FindBugs

William Pugh
Univ. of Maryland

http://www.cs.umd.edu/~pugh/
http://findbugs.sourceforge.net/

ITA Software &
MIT, September
2006
FindBugs

• Open source static analysis tool for finding defects in Java programs

• Analyzes classfiles

• Generates XML or text output
  • can run in Netbeans/Swing/Eclipse/Ant/SCA

• Total downloads from SourceForge: 274,291+
What is FindBugs?

- Static analysis tool to find defects in Java code
- not a style checker
- Can find hundreds of defects in each of large apps such as Bea WebLogic, IBM Websphere, Sun's JDK
- real defects, stuff that should be fixed
- hundreds is conservative, probably thousands
- Doesn’t focus on security
- lower tolerance for false positives
Common Wisdom about Bugs

- Programmers are smart
- Smart people don’t make dumb mistakes
- We have good techniques (e.g., unit testing, pair programming, code inspections) for finding bugs early
- So, bugs remaining in production code must be subtle, and require sophisticated techniques to find
Would You Write Code Like This?

```java
if (in == null)
    try {
        in.close();
    } ...
```

- Oops
- This code is from Eclipse (versions 3.0 - 3.2)
- You may be surprised what is lurking in your code
Why Do Bugs Occur?

- Nobody is perfect
- Common types of errors:
  - Misunderstood language features, API methods
  - Typos (using wrong boolean operator, forgetting parentheses or brackets, etc.)
  - Misunderstood class or method invariants
- Everyone makes syntax errors, but the compiler catches them
- What about bugs one step removed from a syntax error?
Bug Patterns
Infinite recursive loop

- Student came to office hours, was having trouble with his constructor:

```java
/** Construct a WebSpider */

public WebSpider() {
    WebSpider w = new WebSpider();
}
```

- A second student had the same bug

- Wrote a detector, found 3 other students with same bug
Double check against JDK

- Found 4 infinite recursive loops
- Including one written by Joshua Bloch
  ```java
  public String foundType() {
    return this.foundType();
  }
  ```
- Smart people make dumb mistakes
- Embrace and fix your dumb mistakes
Infinite Recursive Loops: Sun JDK history

I inform Sun of infinite recursive loops in their code.
Duration of infinite recursive loop bugs in JDK
Hashcode/Equals

• Equal objects must have equal hash codes

• Programmers sometimes override equals() but not hashCode()
  • Or, override hashCode() but not equals()

• Objects violating the contract won’t work in hash tables, maps, sets

• Examples (53 bugs in 1.6.0-b29)
  • javax.management.Attribute
  • java.awt.geom.Area
Fixing hashCode

• What if you want to define equals, but don't think your objects will ever get put into a HashTable?

• Suggestion:

```java
public int hashCode() {
    assert false : "hashCode method not designed";
    return 42;
}
```
Null Pointer Dereference

• Dereferencing a null value results in NullPointerException

• Warn if there is a statement or branch that if executed, guarantees a NPE

• Example:

  // Eclipse 3.0.0M8

  Control c = getControl();

  if (c == null && c.isDisposed())
     return;
Bad Binary operations

```java
if ((f.getStyle() & Font.BOLD) == 1) {
    sbuf.append("<b>");
    isBold = true;
}

if ((f.getStyle() & Font.ITALIC) == 1) {
    sbuf.append("<i>");
    isItalic = true;
}
```
public static final ASDDVersion getASDDVersion(BigDecimal version) {
    if(SUN_APPSERVER_7_0.toString().equals(version))
        return SUN_APPSERVER_7_0;
    return null;
}
Unintended regular expression

```java
String[] valueSegments = value.split("."); // NOI18N
```
public TagHelpItem(String name, String file, 
            String startText, int startOffset, 
            String endText, int endOffset, 
            String textBefore, String textAfter){
    this.name = name;
    this.file = file;
    this.startText = startText;
    this.startTextOffset = startTextOffset;
    this.endText = endText;
    this.endTextOffset = endTextOffset;
    this.textBefore = textBefore;
    this.textAfter = textAfter;
    this.identical = null;
}
Bad Naming

```java
package org.eclipse.jface.dialogs;
public abstract class Dialog extends Window {
    protected Button getOKButton() {
        return getButton(IDialogConstants.OK_ID);
    }
}
public class InputDialog extends Dialog {
    protected Button getOkButton() {
        return okButton;
    }
}
```
Confusing/bad naming

• Methods with identical names and signatures
  – but different capitalization of names
  – could mean you don’t override method in superclass
  – confusing in general

• Method name same as class name
  – gets confused with constructor
Bad naming in BCEL
( shipped in jdk1.6.0-b29)

/** @return a hash code value
 * for the object. */

public int hashcode() {
    return basic_type.hashCode()
    ^ dimensions; }

Ignored return values

• Lots of methods for which return value always should be checked
  – E.g., operations on immutable objects

// Eclipse 3.0.0M8
String name = workingCopy.getName();
name.replace('/', '.');
Ignored Exception Creation

/**
 * javax.management.ObjectInstance
 * reference impl., version 1.2.1
 **/

public ObjectInstance(ObjectName objectName,
                       String className) {
    if (objectName.isPattern()) {
        new RuntimeOperationsException(
            new IllegalArgumentException(
                "Invalid name->"+ objectName.toString()));
    }
    this.name = objectName;
    this.className = className;
}
Inconsistent Synchronization

• Common idiom for thread safe classes is to synchronize on the receiver object ("this")
• We look for field accesses
  – Find classes where lock on "this" is sometimes, but not always, held
  – Unsynchronized accesses, if reachable from multiple threads, constitute a race condition
Inconsistent Synchronization Example

- GNU Classpath 0.08, java.util.Vector

```java
public int lastIndexOf(Object elem)
{
    return lastIndexOf(elem, elementCount - 1);
}

public synchronized int lastIndexOf(
    Object e, int index)
{
    ...
}
```
Unconditional Wait

- Before waiting on a monitor, the condition should be almost always be checked
  - Waiting unconditionally almost always a bug
  - If condition checked without lock held, could miss the notification

- Example (JBoss 4.0.0DR3):

```java
if (!enabled) {
    try {
        log.debug(...);
        synchronized (lock) {
            lock.wait();
        }
    }
}
```
Bug Categories

• Correctness

• Bad Practice
  – equals without hashCode, bad serialization, comparing Strings with ==, equals should handle null argument

• Dodgy
  – Dead store to local variable, load of known null value, overbroad catch

• Performance

• Multithreaded correctness

• Malicious code vulnerability
Demo

- Live code review
- Available as Java Webstart from
  - http://findbugs.cs.umd.edu/demo/
  - http://findbugs.sourceforge.net/demo.html
Warning Density
Warning density

- Density of high and medium priority correctness warnings

<table>
<thead>
<tr>
<th>Warnings/KNCSS</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>SleepyCat DB</td>
</tr>
<tr>
<td>0.3</td>
<td>Eclipse 3.2</td>
</tr>
<tr>
<td>0.6</td>
<td>JDK 1.5.0_03</td>
</tr>
<tr>
<td>0.6</td>
<td>JDK 1.6.0 b51</td>
</tr>
<tr>
<td>0.9</td>
<td>IBM WebSphere 6.0.3</td>
</tr>
</tbody>
</table>
How we do it
Some detectors are simple

- But specific
- Looking for ignored return values is easy
  - once you know what methods to look at
- value.split(".") also pretty easy
- Experience, taste, and access to lots of bugs is what you need
Some are harder

- Finding uses of `.equals` to compare two objects of different types
  - requires a type analysis
- We do fairly simple analysis: very little interprocedural code analysis
Null pointer analysis

- Where we do a lot of work
- Want to avoid false positives
- Big issue: infeasible paths
An infeasible path?

private int f(Object x, boolean b) {
    int result = 0;
    if (x == null) result++;
    else result--;
    // at this point, we know x is null on a simple path
    if (b) {
        // at this point, x is only null on a complex path
        // we don't know if the path in which x is null
        // and b is true is feasible
        return result + x.hashCode();
    }
    return result;
}
First attempt

• Don’t worry about infeasible paths

• Only report null pointer exceptions that would occur if every statement and branch is covered

• This finds a lot of bugs!

  • with a very low false positive rate
The primary lesson

• You don’t have to be clever to find stupid mistakes

• being stupid works pretty well
A “false” positive

```java
XMLEvent getXMLEvent(XMLStreamReader reader){
    EventBase event = null;
    switch(reader.getEventType()){  
            case XMLEvent.START_ELEMENT:
            event = ...;
            break;
            case XMLEvent.END_ELEMENT:
            event = ...;
            break;
            default:
            throw new MissingDefaultException();
        }
        event.setLocation(reader.getLocation());
        return event ;
    }
```
But clever can find more

- We wanted to find more null pointer bugs
  - wanted to do better than commercial tools that cost $250K
- So we look for situations where a value is known to be null at some statement or branch
- and the value is guaranteed to be dereferenced on all paths to exit
A Guaranteed Dereference

public int f(Object x, boolean b) {
    int result = 0;
    if (x == null) result++;
    else result--;
    // at this point, we know x is null on a simple path
    if (b) {
        // at this point, x is only null on a complex path
        // we don't know if the path in which x is null
        // and b is true is feasible
        return result + x.hashCode();
    }
    else {
        // at this point, x is only null on a complex path
        // we don't know if the path in which x is null
        // and b is false is feasible
        return result - x.hashCode();
    }
}
Advantages of not being too clever

• If your analysis tries to be very clever, and do context sensitive alias resolution and interprocedural analysis

• any developer is going to have to duplicate that analysis to understand your bug report

• they need to be able to understand it in order to fix it
Overall improvements

- We put a lot of effort into improving our null pointer analysis
  - field tracking
  - guaranteed dereferences

- FindBugs 1.1 finds about twice as many null pointer bugs as FindBugs 1.0
  - without an increase in false positives
The questions I want to answer
What is the fruit distribution?

• FindBugs looks for low hanging fruit

• Where is the best place to expend effort to find more bugs?

• Use more sophisticated analysis to find more subtle errors

• Build more shallow and general bug detectors

• Write application-specific bug detectors
What kinds of errors can be detected by static analysis?

• I never would have thought to look for recursive infinite loops

• Or doing an integer division, converting the result to a double, and passing the result to Math.ceil

• Easy to measure false positives, hard to measure false negatives:
  • defects that could be detected by static analysis but aren’t
Turning bug instances into bug patterns

- We need to change our software development process so that we learn from our mistakes
- Evaluate bugs, see if they can be turned into bug patterns
  - many bug patterns manifest themselves over and over again
- Example: the flaw identified in most binary search implementations
Examples of turning bugs into bug patterns

• I read through all the bugs fixed in each build of Sun’s JDK

• Example: In build 89, they fixed a serialization bug in ArrayBlockingQueue

• In 5 hours of work, I wrote and tuned a bug detector for that bug pattern

• Found 17 other erroneous classes in the JDK
Specific details of bug

- Class was serializable, but had a transient field
- that wasn’t reset by a readObject or readResolve method
- Had to tweak priorities for detector
  - raise priority if set to non-default value in constructor, or if set in multiple places
How can we make it easy to write bug detectors?

• We want to allow as many developers as possible to write their own bug detectors
  • some will be generally applicable, some specific to particular projects

• What tools/analysis/pattern languages do we need?
  • now starting to have enough samples to think about this
How can we make static analysis pervasive?

- State of the art static analysis has a lot to offer, more than many people suspected.
- What are the practical and cultural issues that need to be surmounted to make it pervasive?
  - false positive suppression: no one wants to review a false positive more than once.
  - other points of pain?
My Other Cool Project
Marmoset: an advanced project testing framework
Marmoset

A total rethinking of how student programming projects are submitted and tested
designed to provide students, instructors and researchers with lots of feedback, including feedback before submission deadline
Collecting large data sets of student efforts, starting to learn lots of stuff about how students learn to program
Previous Practice

- Everyone agrees we can't just distribute all the test cases to students.
- Instructor has secret test cases.
  - Sometimes not made up until time to grade the project.
- Student code run against secret tests by TA after project deadline.
Release Testing

- If a submission passing all of the public tests, students are given the option to release test their submission.
- Given limited information from a release test.
- Limited opportunities for release tests.
Marmoset

- Students are told # of release tests passed and failed.
- And names of first two release tests that failed.
- For example, on Poker project, might be told that they “fail FourOfAKind, FullHouse, and 2 other tests.”
- Release testing consumes a token.
- Students receive 2-3 tokens.
- Tokens regenerate 24 hours after used.
Advantages of release testing

- Encourages students to think, develop their own tests.
- Gives students an indication of where they are, whether that are having trouble.
- Gives students an incentive to start working early.
- Instructors get live feedback about student progress before project deadline.
Marmoset Research Study

- Students asked consent to participate in research study
- Eclipse plugin captures each save as students work on their projects
- Research database of more than 200,000 snapshots, each of which is run against all the test cases
- ask questions such as what leads to null pointer exceptions in student code
Marmoset data

- From 4 semesters of a CS 2 course
- 147,595 snapshots of student work
- 2,171,812 unit test runs

Exceptions include:

- 31,454 null pointer exceptions
- 8,122 class cast exceptions
- 5,453 index out of bounds
- 4,996 array index out of bounds
- 3,754 stack overflows
Questions?