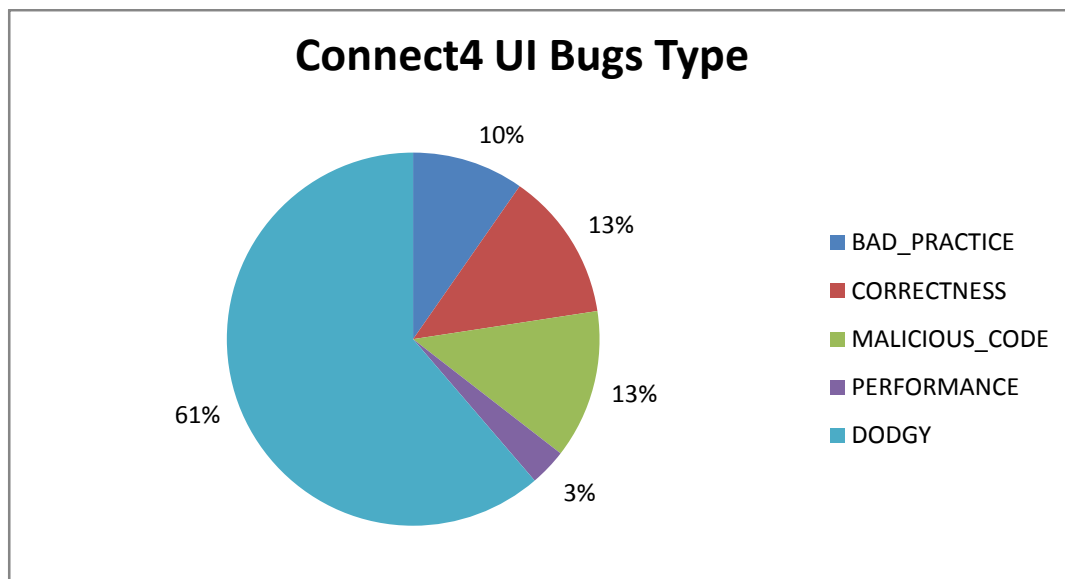


thing we wanted to verify is whether the GWT libraries are reliable according to FindBugs. Since we are using them as a core asset for our studio. Note: Due to the large amount of classes of this project approximately 16,000 we are not going to analyze the false positives and true positives. However, we would like to see an overview of the type of bugs found in the library.

| Project | Decryption     | Bugs | Size LCO |
|---------|----------------|------|----------|
| A       | Connect 4 Game | 32   | 1959     |
| B       | House & Health | 64   | 2609     |
| C       | Hnefatafl      | 5    | 1192     |
| D       | GWT Library    | 4304 | 263,346  |

### Project A – Connect 4 Game UI

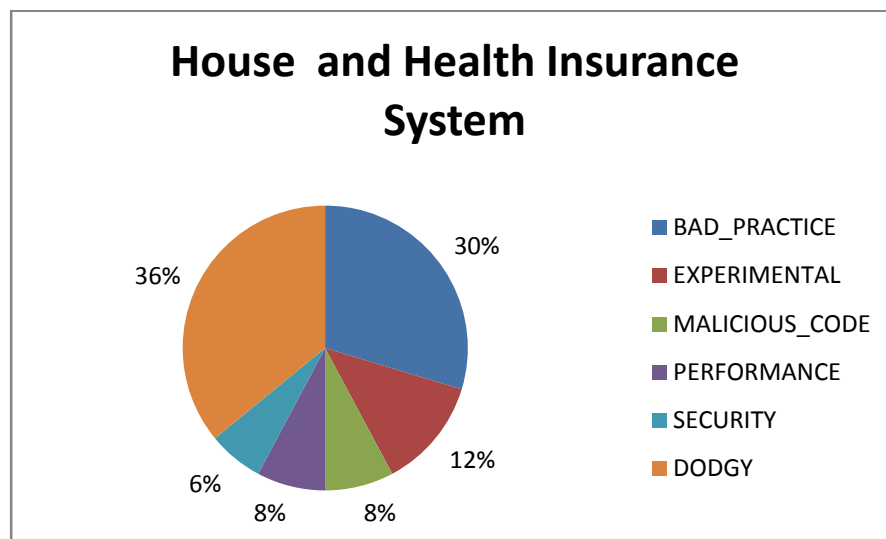
| Bug Category   | Amount    |
|----------------|-----------|
| BAD_PRACTICE   | 3         |
| CORRECTNESS    | 2         |
| MALICIOUS_CODE | 4         |
| PERFORMANCE    | 1         |
| STYLE          | 22        |
| <b>Total</b>   | <b>32</b> |



As can be seen in the chart the majority of the bugs found in the Connect 4 UI project were in the category of DODGY, which is related to coding styles issues according to the FindBugs categories. This might be related to the fact that the majority of the team members are new to Java and lack of coding style and good practice.

## Project B – House and Health Insurance System - (Architecture Project A2)

| Bug Category   | Amount    |
|----------------|-----------|
| BAD_PRACTICE   | 19        |
| EXPERIMENTAL   | 8         |
| MALICIOUS_CODE | 5         |
| PERFORMANCE    | 5         |
| SECURITY       | 4         |
| DODGY          | 23        |
| <b>Total</b>   | <b>64</b> |



In the chart shown above we can see that still one of the main problems with our team code are the DODGY and BAD\_PRACTICE types. Another important thing to highlight is that the SECURITY bug category that was not presents in Project A appeared here. This showed us how important is to try different types of projects in order to properly evaluate a tool.

This is a snippet of the code with security issues:

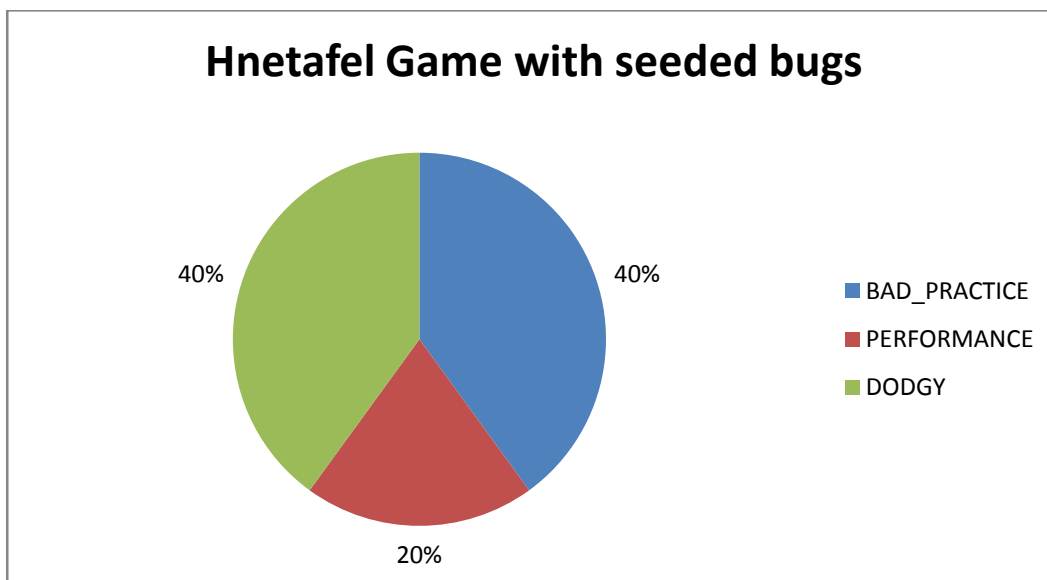
```
Statement sta = con.createStatement();  
    // execute query  
String query = "SELECT * FROM " + TABLE + " WHERE  
CUSTOMER_ID = '" + customerID + "'";  
  
ResultSet table = sta.executeQuery( query );
```

**Pattern id:** SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE,  
**Type:** SQL,  
**Category:** SECURITY

The method invokes the execute method on an SQL statement with a String that seems to be dynamically generated. Consider using a prepared statement instead. It is more efficient and less vulnerable to SQL injection attacks.

### Project C – Hnetafel Game with seeded bugs

| Bug Category | Amount |
|--------------|--------|
| BAD_PRACTICE | 2      |
| PERFORMANCE  | 1      |
| DODGY        | 2      |
| Total        | 5      |



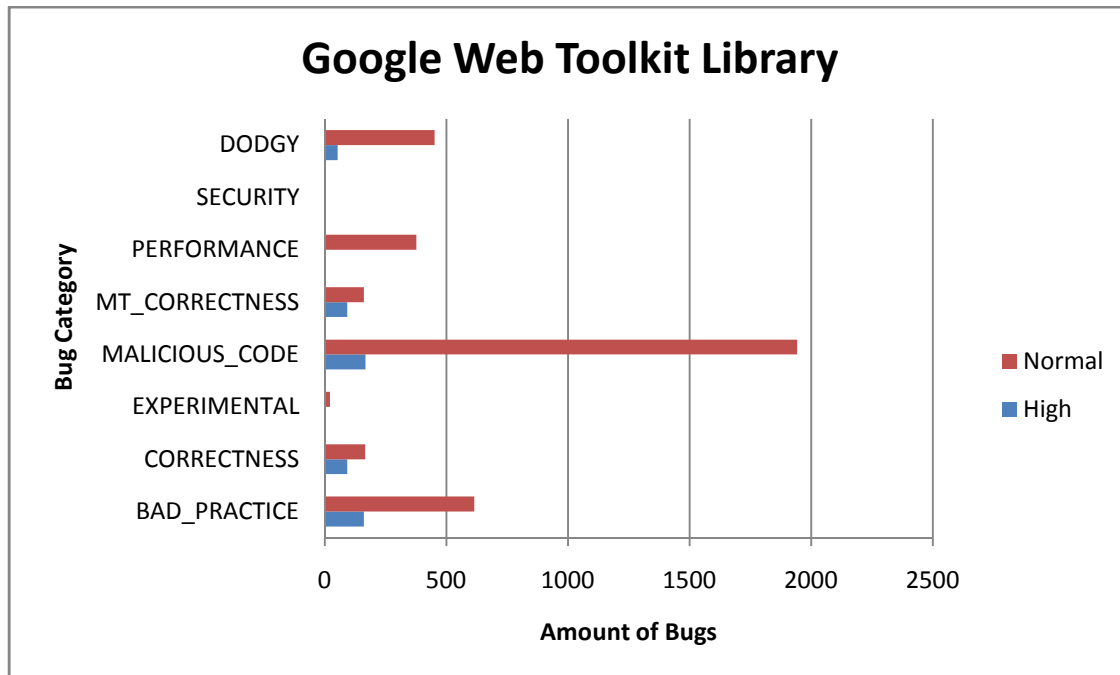
One of the most interesting things we found out here is that none of the seeded bugs were found by the tool which is an indicator that static tools might need to be complemented with other quality assurance techniques. Also, we can see that the proportion of our team bug categories holds across Project A, B and C.

### Project D – Google Web Toolkit libraries

| Bug Category   | High Priority | Normal Priority | Total |
|----------------|---------------|-----------------|-------|
| BAD_PRACTICE   | 160           | 615             | 775   |
| CORRECTNESS    | 92            | 166             | 258   |
| EXPERIMENTAL   |               | 21              | 21    |
| MALICIOUS_CODE | 167           | 1942            | 2109  |
| MT_CORRECTNESS | 92            | 161             | 253   |
| PERFORMANCE    | 4             | 376             | 380   |
| SECURITY       |               | 4               | 4     |

|       |     |      |      |
|-------|-----|------|------|
| DODGY | 53  | 451  | 504  |
| Total | 568 | 3736 | 4304 |

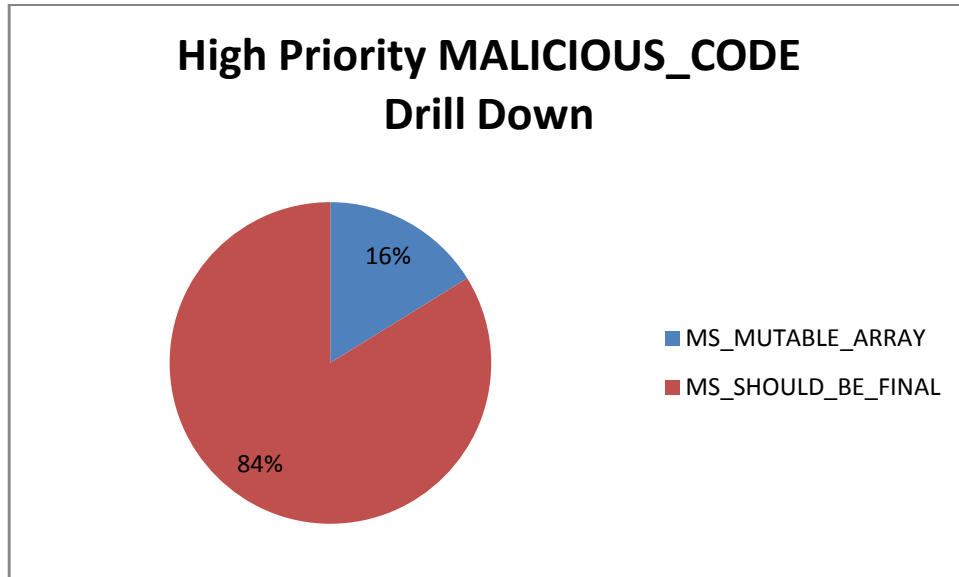
It is interesting how a well know company such as Google has so many bugs according to FindBugs in the famous GWT library.



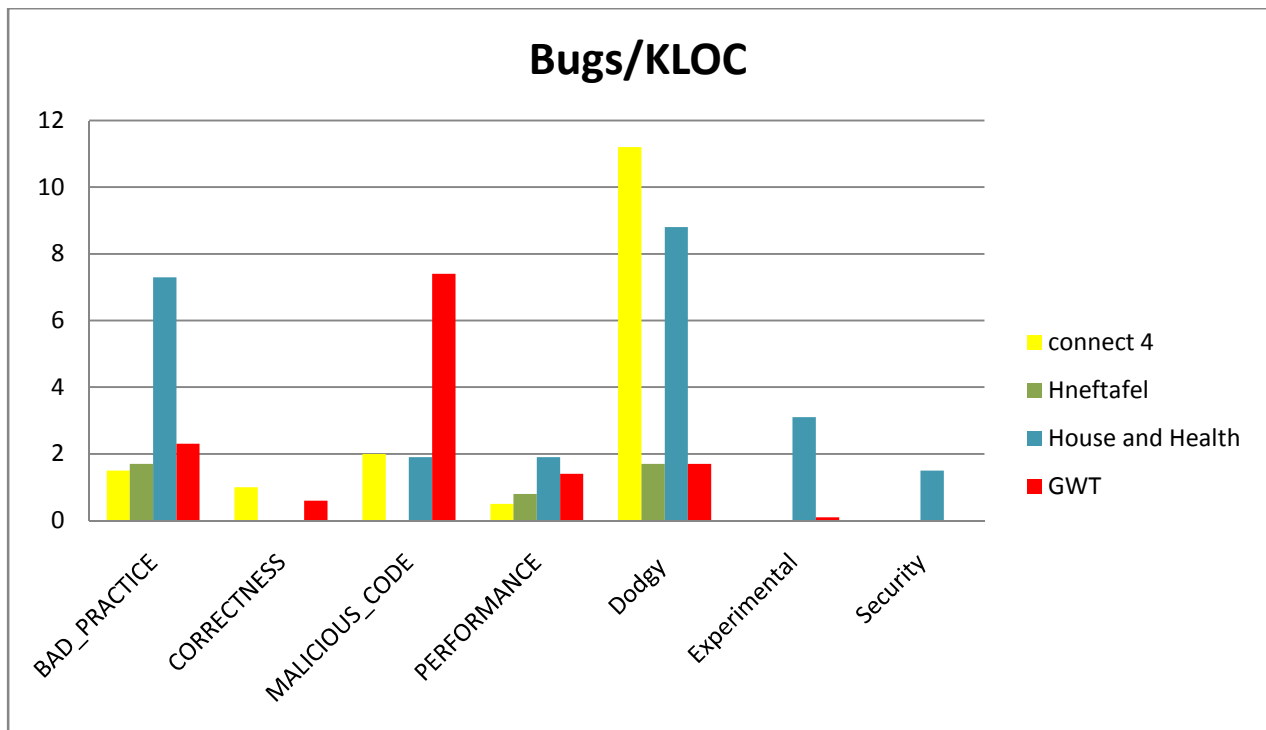
To have a better idea of this we will analyze the high priority bugs with most occurrences according to FindBugs, which in this case are MALICIOUS\_CODE bug category.

Google Web Toolkit – Malicious code category drill down.

| Bug Type           | Amount     |
|--------------------|------------|
| MS_MUTABLE_ARRAY   | 27         |
| MS_SHOULD_BE_FINAL | 140        |
| <b>Total</b>       | <b>167</b> |



We found the bug types in the Malicious\_Code category, were only from two types. MS\_MUTABLE\_ARRAY and MS\_SHOULD\_BE\_FINAL. The description for this type of bugs were similar to this: A mutable static field could be changed by malicious code or by accident from another package. The field could be made final to avoid this vulnerability. Due to the nature of the GWT libraries that could be extended and also modified by injecting JavaScript, this is something that might be consider not relevant for this project. Also, since this an open source code, these bugs are not relevant for Google in this context.



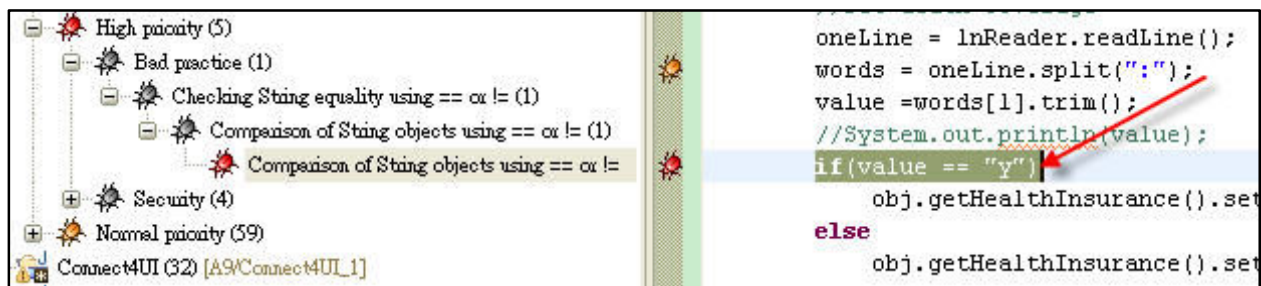
This chart show a picture of all projects with data normalized. We did this because the amount of lines of code varies a lot per project. We can conclude that Dodgy and Bad practices are the categories with most bugs across connect4, Hnefatafl and House and Health Insurance projects.

## True and False Positive Analysis

| Project   | #Bugs | True Positive and Relevant | Ratio | True Positive and Not Relevant | Ratio2 | False Positive | Ratio3 |
|-----------|-------|----------------------------|-------|--------------------------------|--------|----------------|--------|
| A2        | 64    | 29                         | 45%   | 33                             | 52%    | 2              | 3%     |
| Hnefatafl | 5     | 3                          | 60%   | 0                              | 0%     | 2              | 40%    |
| Connect4  | 32    | 25                         | 78%   | 6                              | 19%    | 1              | 3%     |

### True Positives and Relevant

- Pattern Type: Comparison of String objects using == or !=  
 Priority: High  
 Category: Bad Practice  
 Location: Architecture\_A2/insurance.businesslayer/InsuranceLogic.java



This code compares `java.lang.String` objects for reference equality using the `==` or `!=` operators. The FindBugs suggest that instead of using `==` or `!=`, using `equals(Object)` method is a better practice. The reason is that the same string value may be represented by two different String objects.

We consider this bug is true positive and relevant to the project. The reason is that if this string comparison is not executed correctly, the following code will not execute. This will cause the program generate unexpected result. Therefore, it is relevant to the correctness of the project.

- Pattern Type: Non constant string passed to execute method on an SQL statement  
 Priority: High  
 Category: Security

Location: Architecture\_A2/insurance.datalayer/QueryHIDB.java

```

// execute query
String query = "SELECT * FROM " + TABLE + " WHERE CUSTOMER_ID = '" + customerID + "'";
ResultSet table = sta.executeQuery( query );
ArrayList<String> policyList= new ArrayList<String>();

```

The method invokes the execute method on an SQL statement with a String that is dynamically generated. The FindBugs suggest using a prepared statement instead. By doing so, it will be more efficient and less vulnerable to SQL injection attacks.

We consider this bug is true positive and relevant to the project. The reason is that security is one of the requirements of this project, and this part of code will make the system vulnerable. Therefore, it is a bug fails to achieve the security properties of the project.

### True Positives and Not Relevant

- Pattern Type: Method name should start with a lower case letter  
Priority: Normal  
Category: Bad Practice  
Location: Architecture\_A2/insurance.datalayer/QueryHIDB.java

```

} // GePolicyByID
public ArrayList<String> GePolicyListByID( String customerID )
{
// set authentication properties for DB access
Properties aprops = new Properties();
aprops.setProperty( "user", USER );
aprops.setProperty( "password", PASSWORD );

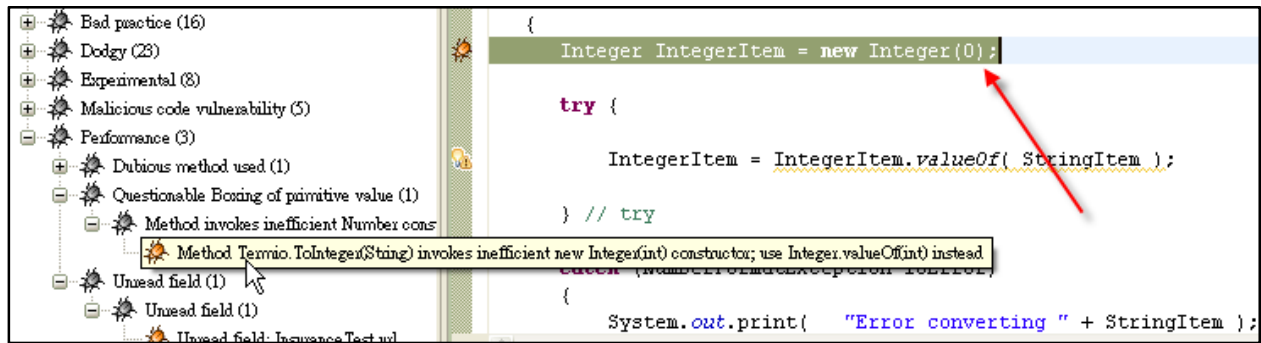
```

The FindBugs suggest that methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

We consider this bug is true positive, but not relevant to the project. This bug does affect the readability or maintainability of the project. However, it will not affect the correctness of program execution. Therefore, it is a bug that programmer does not follow a good coding standard.

- Pattern Type: Method invokes inefficient number constructor  
Priority: Normal  
Category: Performance  
Location: Architecture\_A2/insurance.datalayer/Termio.java





The code use Integer(int) to create instances of integer. Using new Integer(int) is guaranteed to always result in a new object whereas Integer.valueOf(int) allows caching of values to be done by the compiler, class library, or JVM. However, using of cached values avoids object allocation and the code will be faster. The FindBugs suggest that using valueOf is approximately 3.5 times faster than using constructor.

We consider this bug is true positive, but not relevant to the project. The bug does affect the performance of the project. However, it's a technique that can promote the performance of a program in general. Therefore, this part of code should be revised and this technique is applicable to all the projects.

## False Positives

1. The warning is the following:

**Project Bug:** Class com.mycompany.project.server.Connect4ServiceImpl defines non-transient non-serializable instance field adapter

**category:** BAD\_PRACTICE - This Objects of this class will not be deserialized correctly

This is a snippet of illustrates were the bug was identified:

```
public class Connect4ServiceImpl extends RemoteServiceServlet
    implements Connect4Service {

    private static final long serialVersionUID = 1L;
    Connect4ServiceAdapter adapter;

    public Connect4ServiceImpl() {
        adapter =new Connect4ServiceAdapter();
    }

    public MoveCommandResult move(int colum, int turn) {
        return adapter.move(colum, turn);
    }
}
```

The reason why we do not think this is real bug is because, the adapter object is not part of the service interface which is Connect4Service, this adapter object is meant to be a private fields. That is the Connect4ServiceAdapter should never be passed through the wire. Moreover, the adapter.move returns a serialiazable type, which is the important part. Additionally, the code

works just fine, we do not have problem with serialization. That is why we consider this bug as false positive.


2. The warning is the following:

**Bug:** RSPolicy\$HouseInsurance is serializable and an inner class

**Pattern id:** SE\_INNER\_CLASS, **type:** Se, **category:** BAD\_PRACTICE

This Serializable class is an inner class. Any attempt to serialize it will also serialize the associated outer instance. The outer instance is serializable, so this won't fail, but it might serialize a lot more data than intended. If possible, making the inner class a static inner class (also known as a nested class) should solve the problem.

This is a snippet shows the serialization warning:

```
public class RSPolicy implements Serializable{
     class HouseInsurance implements Serializable{
        private boolean hasValues;
        private String billingAddress = "";
    }
}
```


In this case the purpose of the HouseInsurance class is to be always serialized with the outer class which is the RSPolicy. This approach worked fine at runtime. However, when we removed implements Serializable in the inner class, a runtime serialization exception popped up at runtime indicating that the inner class should be serializable. That is why we believe this is a false positive, because it depends in the objective of the implementation.

3. The warning is the following

**Bug:** edu.cmu.isri.analysis654.hnefatafl.rules.MoveImplementation defines equals and uses Object.hashCode()

**Pattern id:** HE\_EQUALS\_USE\_HASHCODE, **type:** HE, **category:** BAD\_PRACTICE

This class overrides equals(Object), but does not override hashCode(), and inherits the implementation of hashCode() from java.lang.Object (which returns the identity hash code, an arbitrary value assigned to the object by the VM). Therefore, the class is very likely to violate the invariant that equal objects must have equal hash codes.

```
public boolean equals(Object obj) {
     if (!(obj instanceof MoveImplementation)) {
        return false;
    }
    MoveImplementation m = (MoveImplementation)obj;
    if (this.source.equals(m.getSource())
        && this.destination.equals(m.getDestination())) {
        return true;
    }
    return false;
}
```

In this case we think this is false positive because, the actual equals(Object obj) work as expected and we do not plan to store this type of objects in a hast table. We think it is a good practice not to violate the invariant that equal objects must have equal hash codes, however, this should be agreed as a team before considering it high priority bug.

## FindBugs Recommendations

**Don't bother with the “preferred” user interface; Eclipse plug-in is amazing.** The FindBugs documentation recommends using a Swing-based user interface over an Eclipse-based plug-in. In our testing we found the Swing UI to awkward in that it broke flow to build a FindBugs project and required context switching to repair found errors. The Eclipse plug-in, on the other hand, integrated smoothly with the development environment, creating warnings on builds. Additionally, the provided FindBugs view makes it easy to navigate the results of the analysis side-by-side with code and detailed descriptions of bug patterns.

**FindBugs catches a lot of bugs but not everything; inspections are still required.** During our testing there were a few instances where a bug pattern would be caught, and then a few lines later not caught. Here's one such example.

```
306  if(value == "y")
307      obj.getHealthInsurance().setDeathCoverage(true);
308  else
309      obj.getHealthInsurance().setDeathCoverage(false);
310
311  //set incapacity coverage
312  oneLine = InReader.readLine();
313  words = oneLine.split(":");
314  value =words[1].trim();
315  //System.out.println(value);
316  if(value == "y")
317      obj.getHealthInsurance().setIncapacityCoverage(true);
318  else
319      obj.getHealthInsurance().setIncapacityCoverage(false);
```

In this code snippet, the first if check on line 306 was caught by FindBugs as a “string comparison using ==” bug but the second, identical if check on line 316 was not. Inspections, or at the least peer reviews, are still needed to identify these sorts of errors before testing.

Further, FindBugs does not detect logical defects such as infinite loops or algorithm correctness.

**Bug priorities seem random; you still have to use your head.** “Dead store to” signifies when a variable has been assigned that isn't used. For some reason this bug pattern shows up under both the high and normal priority lists for different variables in the code base. While it is useful to have some bug patterns marked at varying priorities, the seemingly random assignments make it difficult to trust the results.

**Agree on quality definitions before using the tool.** Given the wide variety of bug patterns recognized it is important to reach a common consensus on which bug types the team will take seriously and which bug types it won't. Doing this ahead of time reduced effort analyzing bugs detected so the team can quickly discard uninteresting things and resolve important things. With this in mind, FindBugs makes it easy to categorize and prioritize found defects.

**FindBugs is great for finding potential performance, security, and style errors.** FindBugs provides suggestions of good coding practices. For example, one should name a method with camel casing and use “.equals ()” instead of “==”. These kinds of defects can affect the readability and maintainability of the software. Additionally, FindBugs captures performance and security defects from the static code, such as invoking constructors inefficiently or passing user input as a SQL executable method. This kind of detection could solve performance and security issues before running dynamic tests.

**It is difficult to decrypt FindBugs errors if you have a functional programming background.** For a function programmer, some defects reported by FindBugs are hard to determine whether they are defects or not. However, if a programmer has profound domain knowledge of the object oriented programming, the defect reports would be very helpful to investigate those defects.

**Use FindBugs to evaluate third-party libraries and components.** Since FindBugs works on the bytecode, access to source code is not required to run it on third-party libraries. Running FindBugs over a library can help evaluate the library to determine if it is worth using. For example, if there are too many security bugs found, the library may not be the right choice.