



Review of  
**Finding Bugs is Easy**  
by David Hovemeyer and William Pugh



presented by:  
Sandeep Sapra

# The BIG picture problem



- Removing bugs from programs

## Specific problem

- Automatic bug detection (in Java programs)

## Why is it hard?

- Undecidable problem
- Use of formal methods – difficult to apply to most programming languages (like C, C++)
- Code inspections – lot of effort, human error
- Testing – achieving high test coverage is difficult



# Do we care?

- “Software errors cost the U.S. economy about \$59.5 billion annually.” (from [Software Bugs Can be Lethal](#) )
- “Incidents of bad software have lead to plane crashes, road and rail accidents, malfunctions of medical gear!” (from [Software Hell](#) )
- “Production quality” code is undoubtedly well-tested! (still found to contain simple yet serious bugs.)
- There’s ongoing research investigating automatic techniques to find bugs in software
  - Lint, MC/Metal, PREFIX, Cqual



# Approach

- **FindBugs** tool - Automatic bug detection
- Implemented in Apache's BCEL
- Analyses Java byte code for about 50 *bug patterns*
- *Open source* –  
*<http://findbugs.sourceforge.net>*

# Bug Pattern detectors

- Bug Pattern detectors implemented using the Visitor design pattern [Gamma et. al.].
- Perform
  - Analysis of class hierarchy, method signatures
  - Linear byte code scan
  - Data flow analysis

# Bug Pattern

## Null pointer dereference (NP) detector

- Intraprocedural
- Does not handle null arguments to functions or null return values.

- Ex.1 pattern

```
if (object == null){  
    object.callSomething(); //B null deref  
}
```

- Ex. 2

```
int return = obj.procCall();  
if (obj != null) { ... }
```

# Bug Pattern

## Non-Short-Circuit Boolean Op (NS) detector

- Checks for confused non-short-circuit operators ‘&’ and ‘|’ with the short-circuit equivalents
- Ex. Eclipse 3.0  
if (cheatSheet != null & cheatSheet.getTitle() != null)  
return cheatSheet.getTitle();

# Bug Pattern

## Open Stream (OS) Detector

- Problem
  - Java finalizers are not guaranteed to run when program exits
  - Limited OS File descriptors
  - Data might never get written
- Detector
  - Performs a reaching definition (DEF = `openStream()`) on streams
  - Keeps track of the open state

# Bug Pattern

## Open Stream (OS) Detector

```
FileInputStream myFile = null;
try {
    myFile = new FileInputStream("abc.dat"); // open the stream
    boolean eof = false;
    while (!eof) {
        int byteValue = myFile.read(); // read the stream
        System.out.print(byteValue + " ");
    }
} catch (IOException e) { //do something
} finally{
    try{
        //myFile.close(); // close the stream
    } catch (Exception e1){
        e1.printStackTrace();
    }
}
```

# Evaluation

- How did they evaluate their tool?
- What was their criteria for evaluating the effectiveness of their tool?
- Did they have general results, or were the results varying for disparate applications?
- Do they have any numbers?
  - machine configuration?
  - memory?
  - time taken?

# Evaluation

- Ran the tool for six applications and libraries (listed in the table).
- 50% of reported warning were *serious* bugs.
- Detectors worked better on some applications.
- Performance- 1.8Ghz P4 Xeon, 1 GB mem, took max 65mins
- Ex. rt.jar – 13083 classes, 40MB size, 45 mins
- Max. memory usage 500MB

Application	Warnings	Serious
classpath-0.08	93	66%
rt.jar 1.5 build 59	349	68%
eclipse-3.0.0	420	65%
drjava-stable-20040326	13	77%
jboss-4.0.0RC1	118	47%
jedit-4.2pre15	22	50%

**Table 1**

# Conclusion

- Simple idiom detection techniques found real bugs in production software.
- Complement traditional testing.

# Future Work

- Integration of new bug patterns into the 'FindBugs' tool.
- Integration with IDE's
- Allowing developers to specify patterns
- False warning suppression based on automatic collection or heuristics

# Evaluation of the paper

- Quite well-written, very helpful examples
- Nicely presented description of ongoing and related work
- Elaborate analysis results and well summarized

# Evaluation of the paper

- Knowledge of Java assumed, also design patterns.
- How did they come up with the bug patterns? (survey? intuition?..)
- The tool's design for better understanding.
- Criteria that decides which warning are real bugs not clear (they say the tool is easy to apply, but which bugs are serious?)

Message: **Say “NO” to BUGS!!**

J

Questions?

# Appendix- Bug Pattern Codes

## Code Description

- CN Cloneable Not Implemented Correctly
- DC Double Checked Locking
- DE Dropped Exception
- EC Suspicious Equals Comparison
- Eq Bad Covariant Definition of Equals
- HE Equal Objects Must Have Equal Hashcodes
- IS2 Inconsistent Synchronization
- MS Static Field Modifiable By Untrusted Code
- NP Null Pointer Dereference
- NS Non-Short-Circuit Boolean Operator
- OS Open Stream
- RCN Redundant Comparison to Null
- RR Read Return Should Be Checked
- RV Return Value Should Be Checked
- Se Non-serializable Serializable Class
- UR Uninitialized Read In Constructor
- UW Unconditional Wait
- Wa Wait Not In Loop