



Lecture 14 — FindBugs

David J. Pearce

*School of Engineering and Computer Science
Victoria University of Wellington*

What is FindBugs?



*“FindBugs is an open source program created by Bill Pugh and David Hovemeyer which looks for bugs in Java code. **It uses static analysis to identify hundreds of different potential types of errors in Java programs.** Potential errors are classified in four ranks: (i) scariest, (ii) scary, (iii) troubling and (iv) of concern. This is a hint to the developer about their possible impact or severity. FindBugs operates on Java bytecode, rather than source code.”*

–Wikipedia

Bug Patterns



FindBugs Bug Descriptions

This document lists the standard bug patterns reported by [FindBugs](#) version 3.0

Summary

Description

[BC: Equals method should not assume anything about the type of its argument](#)

[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](#)

[CNT: Rough value of known constant found](#)

[Co: Abstract class defines covariant compareTo\(\) method](#)

- FindBugs looks for **Bug Patterns**:
 - (Also called **code smells**)
 - Patterns don't **guarantee** bugs
 - Patterns indicate **potential** bugs
 - Patterns are **simple** to check, hence typically **superficial**

Bug Patterns (Cont'd)

```
class Point {  
    private int x, y;  
  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
  
    public boolean equals(Point p) {  
        return this.x == p.x && this.y == p.y;  
    }  
    ...  
}
```

- **Pattern 1:** Equals method should not assume anything about type of its argument.

Bug Patterns (Cont'd)

```
class Point {  
    private int x, y;  
  
    public Point(int x, int y) {  
        x = x; y = y;  
    }  
  
    ...  
}
```

- **Pattern 2:** Self Assignment of local rather than assignment to field.

Bug Patterns (Cont'd)

```
class Point {  
    private int x, y;  
  
    public void add(Point p) {  
        if(p == null) { x = x + p.x; y = y + p.y; }  
    }  
    public boolean lessThan(Point p) {  
        return p != null || (x < p.x && y < p.y);  
    }  
    ...  
}
```

- **Pattern 3:** Null pointer dereference.

(Eclipse checks for these now)

Bug Patterns (Cont'd)

```
class Point {
    private int x, y;

    public Point(int x, int y) { ... }

    public boolean equals(Object o) {
        if(o instanceof Point) {
            Point p = (Point) o;
            return this.x == p.x && this.y == p.y;
        } else {
            return false;
        }
    }
    ...
}
```

- **Pattern 4:** Repeated conditional tests.

Bug Patterns (Cont'd)

```
class Point {  
    private int x, y;  
  
    public Point(int x, int y) { ... }  
  
    public boolean equals(Object o) {  
        if(o instanceof Point) {  
            Point p = (Point) o;  
            return this.x == p.x && this.y == p.y;  
        } else {  
            return false;  
        }  
    }  
}
```

- **Pattern 5:** Class defines `equals()` but not `hashCode()`.

Bug Patterns Categories

- **Class & Inheritance Structure.** These bug detectors look only at structural properties.
- **Linear Code Scan.** These make linear scan through functions, ignoring control flow.
- **Control-Flow Sensitive.** These make use of control flow graph to find bugs.
- **Control & Data Flow.** These model both data and control-flow to find bugs.

See *“Finding Bugs is Easy”*, Hovemeyer and Pugh.

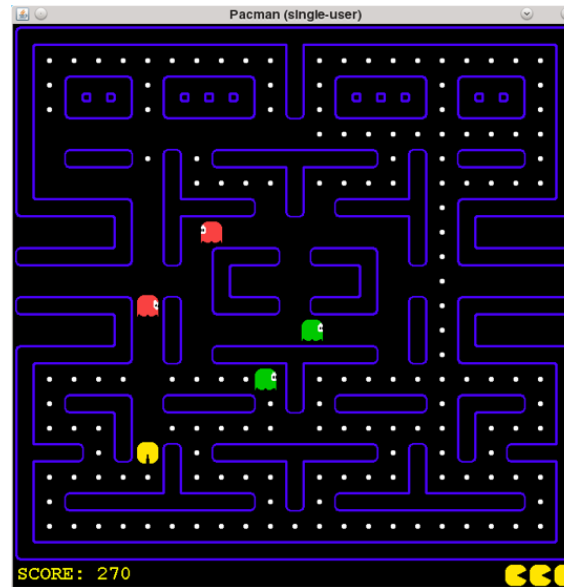
Soundness

- **False Positives.** A warning reported where no problem actually exists in practice.
- **False Negatives.** A problem actually exists, and the warning should have been reported (but was not).

FindBugs suffers both issues.

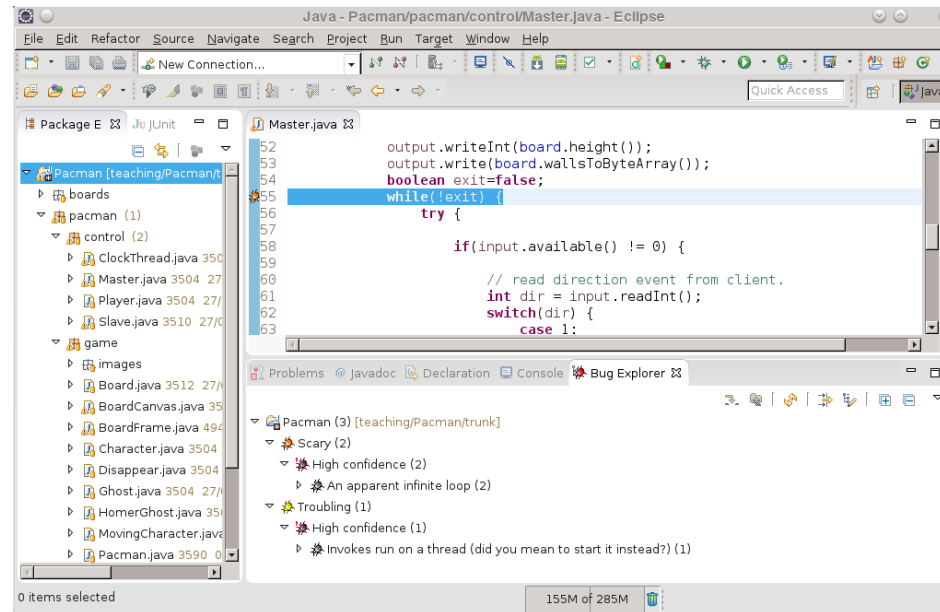
Example: Pacman

Metrics:



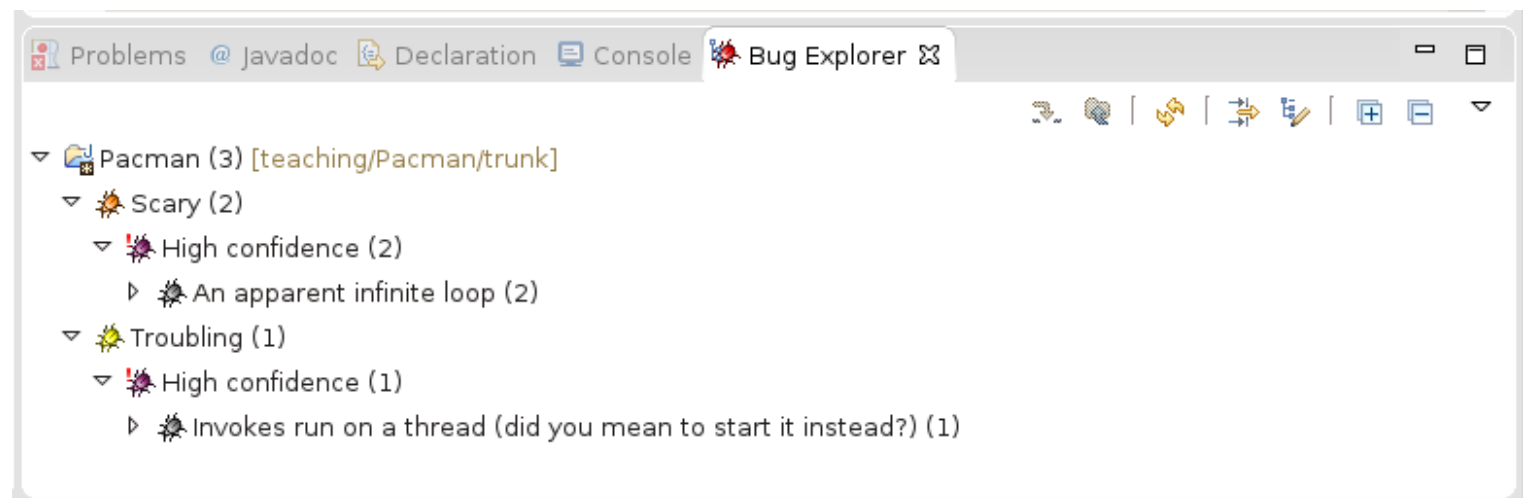
- **2458** Lines of Code (LOC) including whitespace
- **1502** Lines of Code (LOC) excluding whitespace, 722 Comment Lines
- **15** Classes
- **4** Packages

Example: Pacman (Cont'd)



The screenshot shows the Eclipse IDE interface. The main editor displays the code for `Master.java` in the `control` package. The code includes a `while` loop that reads input from a client and processes it based on direction. The `while` loop is highlighted in blue. The `Bug Explorer` window is open at the bottom, showing a tree view of bugs for the `Pacman (3)` project. The bugs are categorized by confidence and type, including 'High confidence (2)', 'Troubling (1)', and 'High confidence (1)'. The status bar at the bottom indicates '0 items selected' and '155M of 285M'.

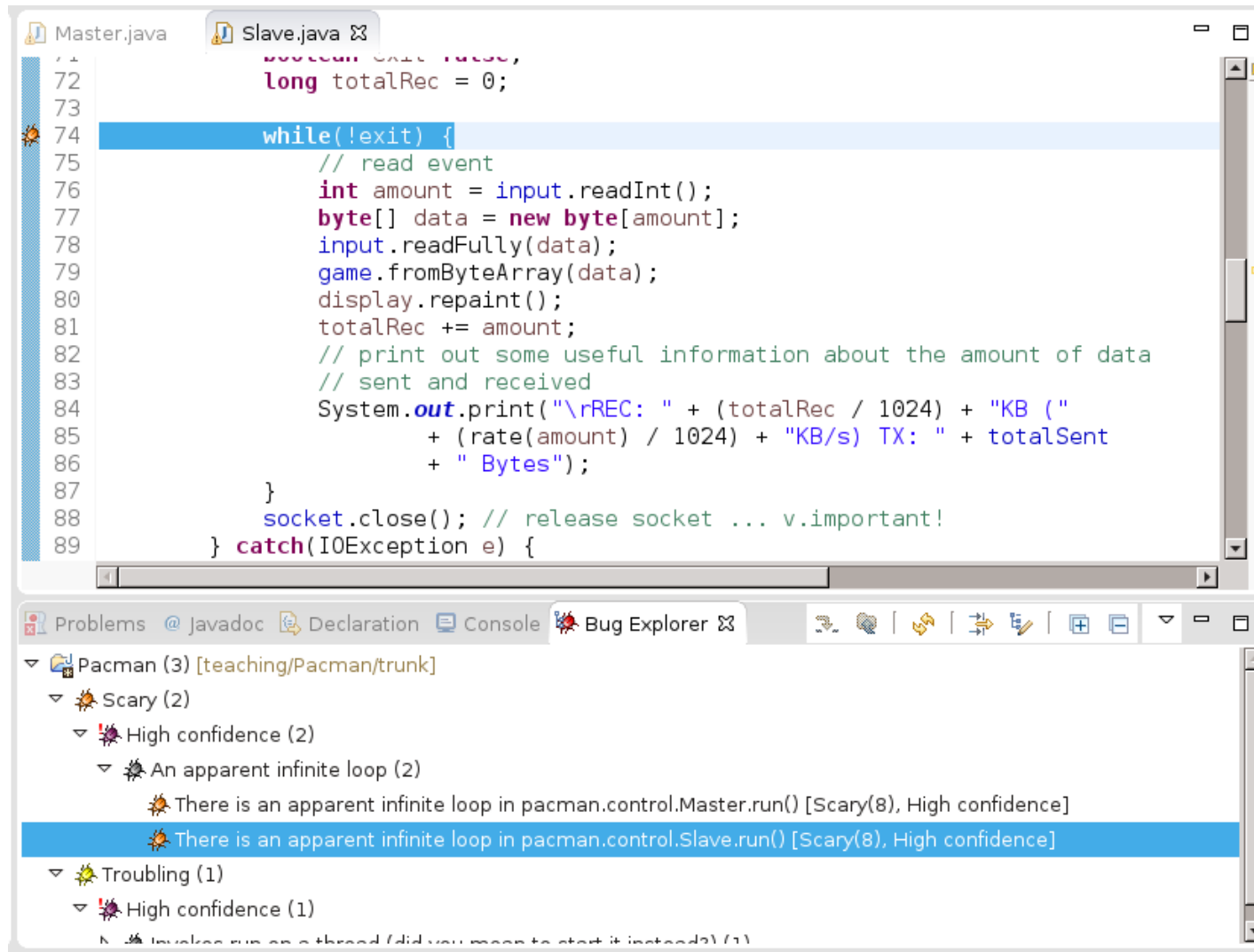
```
52     output.writeInt(board.height());
53     output.write(board.wallsToByteArray());
54     boolean exit=false;
55     while(!exit) {
56         try {
57             if(input.available() != 0) {
58                 // read direction event from client.
59                 int dir = input.readInt();
60                 switch(dir) {
61                     case 1:
```



This is a close-up view of the `Bug Explorer` window. The window title bar includes tabs for `Problems`, `Javadoc`, `Declaration`, `Console`, and `Bug Explorer`. The main area shows a tree view of bugs for the `Pacman (3)` project. The bugs are organized as follows:

- ▼ Pacman (3) [teaching/Pacman/trunk]
 - ▼ Scary (2)
 - ▼ High confidence (2)
 - ▶ An apparent infinite loop (2)
 - ▼ Troubling (1)
 - ▼ High confidence (1)
 - ▶ Invokes run on a thread (did you mean to start it instead?) (1)

Example: Pacman (Cont'd)



```
Master.java  Slave.java ✖
72
73
74 while(!exit) {
75     // read event
76     int amount = input.readInt();
77     byte[] data = new byte[amount];
78     input.readFully(data);
79     game.fromByteArray(data);
80     display.repaint();
81     totalRec += amount;
82     // print out some useful information about the amount of data
83     // sent and received
84     System.out.print("\rREC: " + (totalRec / 1024) + "KB ("
85         + (rate(amount) / 1024) + "KB/s) TX: " + totalSent
86         + " Bytes");
87 }
88 socket.close(); // release socket ... v.important!
89 } catch(IOException e) {
```

Problems @ Javadoc Declaration Console Bug Explorer ✖

- ▼ Pacman (3) [teaching/Pacman/trunk]
 - ▼ Scary (2)
 - ▼ High confidence (2)
 - ▼ An apparent infinite loop (2)
 - ☀ There is an apparent infinite loop in pacman.control.Master.run() [Scary(8), High confidence]
 - ☀ There is an apparent infinite loop in pacman.control.Slave.run() [Scary(8), High confidence]
 - ▼ Troubling (1)
 - ▼ High confidence (1)
 - ☀ Involves run on a thread (did you mean to start it instead?) (1)

Q) *Why does FingBugs think there is an infinite loop?*

Example: Pacman (Cont'd)

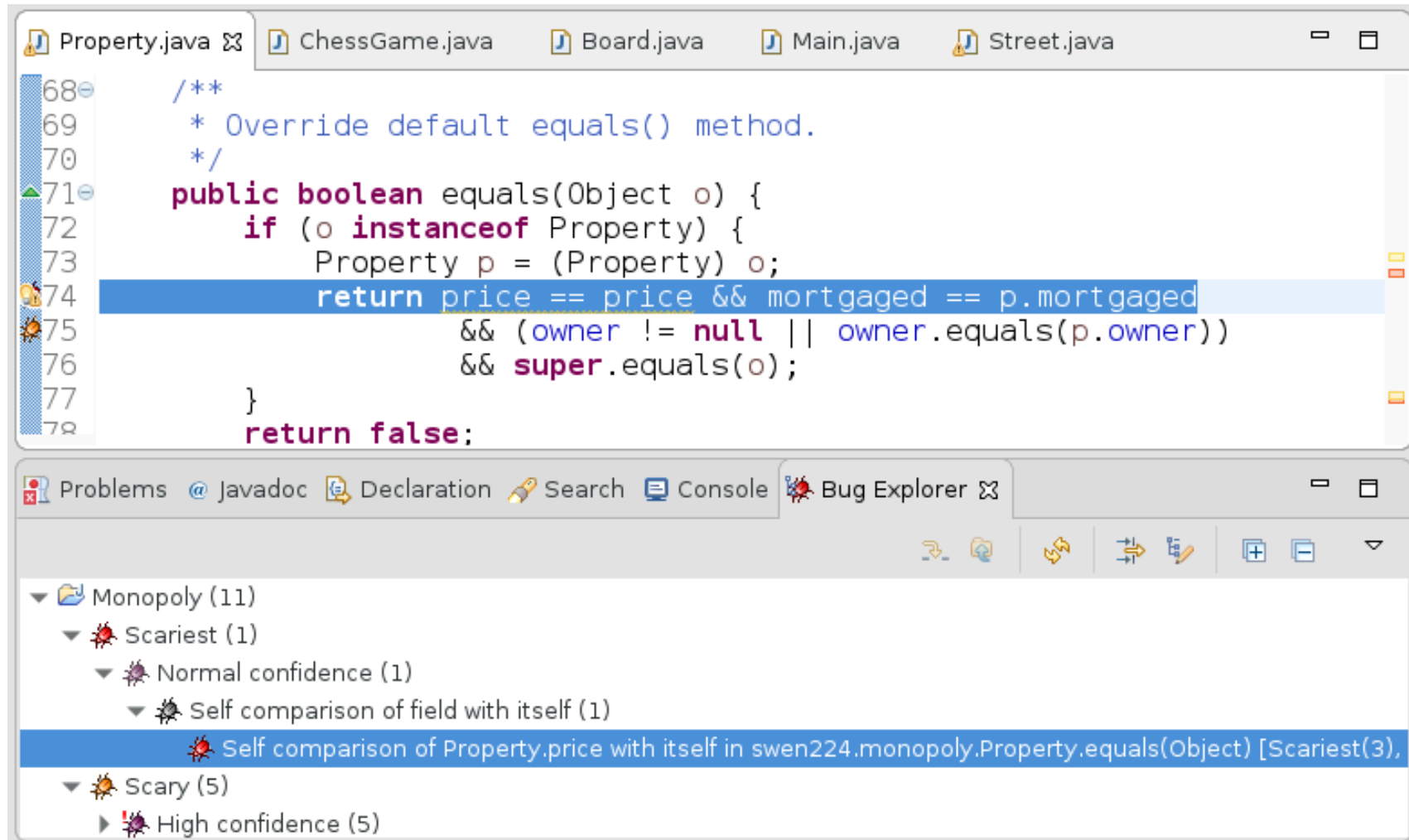
```
133     System.out.println(p[1]);
134     }
135     }
136
137     private static void runClient(String addr, int port) throws IOException {
138         Socket s = new Socket(addr,port);
139         System.out.println("PACMAN CLIENT CONNECTED TO " + addr + ":" + port);
140         new Slave(s).run();
141     }
142
143     private static void runServer(int port, int nclients, int gameClock, int broad
144         ClockThread clk = new ClockThread(gameClock,game,null);
145
```

Problems @ Javadoc Declaration Console Bug Explorer

- ▼ Pacman (3) [teaching/Pacman/trunk]
 - ▶ Scary (2)
 - ▼ Troubling (1)
 - ▼ High confidence (1)
 - ▼ Invokes run on a thread (did you mean to start it instead?) (1)
 - ☀ pacman.Main.runClient(String, int) explicitly invokes run on a thread (did you mean to start it instead?) [Troubling]

Q) Do you agree with FindBugs?

Example: Monopoly



The image shows a screenshot of an IDE with a Java code editor and a Bug Explorer window. The code editor displays the `equals()` method in `Property.java`, with a blue highlight on the line `return price == price && mortgaged == p.mortgaged`. The Bug Explorer window shows a list of bugs, with the following entry selected:

- Monopoly (11)
 - Scariest (1)
 - Normal confidence (1)
 - Self comparison of field with itself (1)
 - Self comparison of Property.price with itself in swen224.monopoly.Property.equals(Object) [Scariest(3),**
- Scary (5)
 - High confidence (5)

Further Reading

- *Finding Bugs is Easy*, Hovemeyer & Pugh, OOPSLA 2004.
- *Using Static Analysis for Software Defect Detection*, Bill Pugh, Google Talk, 2006. <https://youtu.be/8eZ8YWV1-2s>
- *Using Static Analysis to Find Bugs*. Ayewah, Hovemeyer, Morgenthaler, Penix & Pugh, IEEE Software, 2008.
- *A comparison of bug finding tools for Java*, Rutar, C.B. Almazan & J.S. Foster, ISSRE'04.
- *Scaling static analyses at Facebook*, Communications of the ACM, 2019.
- *Lessons from building static analysis tools at Google*, Communications of the ACM, 2018.
- *Why don't software developers use static analysis tools to find bugs?* Johnson, Song, Murphy-Hill & Bowdidge, ICSE'13.