ANALYSIS OF SOFTWARE ARTIFACTS

FINAL REPORT

TEAM NEO

JonggunGim
Chankyu Park
Heewon Lee
Miyul Park
Jeongwook Bang

Project Report

1. Overview

1.1. Title of Project

Evaluation of “FindBugs 0.8.7” as an alternative of Code Review

1.2. Members

JonggunGim	jggim@andrew.cmu.edu	Team Representative
Chankyu Park	ckp@andrew.cmu.edu	Warnings Investigation
Heewon Lee	heel@andrew.cmu.edu	Tool Study
Miyul Park	mip@andrew.cmu.edu	Process Manager
Jeongwook Bang	jbang@andrew.cmu.edu	Warnings Investigation

2. Project Description

Code Inspection is considered to be one of the most effective Software Engineering Practices in improving the quality of software. Although in many cases, its huge cost of initial investment discourages developers of active adoption of the practice. But recent studies show that large portion of errors detected by code inspection could be detected by automated static code inspection [1]. Common belief about logical errors is that they are usually very subtle and that static analyses are not adequate to find such errors. But it is getting more obvious, with empirical data, that static error detectors can sieve a big portion of logical errors. One such example is shown below.

```
If (in == null)
Try {
      In.close();
}///////from Eclipse 3.0.0 M8
```

FindBugs is one such tool. The team initiated the project in the hope of adapting these tools to help the team in code review for the studio project of team neo.

Type of this project was Tool Evaluation of FindBugs version 0.8.7 that was released on April-14, 2005.

1.3. Objectives of Project

Three of team members work in the same MSE studio project developing a Rule Based Management System named BizRules. These members wanted to analyze java libraries to be used for developing BizRules to enter the implementation phases with confidence about its robustness.

Another objective for all team members was to analyze java code created by them to understand common error patterns and prevent them.

3. Tool Description

FindBugs is a static Bug Pattern Detector that looks for both specific bugs and style violation that, in many cases, indicates problematic code. A bug pattern is a code idiom that is likely to be an error. Occurrences of bug patterns are places where code does not follow usual correct practice in the use of a language feature or library API. [01]

FindBugs defines a number of bug patterns, and automatically detects the bugs matched to the patterns. In addition, FindBugs uses static techniques, which explore abstractions of all possible program behaviors, and thus reduces the overhead by code inspection.

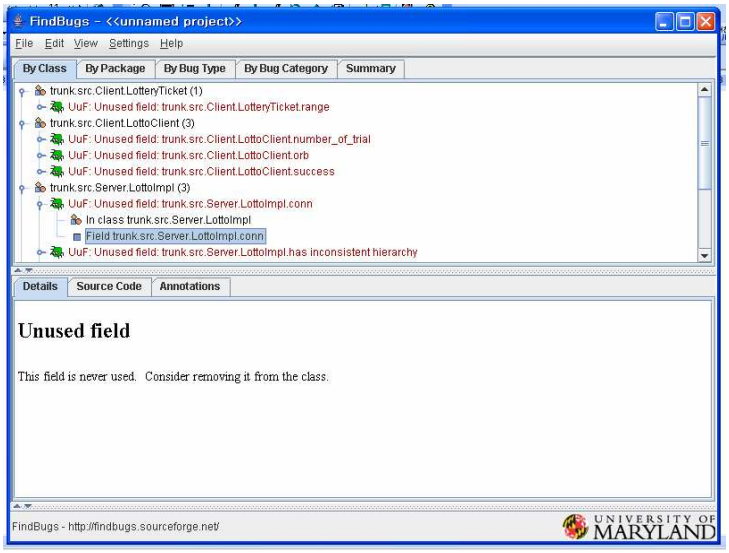

Figure 1. Overview of FindBugs

There are many tools similar to FindBugs. For example, ESC/Java we had considered as our analysis tool is also one of java bug finding tools. ESC/Java stands for the Extended Static Checking system for Java, and it basically performs formal verification of properties of Java source code. Besides, there are other similar tools: JLint, Eclipse (method naming), IntlliJ, IDEA, PMD, CheckStyle Jtest, FlawDetector, Wasp, and inForce.

FindBugs utilizes BCEL (Byte Code Engineering Library) to scan compiled java class files. Hence, source files are not prerequisites for analysis, but they can help the analysis to pinpoint the location of warnings in source codes.

1.4. How does this work?

A short Warning Code represents each bug pattern. Currently, FindBugs contains about 50 bug patterns. Besides, the bug detector part of FindBugs is implemented using the Visitor design pattern; each detector visits each class of the analyzed library or application. [1] One of the main techniques that FindBugs uses is to syntactically match source code to known suspicious programming practice. For example, FindBugs checks that calls to wait(), used in multi-threaded Java programs, are always

within a loop—which is the correct usage in most cases. In some cases, FindBugs also uses dataflow analysis to check for bugs. For example, FindBugs uses a simple, intra-procedural (within one method) dataflow analysis to check for null pointer dereferences. FindBugs can be expanded by writing custom bug detectors in Java. We set FindBugs to report “medium” priority warnings, which is the recommended setting.

1.5. Feature

Bug pattern detectors of FindBugs are divided into 4 categories; they are Single-threaded correctness issue, Thread/synchronized correctness issue, Performance issue, and Security and vulnerability to malicious un-trusted code. These detectors are implemented to catch various bug patterns. That is, Find bugs can find out the following types of bugs by using these detectors:

- OVERRIDE HASHCODE: Classes that override equals() must override hashCode.
- BOOLEAN CONSTANTS: Boolean is immutable; there are only two values.Constructing objects of this type is just a waste of memory.
- SERIALIZABLE: Serializable classes that contained non-Serializable, nontransient attributes. These classes could not be unserialized.
- REDUNDANT NULL COMPARISON: Comparison between two values that are both null, or when exactly one is null. This usually happens when a null check is made on a constructed object (since constructors can't return null), or when a null check is made on an object that is being previously referenced. Mostly these were simply unnecessary (and therefore inefficient), but in some cases they indicated real bugs.
- STRING CONSTRUCTION: String constructor is called unnecessarily. String s = new String (“some string”);
- UNUSED FIELDS: Unused (private) fields.
- STRING COMPARISON: When comparing String objects, the equals() method should be used (unless both strings are constants or have been interned).
- FINALIZE: Some classes that were overriding finalize() were not doing so correctly.
- NULL DEREFERENCE: Possible null pointer dereferences.
- THREAD START: Classes that start a thread in the constructor cannot be reasonably extended.
- UNREAD FIELD: Unused fields.
- STATIC CONSTANTS: Instance fields that are never read.
- MUTABLE STATICS: Non-final static fields. These are fields that can be changed by another package.
- BOOLEAN LOGIC ERROR: if ((tagged == null) && (tagged.length < rev))

Below are examples of Null Dereference warnings and Redundant Null Comparison warnings.

Null Dereference

```
// Eclipse 3.0.0 M8
Control c= getControl();
```

```
if (c == null && c.isDisposed())
return;

// Eclipse 3.0.0 M8
String sig = type.getSignature();
if (sig != null || sig.length() == 1) {
return sig;
}

// JDK 1.5 build 42
if (name != null || name.length > 0) {

if (flags != null) {
if (flags.length >= NUM_FLAGS)
this.flags = ...
else
this.flags = ...
} else
this.flags = ...
if (flags[RENEWABLE_TICKET_FLAG]) {
```

Redundant null comparison

```
protected Node findNode(Fqn fqn, ...) {
int treeNodeSize = fqn.size();
...
if (fqn == null) return null;
```

1.6. Usage

To perform analysis, we installed two kinds of FindBugs programs. One is Eclipse plug-in version, and the other is standalone program. Both are easy to install, but they are not identical. That is, each tool has different advantages, and hence we need to consider what tool is more suitable to us.

The characteristics of each tool are as follows:

	Advantages	Disadvantages
Eclipse plug-in	1. Each bug guides the user on where the relevant bug occurs. For example, if a program has some bugs, and they are detected, FindBugs will display the list of found bugs on the Problem view. On clicking one of the bugs, the user can reach the place the bug occurs. This source trace function of FindBugs Eclipse plug-in is really helpful to fix bugs.	1. FindBugs for Eclipse plug-in is not available in some environment. Even though the program is installed, it is not working. We think that the reason is maturity of the FindBugs. 2. FindBugs for Eclipse plug-in does not provide refined display. Because of this, collecting meaningful bugs can be tricky.

	2. Basic functions of Eclipse plug-in can be used to fix found bugs. Even though this is not the character of FindBugs, as it combining with Eclipse, our objective, which is reducing the inherent bugs, can be achieved more easily.	
Standalone	1. This version can display the categorized results. The supported categories are classes, packages, bug types, and bug category. 2. It supports to export results to XML. This tool has the feature to summarize the analysis result. In addition, this results can be saved as a XML file, and hence we can get more readable results.	1. Standalone program provides source-trace only as using a notepad. In addition, not all bugs can be traced. Therefore, fixing a bug is annoying, because we may add new bugs during editing. 2. Poor GUI can be also one of disadvantages. Poor GUI means poor usability. It has only basic file editing functions such as edit, copy, and paste. However, recently, many analysis tools provide usable design. Therefore, GUI will be a limitation of this tool.

Based on our experience, since we needed to correct discovered bugs and to sort out meaningful errors, FindBugs for Eclipse plug-in was a better choice. The refined source codes would be used for Studio project in the future. In light of this, as using editor functions of Eclipse, we could easily correct the errors which FindBugs found, and hence the codes became more mature.

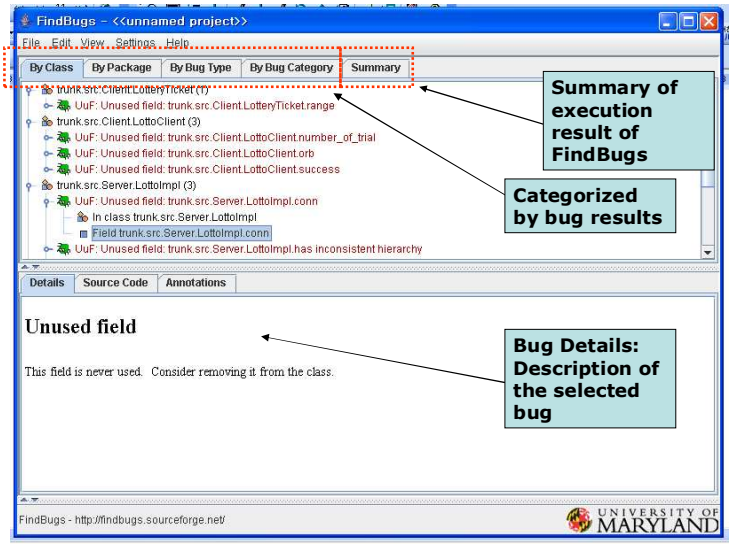

Figure 2. Standalone FindBugs - Categorized results

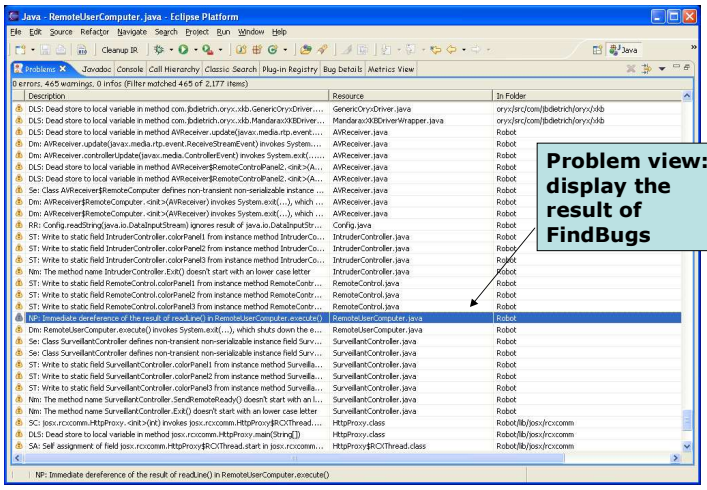

Figure 3. FindBugs for Eclipse plug-in - problem view to display bugs

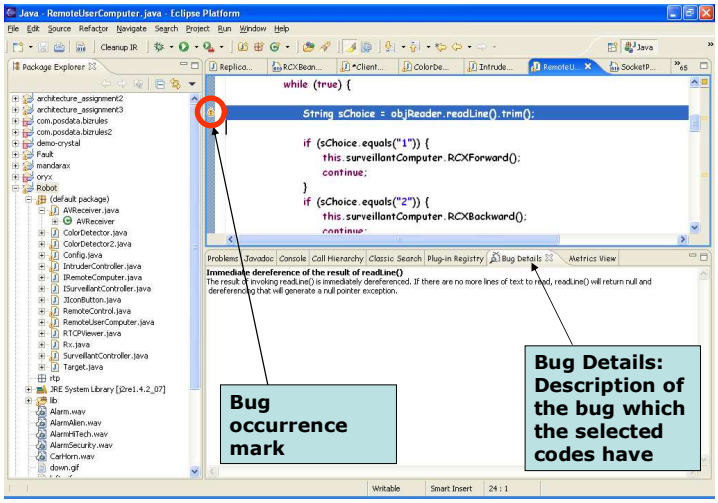

Figure 4. FindBugs for Eclipse Plug-in - Source Traceability

1.7. Strength

- It is easy to install and to run: As stated above, the installation of FindBugs is very simple. We just downloaded a zip file and extracted it. Then, it was done. We just run the bat file.
- FindBugs is extensible: We can add our own bug pattern codes into existing FindBugs. This implies that there is no limitation in term of types of bugs. If we can handle to implement the bug patterns, FindBugs will be a more powerful analysis tool.

1.8. Weakness

- Provided source code metrics are too simple: FindBugs only provides simple data about source codes, such as the number of classes or the number of packages. However, the number of field variables or static variables can be a criterion, because FindBugs catches the case that a static variable is assigned directly. To handle this problem, we have to use another metric tool to analyze the results of FindBugs.

- FindBugs does not have help files: even though it is easy to run, the absence of manual or help files can lead to convenience.

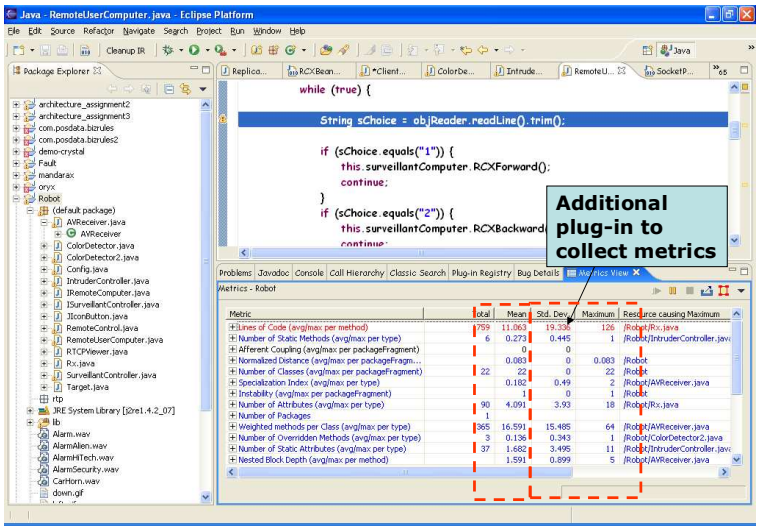

Figure 5. Additional metrics plug-in

4. Analysis Description

1.9. Objectives of Analysis (Expected Results)

Analysis using FindBugs is a two-phase activity; collecting warnings by running FindBugs with java class files / jar files (team decided to call this “Analysis”) and Investigating the errors reported by FindBugs (team decided to call this “Investigation”).

Analyses by FindBugs take trivial amount of time even with programs of very large size. Hence, the team agreed that analyzing the characteristics of the analysis progress itself is of very little value to the team (e.g., measuring analysis time for different size of application, measuring time for analysis for finding different kind of errors).

Rather, investigating the warnings reported by the tool and measuring/analyzing properties associated with this activity were focus of experiments believing this will promote the objective of this project (refer to “objectives of project”).

Main interests of analyses are;

- How much of reported warnings are real errors of the program?
- Do different programs have different error patterns?
- What are common mistakes we make in our code?
- How much time do we need to determine genuineness of each errors? And which types of errors require more times for investigations?

1.10. Scope of Analysis

Mandarax library (332 classes, 10,259 LOC) being used in neo’s BizRules system was fully analyzed and investigated. This experiments also revealed FindBugs’ performance in analyzing bug programs.

Two smaller java based programs (22 classes/1,759 LOC and 39 classes/2,733 LOC, respectively) created by team members as assignments of “Fault Tolerant Middleware” and “Systems Engineering” were analyzed and investigated. This is mainly for uncovering error patterns of our code to help preventing similar mistakes in the future.

1.11. Data Collected, Calculated, and analyzed

Both summary data of each analysis and investigation results of individual errors were collected under following scheme.

4.1.1. Summary of Analysis

Name	Description	Unit	Data Type
Title	Unique Descriptive Title of Analysis		
Date	Date when the analysis was performed		
Target Program	Program Analyzed		
# Class	Number of Class	each	Measured
LOC	Line of Code	Loc	Measured
# Warnings	Warnings reported by FindBugs without any filtering parameter given	each	Measured
# Correctness	Number of warnings under category of correctness problem	each	Measured
# Malicious	Number of warnings under category of malicious code vulnerability	each	Measured
# Multithread	Number of warnings under category of multithread correctness	each	Measured
# Performance	Number of warnings under category of Performance	each	Measured
# Style	Number of warnings under category of Style	each	Measured
# Bug Types	Number of types of warnings reported	each	Measured
# Critical Error	Number of errors of following six types. They are usually considered to be REAL errors	each	Measured
# Eq	Number of warnings of Type Eq.	each	Measured
# HE	Number of warnings of Type HE	each	Measured
# MS	Number of warnings of Type MS	each	Measured
# Se	Number of warnings of Type Se	each	Measured
# DE	Number of warnings of Type DE	each	Measured
# CN	Number of warnings of Type CN	each	Measured
# Serious	Number of Errors determined to be real fault and needing correction after investigation	each	Measured
# Harmless	Number of Errors determined by investigation to be harmless to program and does not require modification.	each	Measured
# False Positive	Number of Errors determined by investigation to be ones due to deficiency of analysis algorithm and not requiring modification.	each	Measured

4.1.2. Investigation Results

Name	Description	Unit	Data Type
Id	Sequential number unique under each analysis and Error Type		

Error Type	Type of Error		Given
Assignee	The name of person assigned to investigate the warning		
Begin Investigation	Time that the assignee began investigating the warning	Time	Measured
Finish Investigation	Time that the assignee finished investigating the warning	Time	Measured
Elapsed Time	Time taken for investigation of each warning	Minutes	Calculated
Conclusion	Serious/Harmless/False Positive, see preceding table for definition of these conclusions		Determined
Remarks	Can indicate sub type of errors		

5. Analysis Result

Below are summaries of three experiment results.

1.12. Program I – Mandarax

Mandarax is java based RBMS system developed by Jens Dietrich of Massey University. Current version is 3.4 released on March 6, 2005.

Mandarax							
#Total Loc	#Static methods	#Classes	#Fields	#Packages	#Static Fields	#Methods	#Interfaces
10,259 lines	66	332	311	28	361	1922	50

Analysis Results From FindBugs					
Bug Category	#Correctness		17	11%	
	#Malicious		109	69%	
	#Multithread		2	1%	
	#Performance		14	9%	
	#Style		16	10%	
	Total # bugs		158	100%	
Bug Code	warnings	serious	mostly harmless	false positive	Investig. Time
CD	4	0%	100%	0%	0:02:00
DLS	7	100%	0%	0%	0:00:26
Dm	3	100%	0%	0%	0:01:00
EI	44	100%	0%	0%	0:00:33
EI2	31	100%	0%	0%	0:02:15
HE	3	100%	0%	0%	0:03:20
IS2	2	0%	100%	0%	0:01:00
MS	34	100%	0%	0%	0:00:58
RCN	6	100%	0%	0%	0:03:40
REC	3	100%	0%	0%	0:01:20
SIC	3	100%	0%	0%	0:02:00
ST	2	0%	100%	0%	0:05:30

Se	2	100%	0%	0%	0:05:00
SnVI	6	100%	0%	0%	0:03:30
UPM	3	100%	0%	0%	0:01:10
UrF	3	100%	0%	0%	0:01:40
WMI	2	0%	100%	0%	0:02:00

1.13. Program II – Logo Robot system for Systems Engineering Class

This java program is developed by one of team member as the project of Systems Engineering class. This program is developed to run on Jini environment and deployed to Logo Robot system to enable it to detect intruders.

Codes were written mainly by one person in half a semester.

ESIS Course Robot Project							
#Total Loc	#Static methods	#Classes	#Fields	#Packages	#Static Fields	#Methods	#Interfaces
1,759 lines	6	22	90	1	37	153	2

Analysis Results From FindBugs					
Bug Category	#Correctness		21	47%	
	#Malicious		0	0%	
	#Multithread		0	0%	
	#Performance		0	0%	
	#Style		24	53%	
	Total # bugs		45	100%	
Bug Code	warnings	serious	mostly harmless	false positive	Investig. Time
DLS	4	100%	0%	0%	0:03:00
Dm	6	0%	100%	0%	0:01:30
FI	1	100%	0%	0%	0:03:00
Nm	5	100%	0%	0%	0:01:36
NP	1	100%	0%	0%	0:02:00
RR	1	100%	0%	0%	0:01:00
RV	2	100%	0%	0%	0:02:00
SA	1	100%	0%	0%	0:02:00
SC	3	100%	0%	0%	0:00:40
Se	6	100%	0%	0%	0:01:10
ST	14	100%	0%	0%	0:01:30
UR	1	100%	0%	0%	0:02:00

1.14. Program III – Fault Tolerant Lotto System

This program was developed for class project of Fault Tolerant Middleware class. Two of team member participated in this project. This program is text menu driven CORBA application including both server and client.

The program was developed by five developers in one semester.

Fault-tolerant Distrubuted Systems Project							
#Total Loc	#Static methods	#Classes	#Fields	#Packages	#Static Fields	#Methods	#Interfaces
2,733 lines	61	39	72	5	72	196	4

Analysis Results From FindBugs					
Bug Category	#Correctness		13	37%	
	#Malicious		13	37%	
	#Multithread		0	0%	
	#Performance		4	11%	
	#Style		5	14%	
	Total # bugs		35	100%	
Bug Code	warnings	serious	mostly harmless	false positive	Investig. Time
CD	2	100%	0%	0%	0:04:00
DLS	2	100%	0%	0%	0:03:00
Dm	3	0%	100%	0%	0:03:00
EI	6	100%	0%	0%	0:02:40
EI2	7	100%	0%	0%	0:01:09
IP	2	100%	0%	0%	0:10:30
Nm	1	0%	100%	0%	0:11:00
ODR	5	100%	0%	0%	0:01:48
REC	1	100%	0%	0%	0:04:00
ST	2	100%	0%	0%	0:04:30
UrF	2	100%	0%	0%	0:04:00
UuF	2	100%	0%	0%	0:01:30

1.15. Observation on Warnings and Errors

Warnings per line of codes are 0.015, 0,025, and 0.012 for each experiment respectively. The E/W (Error/Warning) ratio for each experiments are 76%, 92% and 83% respectively. So we can say program II has highest error density.

Three programs show different patterns of warnings in terms of their categories; program I shows the malicious code vulnerability is most abundant error categories whereas in program II and III, correctness/style and correctness/malicious code vulnerability are most abundant warnings.

Many of malicious code vulnerability warnings for program I come from EI and MS. These broadly mean that many of internal reference values are exposed to calling functions and could be mutated to corrupt internal data.

ST warnings abundantly reported from program II is for writing to static field from instance methods. These usually do not directly mean error, but this style is reported to be a bad practice frequently causing troublesome behavior.

EI is an issue in program III also. And ODR is also extensively reported from it. ODR means unclosed database connection. And this is should be easily investigated and fixed.

1.16. Observation on Investigation Time

Average investigation times for each program are 1:31, 1:35, and 3:12 respectively. Program III was investigated by different members than the previous investigations who were also not the developers of the program. And it required longer time to investigate.

In each experiments, ST/Se, DLS, IP/Nm was found to require longest time to investigate.

ST indicates writing to static field from instance methods.

Se indicates Non-transient non-serializable instance field in serializable class

DLS indicates Dead store to local variable

IP indicates that A parameter is dead upon entry to a method but overwritten

Nm indicates Field/Method names should start with a lower case letter.

Among these five types, now, DLS' and Nm's genuineness can be easily investigated.

1.17. Example of Serious Errors

Followings are examples of errors detected in experiments that we should try to avoid.

NP
The result of invoking readLine() is immediately dereferenced. If there are no more lines of text to read, readLine() will return null and dereferencing that will generate a null pointer exception.

```
while (true) {

        String sChoice = objReader.readLine().trim();

        if (sChoice.equals("1")) {
            this.surveillantComputer.RCXForward();
           continue;
        }
```

RCN
This method contains a redundant comparison of a reference value to null. Two types of redundant comparison are reported:

```
public void add(ResultSet rs) throws AggregationException {
    Object value = null;

    try {
            value = rs.getResult(var);
    }
    catch (Exception x) {
            throw new AggregationException("Cannot fetch value from result set for
variable",x);
    }
    if (type==null)
            setType(value.getClass());

    if (!(type.isAssignableFrom(value.getClass())))
            throw new AggregationException("Cannot use this function with the value has
been computed");
```

```
        values.add(value);
    }
```

REC: java.lang.Exception is caught when Exception is not thrown
This method uses a try-catch block that catches Exception objects, but Exception is not thrown within the try block, and RuntimeException is not explicitly caught. It is a common bug pattern to say try { ... } catch (Exception e) { something } as a shorthand for catching a number of types of exception each of whose catch blocks is identical, but this construct also accidentally catches RuntimeException as well, masking potential bugs.

```
private static org.mandarax.kernel.meta.JPredicate getEqualsNotPredicate() {
        if(equalsNot == null) {
            try {
                Class   obj = Object.class;
                Class[] par = new Class[1];

                par[0]    = obj;
                equalsNot = new JPredicate (obj.getDeclaredMethod ("equals", par), "equalsNot",
true);
            } catch(Exception t) {
```

6. Conclusion

1.18. On Using FindBugs

Suppose, as a pessimistic assumption, the E/W ratio is 0.5, and investigation for a warning takes 5 minutes. Then FindBugs allows us to find errors in programs in average 10 minutes. In case of manual code inspection, recommended inspection rate is 150 lines per hour [02] and if we assume errors per LOC is 0.02, you will find about 3 errors in an hour. This simple calculation shows using FindBugs is two times more efficient than manual error finding with most modest assumptions. We concluded using FindBugs is definitely a good way to reduce bugs in our programs.

But one advantage of manual inspection is that human's inspection is not limited to predefined patterns. Humans can imaginatively define new bug patterns as he inspect codes and can find many more types of errors that we will never be able to patternize. So tools like this will helpful in finding simplest types of errors but doing manual inspection will be still a good practice if time and resources permit.

We will continue to use FindBugs in our studio project. And if we could find error patterns from our code that are not currently caught by FindBugs, we will consider extending FindBugs by developing Bug Pattern Detector plug-in.

The bugs we found in Mandarax library will be reported to the maintainer of the library.

Bug patterns frequently found in our codes are ODR, EI and ST. This information is valuable to understand our common mistakes in java programming.

1.19. On FindBugs 0.8.2

In general, the tool was stable in spite of its low version number and very recent release date.

Performance was not a problem at all in all our experiments. User interface was intuitive enough to let us run the analysis not even reading the manual. Only file or directory selection dialog had us to consult the manual.

We observed one situation where Eclipse plug-in edition and Java application edition giving different result of analysis. This has to be further confirmed with more tests.

We found warnings can either can or cannot be related to specific line of code. For example, Circular Dependency warning cannot be related to one specific line. But we also observed some errors that can be related to a line do not show the file name and line number.

Documentation was satisfactory in general, and explanation for warning types were succinct and to the point in most cases.

Not all command line options could be configured in configuration dialog of GUI application. We hope this can be improved in next versions.

7. Glossary

- Warning: Potential Bugs reported by FindBugs. Requires further manual investigation to determine whether it is a bug.
- Error: Warnings determined to be real faults in the program that can be fixed by changing the code.
- Bug: same with Error
- Analysis: the activity of executing FindBugs to find warnings of a certain program.
- Investigation: manual effort to determine whether the warnings generated by FindBugs analysis is a real error.

8. Reference

[01] Finding Bugs Is Easy, William Pugh, David Hovemeyer

[02] Calculating the Economics of Inspections by Weller
http://www.stickyminds.com/sitewide.asp?ObjectId=3161&Function=DETAILBROWSE&ObjectType=ART

[03] A Comparison of Bug Finding Tools for Java, Nick Rutar and Christian B. Almazan