Defective Java Code: Mistakes That Matter

William Pugh
Univ. of Maryland
Defective Java Code
Learning from mistakes

> I’m the lead on FindBugs
  • static analysis tool for defect detection

> Visiting scientist at Google for the past 10 months
  • learned a lot about coding mistakes, which ones matter, how to catch them, how to allow a community to review them

> A little like programming puzzlers
  • but no quiz
  • and lots of interspersed commentary
Static analysis

> Analyzes code without running it
> FindBugs is an open source static analysis tool, developed at the University of Maryland
  • with a number of additional contributors
  • Looks for bug patterns, inspired by real problems in real code
> Held FindBugs fixit at Google May 13-14th
  • 300 engineers provided 8,000 reviews of 4,000 issues
    • 75+% were marked should fix or must fix
    • more than 1,500 of the issues have already been removed
Learned wisdom

> Static analysis typically finds mistakes
  • but some mistakes don’t matter
  • need to find the intersection of stupid and important

> The bug that *matter* depend on context

> Static analysis, *at best*, might catch 5-10% of your software quality problems
  • 80+% for certain specific defects
  • but overall, not a magic bullet

> Used effectively, static analysis is cheaper than other techniques for catching the same bugs
Null bug

> From Eclipse, 3.5RC3:
org.eclipse.update.internal.ui.views.FeatureStateAction

    if (adapters == null && adapters.length == 0)
    return;

> Clearly a mistake

  • First seen in Eclipse 3.2
  • but in practice, adapters is probably never null

> Is there any impact from this?

  • we would probably notice a null pointer exception
  • we don’t immediately return if length is 0
Cost when a mistake causes a fault/failure

> How quickly/reliability would you notice?
> What is the impact of the misbehavior caused by the mistake?
> How easily could you diagnose the problem and the fix?
> What is the cost to deliver a fix?
Mistakes in web services

- Some mistakes would manifest themselves by throwing a runtime exception
  - Should be logged and noticed
- If it isn’t happening now, a change might cause it to start happening in the future
  - But if it does, the exception will likely pinpoint the mistake
  - And pushing a fix into production is cheaper than pushing a fix to desktop or mobile applications
Expensive mistakes (your results may vary)

> Mistakes that might cost millions of dollars on the first day they manifest

> Mistakes that silently cause the wrong answer to be computed
  • might be going wrong now, millions of times a day
  • or might be OK now, but when it does go wrong, it won’t be noticed until somewhere downstream of mistake

> Mistakes that are expensive or impossible to fix
Using reference equality rather than `.equals` from Google’s code (no one is perfect)

class MutableDouble {
    private double value_;
    public boolean equals(final Object o) {
        return o instanceof MutableDouble && ((MutableDouble)o).doubleValue() == doubleValue();
    }
    public Double doubleValue() {
        return value_;
    }
}
Using `==` to compare objects rather than `.equals`

- For boxed primitives, `==` and `!=` are computed using pointer equality, but `<`, `<=`, `>`, `>=` are computed by comparing unboxed primitive values.
- Sometimes, equal boxed values are represented using the same object.
  - but only sometimes
- This can bite you on other classes (e.g., `String`)
  - but boxed primitives is where people get bit
Heisenbugs vs. deterministic bugs

> A Heisenbug is a mistake that only sometimes manifests itself (e.g., a data race)

> Testing not likely to show error
  - if a test fails, rerunning the test may succeed

> Can be very nasty to track down, impossible to debug

> But how dangerous is a bug that only bites once out of 4 billion times?
Ignoring the return value of putIfAbsent

org.jgroups.protocols.pbcast.NAKACK
ConcurrentMap<Long,XmitTimeStat>
xmit_time_stat = ...;

.....
XmitTimeStat stat = xmit_time_stats.get(key);
if(stat == null) {
    stat = new XmitTimeStat();
    xmit_time_stats.putIfAbsent(key, stat);
}
stat.xmit_reqs_received.addAndGet(rcvd);
stat.xmit_rsps_sent.addAndGet(sent);
misusing putIfAbsent

> ConcurrentMap provides putIfAbsent
  • atomically add key $\rightarrow$ value mapping
    • but only if the key isn’t already in the map
  • if non-null value is returned, put failed and value returned is the value already associated with the key

> Mistake:
  • ignore return value of putIfAbsent, and
  • reuse value passed as second argument, and
  • matters if two callers get two different values
org.jgroups.protocols.pbcast.NAKACK

XmitTimeStat stat=xmit_time_stats.get(key);
if(stat == null) {
    stat=new XmitTimeStat();
    XmitTimeStat stat2
        = xmit_time_stats.putIfAbsent(key, stat);
    if (stat2 != null)
        stat = stat2;
}

stat.xmit_reqs_received.addAndGet(rcvd);
stat.xmit_rsps_sent.addAndGet(sent)
Some lessons

> Concurrency is tricky

> `putIfAbsent` is tricky to use correctly
  • engineers at Google got it wrong more than 10% of the time

> Unless you need to ensure a single value, just use `get` followed by `put` if not found

> If you need to ensure a single unique value shared by all threads, use `putIfAbsent` and check return value
Mistakes That Matter

Unit Testing

System/Integration Testing

Deployment

Static Analysis

Mistakes That Don’t
Static analysis earlier is better

> Find mistakes detected by static analysis before that are detected using more expensive techniques

> Get them to developers while the code is still fresh in developers heads, before anyone else is depending on it or using it

  • Fixing a mistake in code last touched 6 months or 6 years ago isn’t fun

> Of course, this only applies if your mistakes are generally caught by other steps in your quality assurance process at reasonable cost
public void doGet(HttpServletRequest req, HttpServletResponse res) {
    ...
    String target = req.getParameter("url");
    InputStream in = this.getClass()
        .getResourceAsStream("META-INF/resources/
        + target;
    if (in == null) {
        res.getWriter().println(
            "<p>Unable to locate resource: "
            + target);
        return;
    }
}
Cross-site scripting

- Putting untrusted/unchecked data directly into generated html
  - can contain Javascript, which gets executed in your context
  - untrusted input can be injected into your database, or through a URL query parameter
    - via a link sent from attacker to victim
Cross site scripting

**Attacker**

```
<a href="http://host/index.html?variable=<script>...
</script>">Check this out</a>
```

**Trusted WebSite**

html response contains script injected by attacker, but treated by victim’s web browser as though it came from trusted web site

**Victim**
Security vulnerabilities

> Not exposed by normal/expected use cases
> Need some combination of:
  • architectural risk analysis
  • careful design
  • static analysis
  • dynamic testing and analysis
> FindBugs only does simple, shallow analysis for network security vulnerabilities
Returning references to internal mutable state

jdk1.7.0-b59
sun.security.x509.InvalidityDateExtension:

private Date date;
public Object get(String name) {
    if (name.equalsIgnoreCase(DATE)) {
        return date;
    } else {...}
}
Vulnerability to malicious code

> In some cases, your code should preserve certain safety guarantees even if untrusted code is running in the same JVM

  • An issue for the JDK, not an issue for most web services

> Many cases are easy to check for

> I’ve complained about vulnerabilities in Sun’s JDK at JavaOne every year for several years

  • why stop now?
JDK 7 status report

> Overall, *good* progress over JDK 6
  
  • 188 warnings about mutable static fields in JDK 6
  • 133 warnings in JDK 7
    • 14 new ones, 119 retained from JDK 6

> Some of the new issues ones are trivial to fix
  
  • `com.sun.xml.internal.stream.util.BufferAllocator.LARGE_SIZE_LIMIT` is public, static and non-final

> I can suggest tools to help you with this...
Incomparable equality

```
org.eclipse.jdt.internal.debug.eval.ast.engine.AstInstructionCompiler

SimpleType simpleType = (SimpleType) type;
if ("java.lang.String".equals(simpleType.getName()))
    return Instruction.T_String;
```

> SimpleType.getName() returns a org.eclipse.jdt.core.dom.Name

> In Eclipse since 2.0 (June 2002)
Many variations, assisted by weak typing in APIs

- Using `.equals` to compare incompatible types
- Using `.equals` to compare arrays
  - only checks if the same array
- Checking to see if a `Set<Long>` contains an `Integer`
  - never found, even if the same integral value is contained in the map
- Calling `get(String)` on a `Map<Integer,String>`
Silent, nasty bugs

> Very hard to find these bugs by inspection
  * types not always visible/explicit
> In some cases, could be introduced by refactoring
  * Change the key type of a `Map` from `Integer` to `Long`
  * Fix all the places where you get type errors
  * Leave behind bugs
> Google had an issue with a refactoring that changed a method to return `byte[]` rather than `String`
  * introduced silent errors
Bug introduced between Eclipse 3.5RC1 and RC2

org.eclipse.pde.internal.build.BrandingIron

File rootFolder
   = getCanonicalFile(new File(initialRoot));
if (!rootFolder.equals(target)) {
   rootFolder.delete();
   ...
}
Lost logger

```java
void initLogger() {
    Logger logger = Logger.getLogger("edu.umd.cs");
    logger.addHandler(new FileHandler());
    logger.setUseParentHandlers(false);
}
```

> Loggers are retained by weak references
  > always allowed by spec, recent change to OpenJDK implementation

> If GC happens immediately after the call to `initLogger`, changes to logger will be lost
Lost Loggers at Google

This bug pattern was contributed by Ulf Ochsenfahrt and Eric Fellheimer at Google

- had manually tracked down a dozen or so instances, came to static analysis team
- in 30 minutes, I wrote something that found 200+ instances of this problem in Google’s code base
- Decision was made to fix all of them
Is this change compatible?

> You can argue that this change in the implementation is a bad idea
  • but it is allowed by the spec

> Perhaps if a change is made to a logger, the LogManager should store a strong reference to the logger
  • a quality of service improvement, even if spec not changed
Listen to your bug stories

> In Joshua Bloch’s talk, he said that his #1 takeaway message was don’t lock on `ConcurrentMaps`

• My reaction was “Really?”

• Clearly wrong and a bug, but surely that so obviously wrong it would be exceptionally rare

• But I wrote a detector for FindBugs
JBoss 5.1.0-GA

> 22 synchronizations on ConcurrentHashMap
> 9 synchronizations on CopyOnWriteArrayList
  • In Java 5, **COWAL** implementation using synchronized(this)
  • in Java 6+ **COWAL** implementation synchronizes on internal **Lock** object
> 3 synchronizations on AtomicBoolean
Improving software quality

> Many different things can catch mistakes and/or improve software quality

  • Each technique more efficient at finding some mistakes than others
  • Each subject to diminishing returns
  • No magic bullet
  • Find the right combination for you and for the mistakes that matter to you
Test, test, test...

Many times FindBugs will identify bugs
  - that leave you thinking “Did anyone test this code?”
    - And you find other mistakes in the same vicinity
  - FindBugs might be more useful as an untested code detector than as a bug detector

Overall, testing is far more valuable than static analysis
  - I’m agnostic on unit tests vs. system tests
  - But no one writes code so good you don’t need to check that it does the right thing
    - I’ve learned this from personal painful experience
Dead code

> Many projects contain lots of dead code
  • abandoned packages and classes
  • classes that implement 12 methods; only 3 are used

> Code coverage is a very useful tool
  • but pushing to very high code coverage may not be worthwhile
  • you’d have to cover lots of code that never gets executed in production
Code coverage from production

> If you can sample code coverage from production, great
  • look for code executed in production but not covered in unit or system test

> Note: enforce coding standard that body of if statement must be on separate line than if statement guard
  • Most statement level code coverage tools need this to tell you whether body of if statement executed
Cool idea

> If you can’t get code coverage from production
> Just get list of loaded classes
  • just your code, ignoring classes loaded from core classes or libraries
  • Very light weight instrumentation
> Log the data
  • could then ask queries such as “Which web services loaded the FooBar class this month?”
Leveraging class initialization logging

> You’ve got class initialization logging
> But want to know if a particular method or statement is reached
> Define a nested class with a static method with an empty body

```java
static class Foo {
    static void loadClass() {
    }
}
```
Using FindBugs to find mistakes

- FindBugs is accurate at finding coding mistakes
  - 75+% evaluated as a mistake that should be fixed
- But many mistakes have low costs
  - memory/type safety lowers cost of mistakes
  - If applied to existing production code, many expensive mistakes have already been removed
    - perhaps painfully
- Need to lower cost of using FindBugs to sell to some projects/teams
FindBugs 1.x

> First research paper published in 2004
> FindBugs 1.0 released in 2006
> 850,000+ downloads from 160+ countries
> Released 1.3.8 in March
FindBugs 2.0

> FindBugs analysis engine continues to improve, but only incrementally

> Focus on efficiently incorporating static analysis into the large scale software development

• Review of issues done by a community
• Once issue is marked as “not a bug”, never forget
• Integration into bug tracking and source code version control systems
Bug ranking

> FindBugs reported a priority for an issue, but it was only meaningful when comparing instances of the same bug pattern
  • a medium priority X bug might be more important than a high priority Y bug

> Now each issue receives a bug rank (a score, 1-20)
  • Can be customized according to your priorities
  • Grouped into Scariest, Scary, Troubling, and Of Concern
FindBugs community review

> Whenever / where ever you run FindBugs, after completing or loading an analysis
  • it talks to the cloud
  • sees how we’ve been seeing this issue
  • sees if anyone has marked the issue as “should fix” or “not a bug”

> As soon you classify an issue or enter text about the issue, that is sent to the cloud
More cloud integration

Integration with bug tracking systems
- One click to bring up pre-populated web page in bug tracker describing issue
- If bug already filed against issue, click shows you existing issue in bug tracker

Integration with web based source viewers, such as FishEye
- Allow viewing of file history, change lists, etc.
General availability Fall 2009

> Already in use at Google
  - need to also provide hooks into other bug tracking and web source viewers

> Cloud storage needs to be made more robust and scalable

> Needs to be integrated into Eclipse plugin

> Need to replace bubble gum and duct tape with something more stable
FindBugs community review

> Go to http://findbugs.sourceforge.net/review
> Launch FindBugs GUI via webstart
> Review issues in
  • jdk1.7.0
  • Glassfish-v3
  • Eclipse 3.5
> Everyone welcome
  • very much a beta
  • no integration with bug tracking systems yet
Demo
William Pugh
pugh@cs.umd.edu
http://findbugs.sourceforge.net/