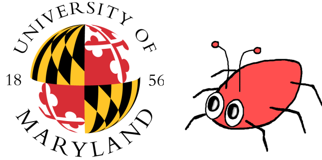
Using Static Analysis to Find Bugs

David Hovemeyer



April 28, 2005 — CMSC 433

Bugs in software

- Programmers are smart people
- We have good techniques for finding bugs early:
  - Unit testing, pair programming, code inspections
- So, most bugs should be subtle, and require sophisticated analysis techniques to find
  - Right?

- Apache Ant 1.6.2, org.apache.tools.ant.taskdefs.optional.metamata.MAudit

```
if (out == null) {
    try {
        out.close();
    } catch (IOException e) {
    }
}
```


- Eclipse 3.0.1, org.eclipse.update.internal.core.ConfiguredSite

```
if (in == null)
    try {
        in.close();
    } catch (IOException e1) {
    }
```


- Eclipse 3.0.1, org.eclipse.jdt.internal.debug.ui.JDIModelPresentation

```
if (sig != null || sig.length() == 1) {
  return sig;
}
```


- Eclipse 3.0.1, org.eclipse.jdt.internal.ui.compare.JavaStructureDiffViewer

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

- JBoss 4.0.0RC1, org.jboss.cache.TreeCache

```
int treeNodeSize=fqn.size();
if(fqn == null) return null;
```


One more...

- J2SE version 1.5 build 63 (released version), java.lang.annotation.AnnotationTypeMismatchException

```
public String foundType() {
   return this.foundType();
}
```


- J2SE version 1.5 build 63 (released version), java.lang.annotation.AnnotationTypeMismatchException

```
public String foundType() {
   return this.foundType();
}
```

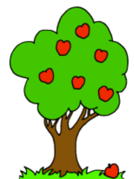
Written by Josh Bloch, author of Effective Java

Bug-finding tools and techniques

- Much research has been done on static program analysis techniques to find bugs
- Recent research has moved towards increasingly more sophisticated analysis techniques
- Our question: what bugs can be found using *simple* analysis techniques?

Where is the low-hanging fruit?

- *Bug-driven* research: start by looking at real bugs, then think of ways to find similar bugs
  - Using simplest possible analysis techniques
- Try bug finders on real software
- Result: we found a surprising number of obvious bugs in production software



Talk overview

In this talk I will

- Discuss ways to find bugs in software
- Demonstrate that simple static analysis techniques can find lots of bugs in real software

Finding bugs in software

Testing

- Run the program, see if it behaves correctly
- Limitations:
  - Error handling code is difficult to test
  - Threading bugs can be very hard to reproduce
  - Test scaffolding is time-consuming to create

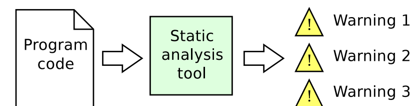
Code inspection

- Manually examine source code, look for bugs
- Limitations:
  - Labor intensive
  - Subjective: source code might appear to be correct when it is is not
    - Can you spot the typo in this slide?
    - People have similar blind spots reading source code

Code inspection

- Manually examine source code, look for bugs
- Limitations:
  - Labor intensive
  - Subjective: source code might appear to be correct when it is is not
    - Can you spot the typo in this slide?
    - People have similar blind spots reading source code

Static analysis



- Idea: *automated code inspection*
- Use a program to analyze your program for bugs
  - Analyze statements, control flow, method calls
- Advantages over testing and manual code inspection:
  - Can analyze many potential program behaviors
  - Doesn't get bored
  - Relatively objective

Limits of static analysis

- Nontrivial properties of programs are *undecidable*

  "Does program P have bug X?"
  $\equiv$ "Can program P reach state X?"
  $\equiv$ Halting problem

- Static analysis can (in general) never be fully precise, so it must *approximate* the behavior of the program

Approximating towards completeness

- We could design a bug-finding analysis so that it always overestimates possible program behaviors
  - *Never* misses a bug, but might report some false warnings
- Problem: the analysis may report so many false warnings that the real bugs cannot be found!
  - Trivial version: report a bug at every point in the program

Approximating towards soundness

- We could design a bug-finding analysis so that it always underestimates possible program behaviors
  - Never reports a false warning, but might miss some real bugs
- Problem: analysis may not find as many bugs as we would like
  - Trivial version: never report any warnings

Heuristic analysis

- A static analysis to find bugs does not need to be *consistent* in its approximations
  - Neither complete nor sound: miss some real bugs, and report some false warnings
- This gives the analysis more flexibility to estimate *likely* program behaviors
- May allow the analysis to be more precise in general

Practical issues

- Say your program has 100 real bugs
- Would you rather use
  - A tool that finds all 100 bugs, but reports 1,000,000 warnings
  - A tool that finds only 25 bugs, but reports 50 warnings
- Using a bug-finding tool must be a productive use of the developer's time
- No useful tool will find *every* bug

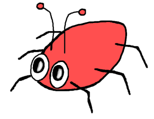
Bug patterns

Bug patterns

- Not all bugs are subtle and unique
- Many bugs share common characteristics
- A *bug pattern* is a code idiom that is usually a bug
  - Detection of many bug patterns can be automated using simple analysis techniques

The FindBugs tool

- We have implemented automatic detectors for about 100 bug patterns in a tool called FindBugs
  - Open source
  - http://findbugs.sourceforge.net
- Analyzes Java bytecode



Java bytecode

- Compiling and running a Java program:
  1. Source code compiled to class files containing *bytecode*
  2. Bytecode executed by the *Java virtual machine* (JVM)
- Bytecode is the machine language of the JVM
- Stack-based:
  - Most bytecode instructions work by pushing values onto or consuming values from the *operand stack*
  - Local variables are used for method parameters and longer-lived values
    - They are analogous to CPU registers

Bytecode example

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, world");
  }
}
```

```
getstatic       #2; //Field System.out
ldc             #3; //String "Hello, world"
invokevirtual   #4; //Method PrintStream.println(String)
return
```


Bytecode analysis techniques

Implementation techniques used in FindBugs:

- Examine class hierarchy
  - Find methods implemented or overridden improperly
  - Easy to implement, very accurate for finding some kinds of bugs
- Linear bytecode scan
  - Use a state machine to search for suspicious instruction sequences
  - Can find bugs involving short code sequences
- Dataflow analysis
  - Symbolically execute methods (keeping track of values)
  - Look for places where values are used in a suspicious way

Example bug patterns

What is wrong with this class?

```
public class Person {
  private String firstName, lastName;

  public boolean equals(Person other) {
    return this.firstName.equals(other.firstName)
        && this.lastName.equals(other.lastName);
  }
}
```


What is wrong with this class?

```
public class Person {
  private String firstName, lastName;

  public boolean equals(Person other) {
    return this.firstName.equals(other.firstName)
        && this.lastName.equals(other.lastName);
  }
}
```


Covariant equals

- When defining an equals() method, the parameter must be of type Object
  - Otherwise it doesn't override the equals() method in the base Object class
- Why is this bad?
  - Container classes (like hash tables) need to use equals(Object)
  - A covariant equals() method won't be called
- Found: 15 cases in core Java 1.5 libraries, 4 in Eclipse 3.0, 2 in JBoss 4.0.0RC1

What is wrong with this code?

From JBoss 4.0.0RC1

```
if (!enabled) {
  log.debug("Disabled, waiting for notification");
  synchronized (lock) {
    lock.wait();
  }
}
```


Unconditional Wait

Thread 1

```
// If we are not enabled, then wait
if (!enabled) {


  try {
    synchronized (lock) {
      lock.wait();
```

Thread 2

```
synchronized (lock) {
  enabled = true;
  lock.notifyAll();
}
```


Unconditional Wait

Thread 1

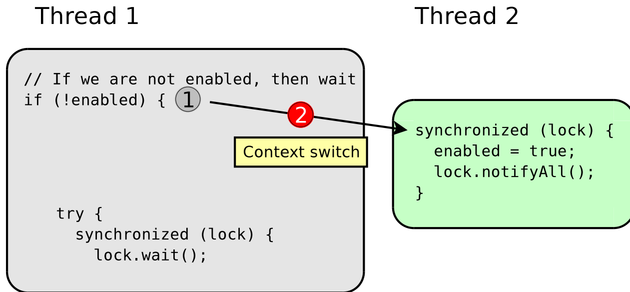
```
// If we are not enabled, then wait
if (!enabled) {  1  Condition is false


  try {
    synchronized (lock) {
      lock.wait();
```

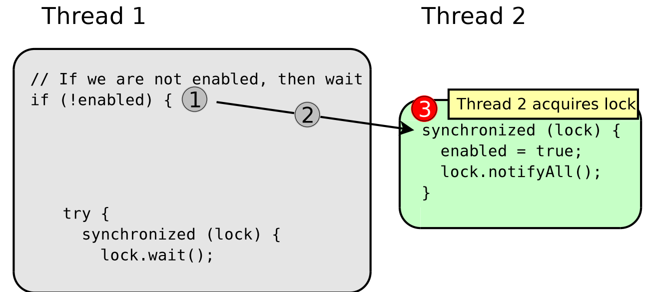
Thread 2

```
synchronized (lock) {
  enabled = true;
  lock.notifyAll();
}
```

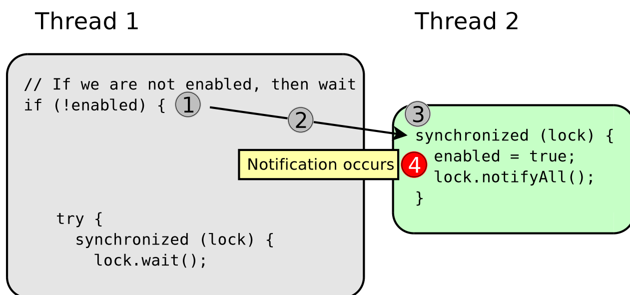
Unconditional Wait



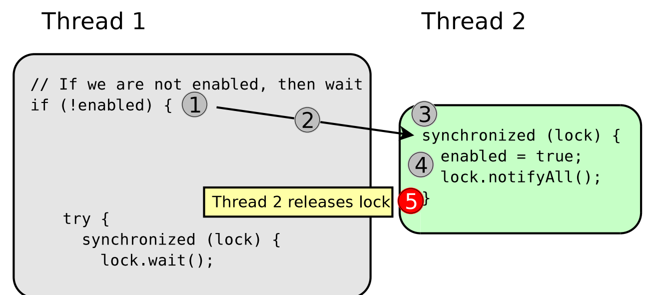
Unconditional Wait



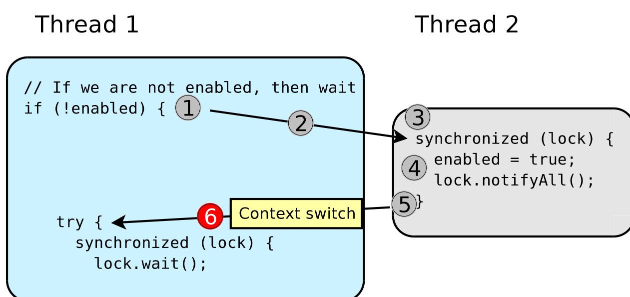
Unconditional Wait



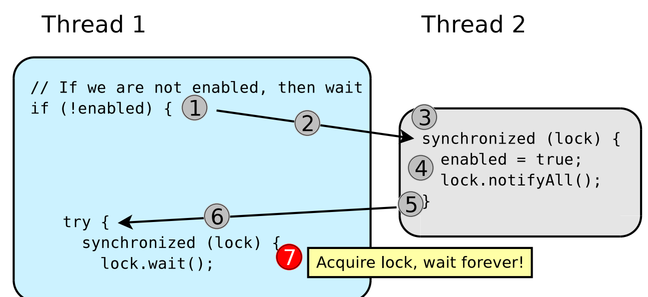
Unconditional Wait



Unconditional Wait



Unconditional Wait



Correct code to wait on a condition

- Correct code for waiting on a condition:

```
synchronized (lock) {    Acquire lock
  if (!condition) {      Check condition
    lock.wait();         Wait for notification
  }
}
```

- Lock must be held while checking condition and waiting
- Otherwise the notification could be missed

Detecting Unconditional Wait

- Idea:
  - If lock acquisition is immediately followed by a wait,
  - the condition was probably checked without the lock held
- Look for sequence containing instructions:
  - `monitorenter`
  - `invokevirtual Object.wait()`
- No branches between acquiring lock and waiting imply presence of bug
- Found 3 real bugs in core Java 1.5 libraries, 2 in Eclipse 3.0, 2 in JBoss 4.0.0RC1

What is wrong with this code?

From JBoss 4.0.0RC1

```
public String getContentId()
{
   String[] header = getMimeHeader("Content-Id");
   String id = null;
   if( header != null || header.length > 0 )
      id = header[0];
   return id;
}
```


What is wrong with this code?

From JBoss 4.0.0RC1

```
public String getContentId()
{
   String[] header = getMimeHeader("Content-Id");
   String id = null;
   if( header != null || header.length > 0 )
      id = header[0];
   return id;
}
```

This one was fairly obvious

Null pointer dereferences

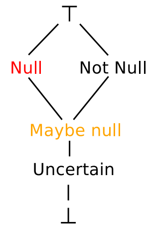
- Some null pointer deferences require sophisticated analysis to find
  - Analyzing across method calls, modeling the contents of heap objects
- We have seen many examples of *obvious* null pointer dereferences:
  - Values which are always null
  - Values which were null on some control path
- How can we construct an analysis to find obvious null pointer dereferences?

Dataflow analysis

- At each point in a method, keep track of *dataflow facts*
  - E.g., which local variables and stack locations might contain null
- Symbolically execute the method:
  - Model instructions
  - Model control flow
  - Iterate until a fixed point solution is reached

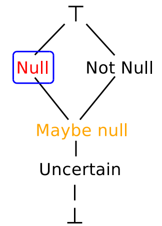
Dataflow values

- Model values of local variables and stack operands using lattice of symbolic values
- When to control paths merge, use *meet* operator to combine values
  - This is the greatest lower bound of the values



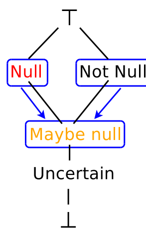
Meet example

Null ◇ Null = Null



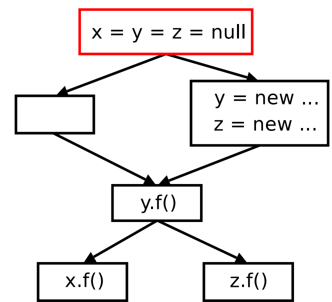
Meet example

Null ◇ Not null = Maybe null



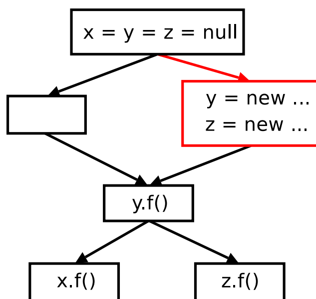
Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
  y = new ...;
  z = new ...;
}
y.f();
if (cond2)
  x.f();
else
  z.f();
```



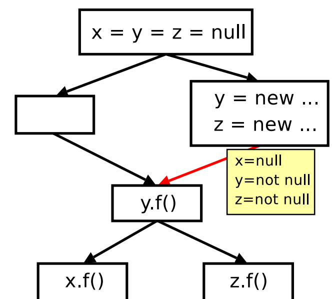
Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
  y = new ...;
  z = new ...;
}
y.f();
if (cond2)
  x.f();
else
  z.f();
```



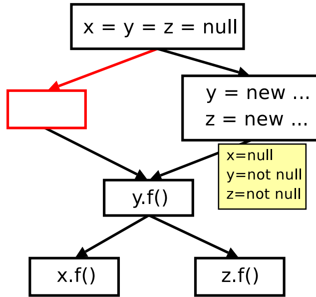
Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
  y = new ...;
  z = new ...;
}
y.f();
if (cond2)
  x.f();
else
  z.f();
```



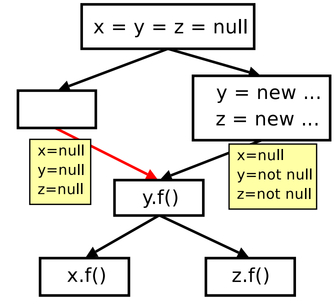
Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
  y = new ...;
  z = new ...;
}
y.f();
if (cond2)
  x.f();
else
  z.f();
```



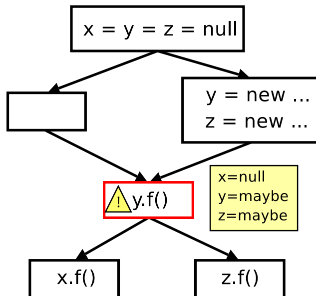
Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
  y = new ...;
  z = new ...;
}
y.f();
if (cond2)
  x.f();
else
  z.f();
```



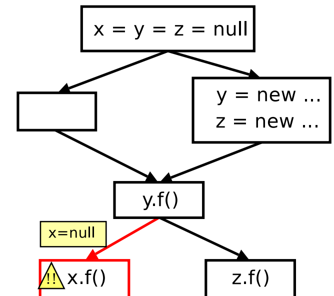
Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
  y = new ...;
  z = new ...;
}
y.f();
if (cond2)
  x.f();
else
  z.f();
```



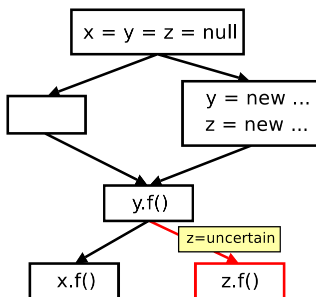
Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
  y = new ...;
  z = new ...;
}
y.f();
if (cond2)
  x.f();
else
  z.f();
```



Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
  y = new ...;
  z = new ...;
}
y.f();
if (cond2)
  x.f();
else
  z.f();
```



Conclusions

- There are more obvious bugs lurking in Java code than most people realize
  - Static analysis can find many of these
- Lots of interesting properties of programs can be found using static analysis
  - Very active research area
  - Starting to be widely adopted in practice
  - The compiler course is still relevant!