

Improving Software Quality with Static Analysis and Annotations for Software Defect Detection


William Pugh

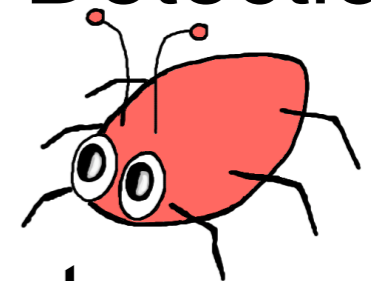
Professor, Univ. of Maryland

<http://www.cs.umd.edu/~pugh>



About Me

- Professor at Univ. of Maryland since 1988, doing research in programming languages, algorithms, software engineering
- Technical Lead on JSR-133 (Memory model), JSR-305 (Annotations for Software Defect Detection)
- Founder of the FindBugs™ project
 - Open source static analysis tool for defect detection in the Java™ Programming Language
- Technical advisory board of 



Static Analysis

- Analyzes your program without executing it
- Doesn't depend on having good test cases
 - or even any test cases
- Generally, doesn't know what your software is supposed to do
 - Looks for violations of reasonable programming
 - Shouldn't throw NPE
 - Shouldn't allow SQL injection
- Not a replacement for testing
 - Very good at finding problems on untested paths
 - But many defects can't be found with static analysis

Common Wisdom about Bugs and Static Analysis

- 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 finding them must require sophisticated static analysis techniques
 - I tried lint and it sucked: lots of warnings, few real issues

Can You Find The Bug?

Can You Find The Bug?

```
if (listeners == null)
    listeners.remove(listener);
```

- JDK1.6.0, b105, sun.awt.x11.XMSelection
 - lines 243-244

Can You Find The Bug?

```
if (listeners == null)
    listeners.remove(listener);
```

- JDK1.6.0, b105, sun.awt.x11.XMSelection
 - lines 243-244

Can You Find The Bug?

```
if (listeners == null)
    listeners.remove(listener);
```

- JDK1.6.0, b105, sun.awt.x11.XMSelection
 - lines 243-244

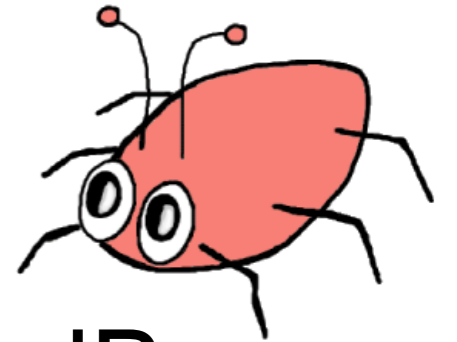
Why Do Bugs Occur?

- 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?

Who Uses Static Analysis?

- Lots and lots of projects and companies
- Among many others, Glassfish and Google use FindBugs
 - Many companies are weird about letting you say they use your open source tool
- Lots of open source tools: PMD, CheckStyle, etc.
- IDEs include some: Eclipse, IntelliJ, Netbeans
- Commercial tools available from Fortify Software, KlocWork, Coverity, Parasoft, SureLogic
- Static analysis used even more widely/intensely for C/C++
 - More bugs to find
 - Bugs a lot scarier
 - Free tools not as good

FindBugs



- I'm mostly going to be talking about FindBugs
 - I know it best
- Some things will be specific to FindBugs
 - What we classify as a "correctness" issue
 - Which potential null pointer issues we report
- But most of the concepts apply to other tools

Bug Categories

Selected categories for today's discussion

- Correctness - the code seems to be clearly doing something the developer did not intend
- Bad practice - the code violates good practice

Bug Patterns

- Some big, broad and common patterns
 - Dereferencing a null pointer
 - An impossible checked cast
 - Methods whose return value should not be ignored
- Lots of small, specific bug patterns, that together find lots of bugs
 - Every Programming Puzzler
 - Every chapter in *Effective Java*
 - Many postings to <http://thedailywtf.com/>

Analysis Techniques

Whatever you need to find the bugs

- Local pattern matching
 - If you invoke `String.toLowerCase()`, don't ignore the return value
- Intraprocedural dataflow analysis
 - Null pointer, type cast errors
- Interprocedural method summaries
 - This method always dereferences its parameter
- Context sensitive interprocedural analysis
 - Interprocedural flow of untrusted data
 - SQL injection, cross site scripting

Categories, ranking, use cases

- Every tool has categories, rules/patterns, priorities
- You can generally customize what you want to look at
- Sometimes, you want to do a code audit of a newly written module with 1,000 lines of code
 - and sometimes you want to scan 1,000,000 lines of code that has been in production for a year
- Different use cases require different tunings, different tools

Correctness issues

Stuff you *really* want to look at

- In FindBugs, we reserve the Correctness category for issues we are most confident are wrong
 - code does something the developer didn't intend
- Many of the other categories reflect correctness issues
- But correctness issues are the things we think you should look at when scanning that million line code base
- low false positive rate, few low impact bugs

... Students are good bug generators

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

```
/** 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

Infinite recursive loop

... Students are good bug generators

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

```
/** 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 JDK1.6.0-b13

- Found 5 infinite recursive loops
- Including one written by Joshua Bloch

```
public String foundType() {  
    return this.foundType();  
}
```
- Smart people make dumb mistakes
 - 27 across all versions of JDK, 40+ in Google's Java code
- Embrace and fix your dumb mistakes

Finding Null Pointer Bugs with FindBuags

- FindBuags looks for a statement or branch that, if executed, guarantees a null pointer exception
- Either a null pointer exception could be thrown, or the program contains a statement/branch that can't be executed
- Could look for exceptions that only occur on a path
 - e.g., if the condition on line 29 is true and the condition on line 38 is false, then a NPE will be thrown
 - but would need to worry about whether that path is feasible

Null Pointer Bugs Found by FindBuqs

JDK1.6.0-b105

- 109 statements/branches that, if executed, guarantee NPE
 - We judge at least 54 of them to be serious bugs that could generate a NPE on valid input
- Most of the others were deemed to be unreachable branches or statements, or reachable only with erroneous input
 - Only one case where the analysis was wrong

Examples of null pointer bugs

simple ones

```
//com.sun.corba.se.impl.naming.cosnaming.NamingContextImpl  
if (name != null || name.length > 0)
```

```
//com.sun.xml.internal.ws.wSDL.parser.RuntimeWSDLParser  
if (part == null | part.equals(""))
```

```
// sun.awt.x11.ScrollPanePeer  
if (g != null)  
    paintScrollBars(g, colors);  
g.dispose();
```

Redundant Check For Null

Also known as a reverse null dereference error

- Checking a value to see if it is null
 - When it can't possibly be null

```
// java.awt.image.LoopupOp, lines 236-247
```

```
public final WritableRaster filter(  
    Raster src, WritableRaster dst) {  
    int dstLength = dst.getNumBands();  
    // Create a new destination Raster,  
    // if needed  
    if (dst == null)  
        dst = createCompatibleDestRaster(src);  
}
```

Redundant Check For Null

Is it a bug or a redundant check?

- Check the JavaDoc for the method
- Performs a lookup operation on a **Raster**.
 - If the destination **Raster** is **null**,
 - a new **Raster** will be created.
- Is this case, a bug
 - particularly look for those cases where we know it can't be null because there would have been a NPE if it were null

Bad Method Invocation

- Methods whose return value shouldn't be ignored
 - Strings are immutable, so functions like `trim()` and `toLowerCase()` return new String
- Dumb/useless methods
 - Invoking `toString` or `equals` on an array
- Lots of specific rules about particular API methods
 - Hard to memorize, easy to get wrong

Examples of bad method calls

```
// com.sun.rowset.CachedRowSetImpl
if (type == Types.DECIMAL || type == Types.NUMERIC)
    ((java.math.BigDecimal)x).setScale(scale);
```

```
// com.sun.xml.internal.txw2.output.XMLWriter
try { ... }
catch (IOException e) {
    new SAXException("Server side Exception:" + e);
}
```

Type Analysis

- Impossible checked casts
- Useless calls
 - `equals` takes an `Object` as a parameter
 - but comparing a `String` to `StringBuffer` with `equals(...)` is pointless, and almost certainly not what was intended
 - `Map<K, V>.get` also takes an `Object` as a parameter
 - supplying an object with the wrong type as a parameter to `get` doesn't generate a compile time error
 - just a `get` that always returns null

Lots of Little Bug Patterns

- checking if `d == Double.NaN`
- Bit shifting an `int` by a value greater than 31 bits
- Every Puzzler this year
 - more than half for most years

When Bad Code Isn't A Bug

- Static analysis tools will sometimes find ugly, nasty code
 - that can't cause your application to misbehave
- Cleaning this up is a good thing
 - makes the code easier to understand and maintain
- But for ugly code already in production
 - sometimes you just don't want to touch it
- We've found more cases like this than we expected

When Bad Code Isn't A Bug

bad code that does what it was intended to do

```
// com.sun.jndi.dns.DnsName, lines 345-347
if (n instanceof CompositeName) {
    // force ClassCastException
    n = (DnsName) n;
}
```

```
// sun.jdbc.odbc.JdbcOdbcObject, lines 85-91
if ((b[offset] < 32) || (b[offset] > 128)) {
    asciiLine += ".";
}
```

When Bad Code Isn't A Bug

Code that shouldn't go wrong

```
// com.sun.corba.se.impl.dynamicany.DynAnyComplexImpl
String expectedMemberName = null;
try {
    expectedMemberName
        = expectedTypeCode.member_name(i);
} catch (BadKind badKind) { // impossible
} catch (Bounds bounds) { // impossible
}
if ( ! (expectedMemberName.equals(memberName) ... ) )
{
```

When Bad Code Isn't A Bug

When you are already doomed

```
// com.sun.org.apache.xml.internal.security.encryption.XMLCipher  
// lines 2224-2228
```

```
if (null == element) {  
    //complain  
}
```

```
String algorithm = element.getAttributeNS(...);
```


Overall Correctness Results From FindBugs

*Evaluating Static Analysis Defect Warnings On Production Software, ACM
2007 Workshop on Program Analysis for Software Tools and Engineering*

- JDK1.6.0-b105
 - 379 correctness warnings
 - we judge that at least 213 of these are serious issues that should be fixed
- Google's Java codebase
 - over a 6 month period, using various versions of FindBugs
 - 1,127 warnings
 - 807 filed as bugs
 - 518 fixed in code

Results on openjpa nightly build

- 39 correctness issues
- 35 of which I marked as must/should fix
- 2 doomed casts
- 1 uninitialized reads
- 6 null pointer issues
- 26 comparing Boolean objects using ==
 - feel free to argue with me on this one

Bad Practice

- A class that defines an equals method but inherits hashCode from Object
 - Violates contract that any two equal objects have the same hash code
- equals method doesn't handle null argument
- Serializable class without a serialVersionUID
- Exception caught and ignored
- Broken out from the correctness category

Fixing hashCode

- What if you want to define equals, but don't think your objects will ever get put into a HashMap?
- Suggestion:

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

Use of Unhashable Classes

- FindBugs previously reported all classes that defined equals but not hashCode as a correctness problem
 - but some developers didn't care
- Now reported as bad practice
 - but separately report use of such a class in a HashMap/HashTable as a correctness warning

Integrating Static Analysis

- Want to make it part of your development process
 - Just like running unit tests
- Have to tune the tool to report what you are interested in
 - Different situations have different needs
- Need a workflow for issues
 - Almost all tools will report some issues that, after reviewing, you decide not to fix
 - Need to have a way to manage such issues

Running Static Analysis

- "We've got it in our IDE, so we're done, right?"
 - no, it really needs to also be done automatically as part of your build process
- Are you scanning 2 million lines of code?
 - You probably don't want 20,000 issues to examine

Defect/Issue Workflow

- How do issues get reviewed/audited?
- Can you do team auditing and assign issues?
- Once you've reviewed an issue, does the system remember your evaluation when it analyzes that code again?
 - even if it is now reported on a different line number?
- Can you identify new issues
 - since last build?
 - since last release to customer/production?

Learning from mistakes

- With FindBugs, we've always started from bugs
- We need API experts to feed us API-specific bugs
 - Swing, EJB, J2ME, localization, Hibernate, ...
- When you get bit by a bug
 - writing a test case is good
 - considering whether it can be generalized into a bug pattern is better
 - You'd be surprised at the number of times you make a mistake so stupid “no one else could possibly make the same mistake”
 - but they do

JSR-305: Annotations for Software Defect Detection

William Pugh

Professor

Univ. of Maryland

pugh@cs.umd.edu

<http://www.cs.umd.edu/~pugh/>

Status Report



Why annotations?

- Static analysis can do a lot
 - can even analyze interprocedural paths
- Why do we need annotations?
 - they express design decisions that may be implicit, or described in documentation, but not easily available to tools

Where is the bug?

```
class Foo {
    int value;
    public boolean equals(Object obj) {
        if (this.getClass()
            != obj.getClass())
            return false;
        Foo that = (Foo) obj;
        return this.value == that.value;
    }
    ...
}
```

Where is the bug?

```
if (spec != null) fFragments.add(spec);
```

```
if (isComplete(spec)) fPreferences.add(spec);
```

Where is the bug?

```
if (spec != null) fFragments.add(spec);
```

```
if (isComplete(spec)) fPreferences.add(spec);
```

```
boolean isComplete(AnnotationPreference spec) {  
    return spec.getColorPreferenceKey() != null  
        && spec.getColorPreferenceValue() != null  
        && spec.getTextPreferenceKey() != null  
        && spec.getOverviewRulerPreferenceKey() != null;  
}
```

Finding the bug

- Many bugs can only be identified, or only localized, if you know something about what the code is supposed to do
- Annotations are well suited to this...

JSR-305

- At least two tools already have defined their own annotations:
 - FindBugs and IntelliJ
- No one wants to have to apply two sets of annotations to their code
 - come up with a common set of annotations that can be understood by multiple tools

JSR-305 target

- JSR-305 is intended to be compatible with JSE 5.0+ Java
- Hope to have usable drafts and preliminary tool support out by the end of the summer

JSR-308

- Annotations on Java Types
- Designed to allow annotations to occur in many more places than they can occur now
 - `ArrayList<@Nonnull String> a = ...`
- Targets JSE 7.0
- Will add value to JSR-305, but JSR-305 cannot depend upon JSR-308

Nullness

- Nullness is a great motivating example
- Most method parameters are expected to always be nonnull
 - some research papers support this
- Not always documented in JavaDoc

Documenting nullness

- Want to document parameters, return values, fields that should always be nonnull
- And which should not be presumed nonnull
 - argument to equals(Object)
- Should warn if null passed where nonnull value required
- Should warn if argument to equals is immediately dereferenced

Only two cases?

- What about `Map.get(...)`?
- Return nulls if key not found
 - even if all values in Map are nonnull
- So the return value can't be `@Nonnull`
- But lots of places where you “know” that the value will be nonnull
 - you know key is in table
 - you know value is nonnull

more examples

- `Object.equals(Object obj)`
 - must handle a null value for obj
- `Method.invoke(Object obj, Obj.. args)`
 - obj should be null if method is a static method, nonnull if an instance method
- `ConcurrentHashMap.get(K k)`
 - k must not be null

3 cases?

- May need to have 3 cases for nullness
 - @Nonnull
 - @CheckForNull
 - @UnknownNullness
 - same as no annotation
- Names in flux, might use Nullable for one of these (but which one?)

@Nonnull

- Should not be null
 - For fields, should be interpreted as should be nonnull after object is initialized
- Tools will try to generate a warning if they see a possibly null value being used where a nonnull value is required
 - same as if they see a dereference of a possibly null value

@CheckForNull

- Code should always worry that this value might be null
 - e.g., argument to equals

@UnknownNullness

- Same as no annotation
 - Needed because we are going to introduce default and inherited annotations
 - Need to be able to get back to unannotated state
- Null under some circumstances
 - might vary in subtypes

@CheckForNull requires work

- If you mark a return value as @CheckForNull, you will likely have to go make a bunch of changes
 - kind of like const in C++
- My experience has been that there are lots of methods that could return null
 - but that in a particular calling context, you might know that it can't

Nullness annotations

- Tools that try to prove absence of NPE might treat `@CheckForNull` and `@UnknownNullness` the same
- This 3-way logic might reappear for some other annotations

Type Qualifiers

- Many of the JSR-305 annotations will be type qualifiers: additional type constraints on top of the existing Java type system

@Untainted / @Tainted

- Needed for security analysis
- Information derived directly from web form parameters is tainted
 - can be arbitrary content
- Strings used to form SQL queries or HTML responses must be untainted
 - otherwise get SQL Injection or XSS

@Syntax

- Used to indicate String values with particular syntaxes
 - @Syntax("Regex")
 - @Syntax("Java")
 - @Syntax("SQL")
- Allows for error checking and used by IDE's in refactoring

@Pattern

- Provides a regular expression that describes the legal String values
 - @Pattern("\\d+")

@Nonnegative and friends

- Fairly clear motivation for @Nonnegative
- More?
 - @Positive
- Where do we stop?
 - @NonZero
 - @PowerOfTwo
 - @Prime

Three-way logic again

- If we have @Nonnegative, do we also need:
 - @Signed
 - similar to check for null
 - @UnknownSign
 - similar to unknown nullness

User defined type qualifiers

- In (too many) places, Java APIs use integer values or Strings where enumerations would have been better
 - except that they weren't around at the time
- Lots of potential errors, uncaught by compiler

Example in java.sql.Connection

`createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)`

Creates a Statement object that will generate ResultSet objects with the given type, concurrency, and holdability.

`resultSetType`: one of the following ResultSet constants:

`ResultSet.TYPE_FORWARD_ONLY`,

`ResultSet.TYPE_SCROLL_INSENSITIVE`, or

`ResultSet.TYPE_SCROLL_SENSITIVE`

`resultSetConcurrency`: one of the following ResultSet constants:

`ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

`resultSetHoldability`: one of the following ResultSet constants:

`ResultSet.HOLD_CURSORS_OVER_COMMIT` or

The fix

- Declare
 - `public @TypeQualifier @interface ResultSetType {`
 - `public @TypeQualifier @interface ResultSetConcurrency {`
 - `public @TypeQualifier @interface ResultSetHoldability {`
- Annotate static constants and method

User defined Type Qualifiers

- JSR-305 won't define @ResultSetType
- Rather JSR-305 will define the meta-annotations
 - that allow any developer to define their own type qualifier annotations
 - which they can apply and will be interpreted by defect detection tools

Other ideas

- Validators
- Subtypes
- exclusive type qualifiers (e.g., it can't be both A and B)

CreditCard example

```
@Documented @TypeQualifier @Retention(RetentionPolicy.RUNTIME)
    @Pattern("[0-9]{16}")
public @interface CreditCardNumber {
    class Validator implements TypeQualifierValidator<CreditCardNumber> {
        public boolean forConstantValue(CreditCardNumber annotation, Object v) {
            if (v instanceof String) {
                String s = (String) v;
                if (java.util.regex.Pattern.matches("[0-9]{16}", s)
                    && LuhnVerification.checkNumber(s))
                    return true;
            }
            return false;
        }
    }
}
```


Default and Inherited Annotations

Most parameters are nonnull

- Most references parameters are intended to be non-null
 - many return values and fields as well
- Adding a `@Nonnull` annotation to a majority of parameters won't sell
- Treating all non-annotated parameters as nonnull also won't sell

Default annotations

- Can mark a method, class or package as having nonnull parameters by default
 - If a parameter doesn't have a nullness annotation
 - climb outwards, looking at method, class, outer class, and then package, to find a default annotation
- Can mark a package as nonnull parameters by default, and change that on a class or parameter basis as needed

Inherited Annotations

- We want to inherit annotations
 - `Object.equals(@CheckForNull Object obj)`
 - `int compareTo(@Nonnull E e)`
 - `@Nonnull Object clone()`

Do defaults apply to most JSR-305 annotations?

- Case for default and inherited nullness annotations is very compelling
- Should it be general mechanism?

Thread/Concurrency Annotations

- Annotations to denote how locks are used to guard against data races
- Annotations about which threads should invoke which methods
- See annotations from *Java Concurrency In Practice* as a starting point

What is wrong with this code?

```
Properties getProps(File file)
throws ... {
    Properties props = new Properties();
    props.load(new FileInputStream(file));
    return props;
}
```

What is wrong with this code?

```
Properties getProps (File file)
throws ... {
    Properties props = new Properties ();
    props.load(new FileInputStream(file));
    return props;
}
```

Doesn't close file

Resource Closure

- `@WillNotClose`
 - this method will not close the resource
- `@WillClose`
 - this method will close the resource
- `@WillCloseWhenClosed`
 - Usable only in constructors: constructed object decorates the parameter, and will close it when the constructed object is closed

Miscellaneous

- @CheckReturnValue
- @InjectionAnnotation

@CheckReturnValue

- Indicates a method that should always be invoked as a function, not a procedure.
- Example:
 - `String.toLowerCase()`
 - `BigInteger.add(BigInteger val)`
- Anywhere you have an immutable object and methods that might be thought of as mutating methods return the new value

@InjectionAnnotation

- Static analyzers get confused if there is a field or method that is accessed via reflection/injection, and they don't understand it
- Many frameworks have their own annotations for injection
- Using @InjectionAnnotation on an annotation @X tells static analysis tools that @X denotes an injection annotation

Wrap up

- Static analysis is effective at finding bad code
 - Is bad code found by static analysis an important problem?
- Getting static analysis into the software development process can't be taken for granted
- Annotations will be helpful
 - If we can get developers to use them