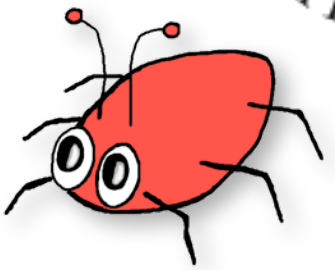
Making Static Analysis Part Of Your Build Process

William Pugh

Professor, Univ. of Maryland
Visiting Scientist, Google



Learn how to effectively use FindBugs on large software projects (100,000+ lines of code), and make effective use of the limited time you can schedule/afford for static analysis

Agenda

- **FindBugs and static analysis**
- Using FindBugs effectively
- Running FindBugs
- Scaling up FindBugs
- Historical Bug results

Static Analysis

- Analyzes your program without executing it
- Doesn’t depend on having good test cases
  - or even any test cases
- Doesn’t know what your software is supposed to do
  - Looks for violations of reasonable programming practices
    - Shouldn’t throw NPE
    - All statements should be reachable
    - 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 (Incorrect) 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
- I tried lint and it sucked: lots of warnings, few real issues
- So, bugs remaining in production code must be subtle, and finding them must require sophisticated static analysis techniques

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?

You may not need this talk

- If you just want to run FindBugs over a few thousand lines of code
  - just do it
- We won't be showing examples of the many problems FindBugs can find
  - Talk assumes some familiarity with static analysis tools like FindBugs
- This talk is focused on the problems involved in trying to apply FindBugs, or any static analysis tool, to a project with 100,000+ lines of code
  - useful for smaller code base, but not essential

FindBugs does scale

- Both Google and eBay have put substantial effort into tuning FindBugs for their environment and building it into their standard software development process
- Google has fixed more than 1,000 issues identified by FindBugs.
  - I can't tell you how large their code base is, but it is big
- But even at Google, scaling up static analysis is a challenge

Agenda

- FindBugs and static analysis
- **Using FindBugs effectively**
- Running FindBugs
- Scaling up FindBugs
- Historical Bug results

No silver bullets

- Static analysis isn’t a silver bullet
  - won’t ensure your code is correct or of high quality
- Other techniques are just as valuable, if not more so
  - careful design
  - testing
  - code review

Finding the right combination

- Everything you might do to improve software quality
  - is very effective at finding some kinds of problems
  - is subject to diminishing returns
- You have a finite and fixed time budget
  - spending time on static analysis means less time on something else
- Want to find an effective/profitable way to use static analysis to improve software quality

This talk

- Understanding the FindBugs ecosystem
- Customizing FindBugs to your needs
- Adapting FindBugs to your time budget
  - Find your sweet spot
- Making FindBugs part of your continuous build and test framework

- Only enough time to tell you what approaches and strategies help
  - not enough time to walk you through using them

Running the analysis and finding obviously stupid code is easy

- Need to budget time for more than just running the analysis and reviewing the bugs
- Often, the hard part is stuff like:
  - Figuring out who is responsible for that code
  - Understanding what the code is actually supposed to do
  - Figuring out if stupid code can cause the application to misbehave
  - Writing a test case that demonstrates the bug
  - Getting approval to change the code

Agenda

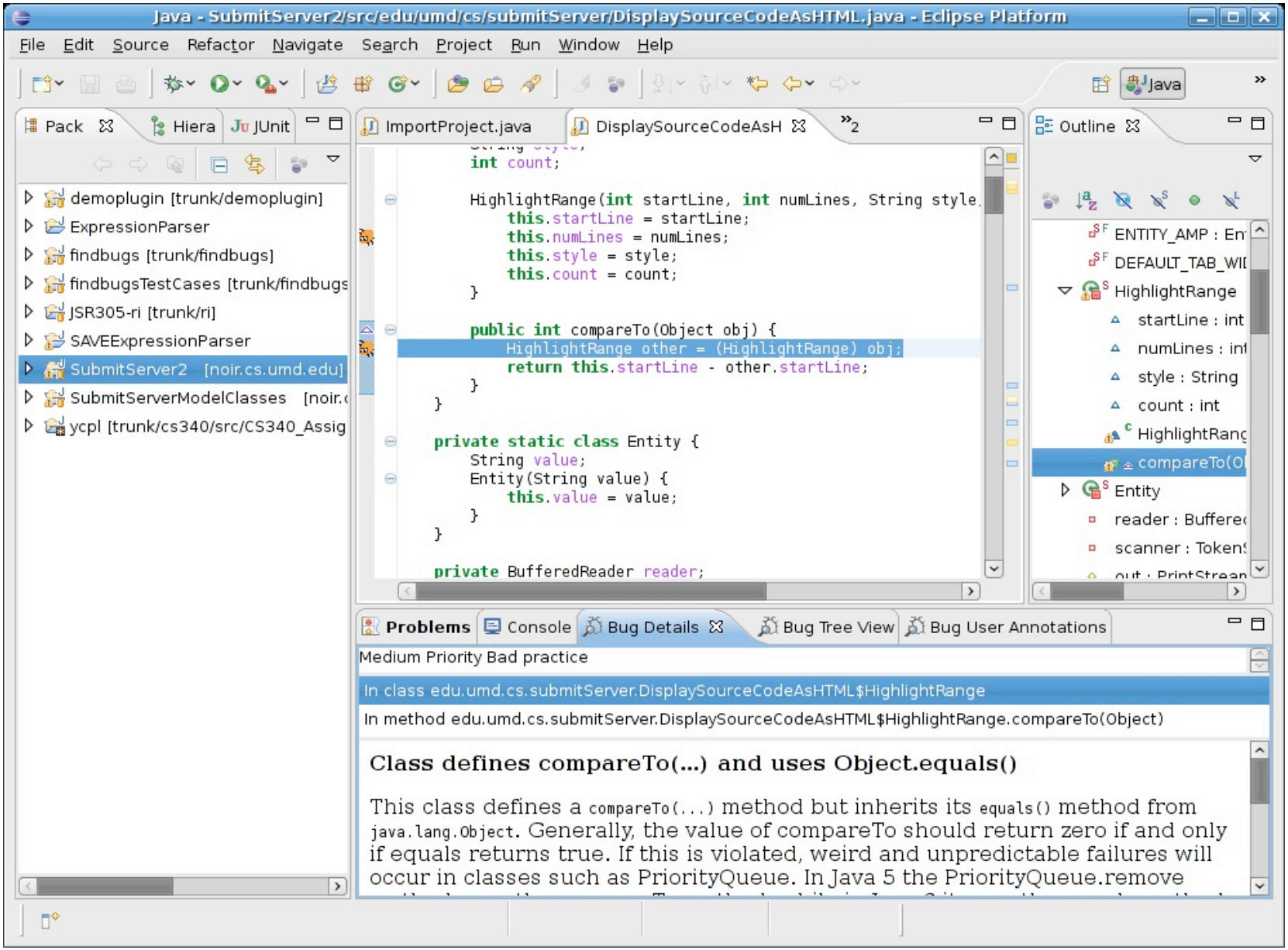
- FindBugs and static analysis
- Using FindBugs effectively
- **Running FindBugs**
- Scaling up FindBugs
- Historical Bug results

FindBugs ecosystem

- FindBugs analyzes classfiles
  - sourcefiles used only for display
  - can analyze jsp from resulting classfiles, but defects mapped to Java™ source files (no SMAP parsing yet)
- Filter files can be used to include or exclude certain issues
- Output stored in XML format
- Many tools for post-processing XML result

Ways of performing analysis

- Supported by FindBugs project:
  - Swing GUI
  - Command line
  - Eclipse IDE
  - Ant
- Others
  - NetBeans™ IDE - SQE suite
  - Maven
  - Cruise Control
  - Hudson


Java - SubmitServer2/src/edu/umd/cs/submitServer/DisplaySourceCodeAsHTML.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Pack
Hiera
JUnit
demoplugin [trunk/demoplugin]
ExpressionParser
findbugs [trunk/findbugs]
findbugsTestCases [trunk/findbugs
JSR305-ri [trunk/ri]
SAVEExpressionParser
SubmitServer2 [noir.cs.umd.edu]
SubmitServerModelClasses [noir.
ycpl [trunk/cs340/src/CS340_Assig
ImportProject.java
DisplaySourceCodeAsH
int count;
HighlightRange(int startLine, int numLines, String style.
this.startLine = startLine;
this.numLines = numLines;
this.style = style;
this.count = count;
public int compareTo(Object obj) {
HighlightRange other = (HighlightRange) obj;
return this.startLine - other.startLine;
private static class Entity {
String value;
Entity(String value) {
this.value = value;
private BufferedReader reader;
Outline
ENTITY_AMP : En
DEFAULT_TAB_WI
HighlightRange
startLine : int
numLines : int
style : String
count : int
HighlightRanc
compareTo(O
Entity
reader : Buffere
scanner : Token
Problems
Console
Bug Details
Bug Tree View
Bug User Annotations
Medium Priority Bad practice
In class edu.umd.cs.submitServer.DisplaySourceCodeAsHTML$HighlightRange
In method edu.umd.cs.submitServer.DisplaySourceCodeAsHTML$HighlightRange.compareTo(Object)
Class defines compareTo(...) and uses Object.equals()
This class defines a compareTo(...) method but inherits its equals() method from java.lang.Object. Generally, the value of compareTo should return zero if and only if equals returns true. If this is violated, weird and unpredictable failures will occur in classes such as PriorityQueue. In Java 5 the PriorityQueue.remove

Command line

- In the beginning was the command line...

```
findbugs -textui -project myProj.fbp \
   -xml -outputFile myProj-analysis.fba
```

- or

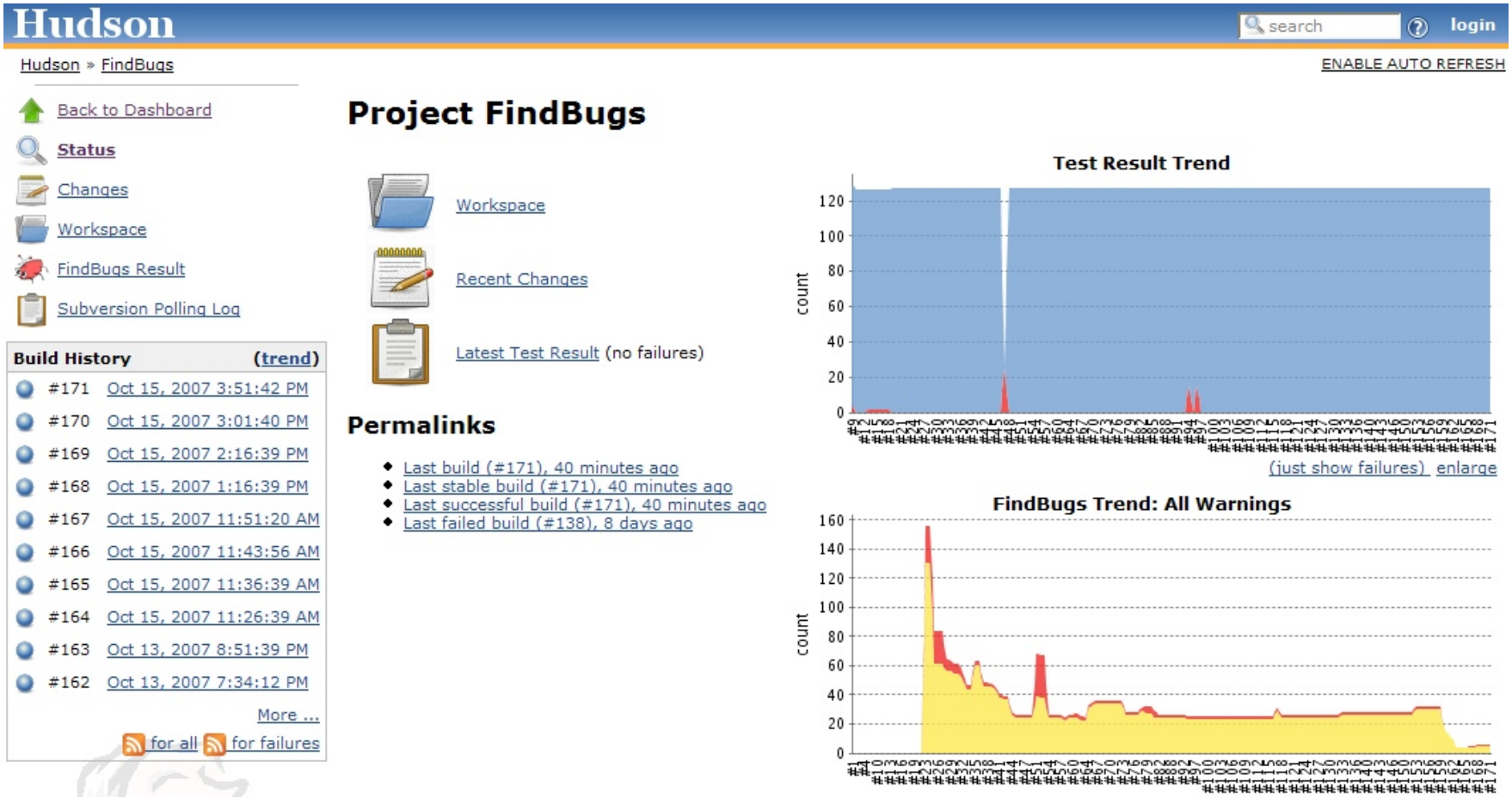
```
findbugs -textui -project myProj.fbp \
   -xml:withMessages -outputFile myProj-analysis.fba
```

- Using -xml:withMessages writes human-readable message strings in the XML output
  - Useful if any tool other than FindBugs will use the output

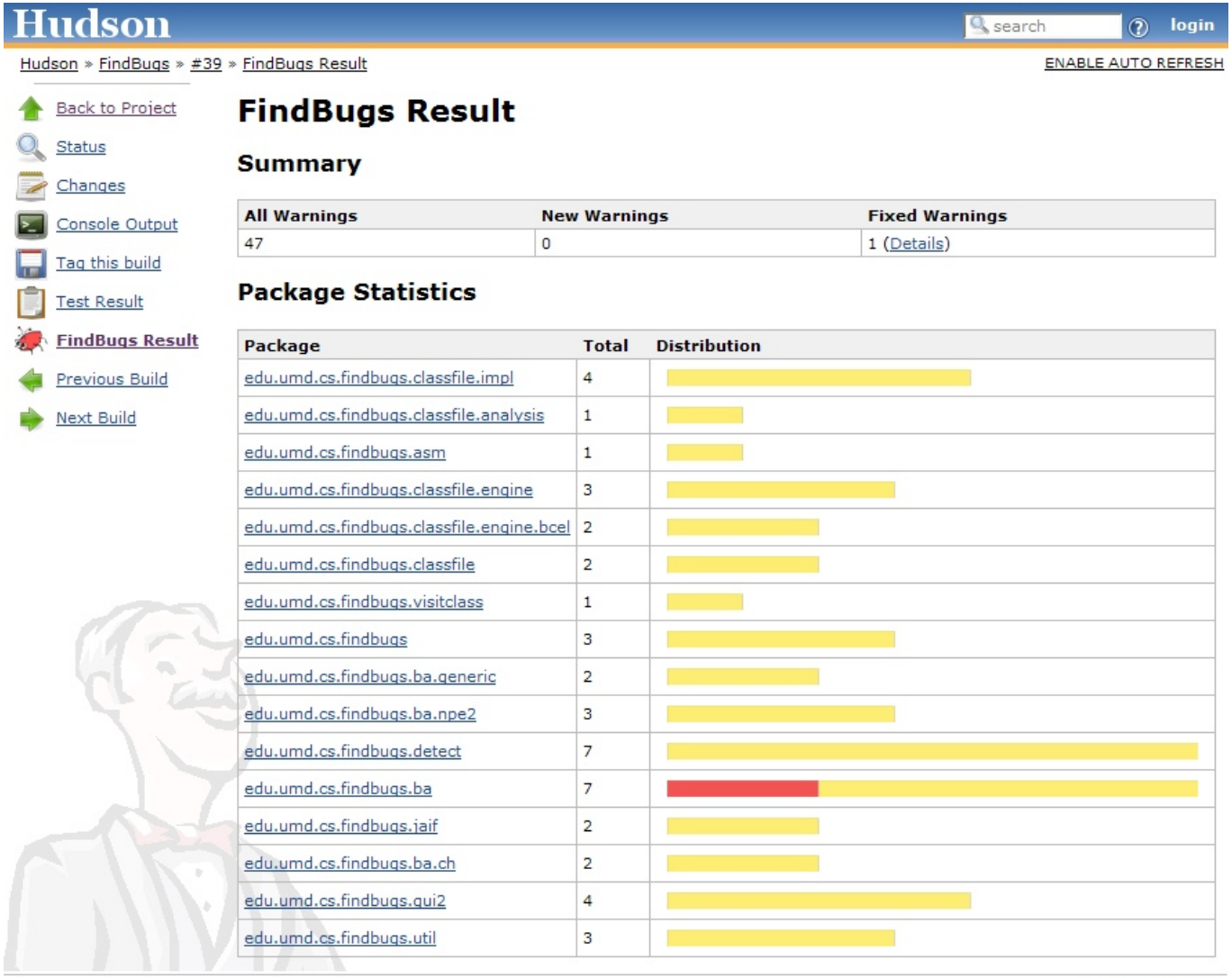
Plugin for Hudson

- Reads FindBugs xml output for each build
- Presents:
  - Warning trend graph
  - Warning deltas for each build
  - Per-package warning bar graphs
  - Links into source code
- Warnings may optionally affect project “health”
- Plugin by Ullrich Hafner
- Hudson by Kohsuke Kawaguchi

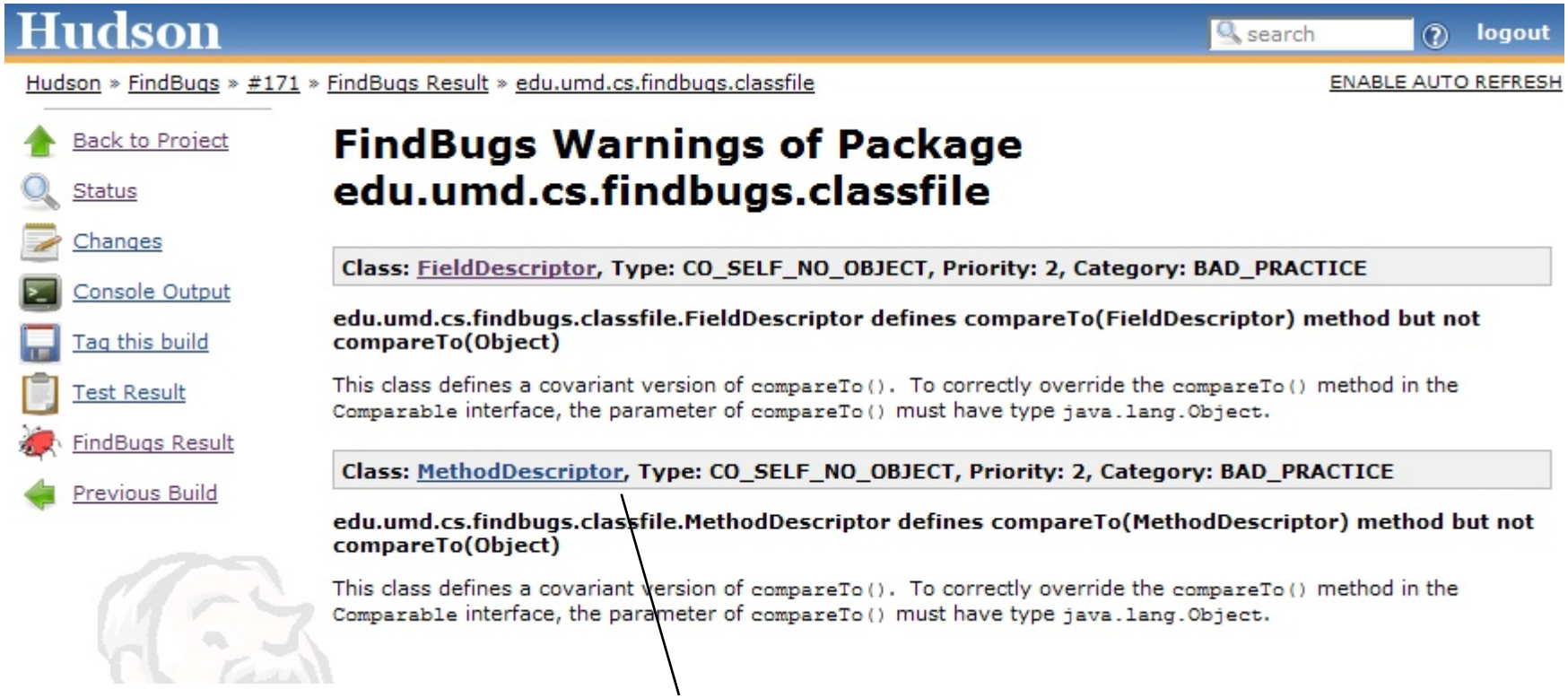
Plugin for Hudson



Plugin for Hudson



Plugin for Hudson



Link into source

Maven

- We've let this slip
  - documentation isn't good
- Want to use version 2.0 of Maven FindBugs plugin
  - requires Maven 2.0.8+
- Use mvn findbugs:check or mvn findbugs:findbugs

```
<plugins>
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>findbugs-maven-plugin</artifactId>
    <version>2.0</version>
    <configuration>
      <xmlOutput>true</xmlOutput>
      <xmlOutputDirectory>out</xmlOutputDirectory>
      <findbugsXmlOutput>true</findbugsXmlOutput>
      <findbugsXmlOutputDirectory>out</findbugsXmlOutputDirectory>
    </configuration>
  </plugin>
</plugins>
```

Agenda

- FindBugs and static analysis
- Using FindBugs effectively
- Running FindBugs
- **Scaling up FindBugs**
  - Workload
  - What issues are you interested in?
  - filter files
- Historical Bug results

OK, now what...

- You've gotten FindBugs installed
- You've run it over your code, found a few issues you wanted to fix immediately
  - some other issues look scary, but don't cry out for immediate action
  - other issues are harmless (even if dumb)

Make it manageable

- FindBugs reported 36,062 issues on Eclipse 3.4M2
  - Can’t cope…
- Filter out low priority issues… 25,952 issues
- Filter out vulnerability to malicious code… 5,172 issues
- Filter out issues also present in Eclipse 3.3… 62 issues
  - uses approximate matching, ignoring line numbers

Remembering evaluations

- If you evaluate an issue but don't immediately fix the code, want to remember your evaluation
  - issues that must be addressed/fixed/reviewed before the next release
  - issues that are harmless and you don't want to review again
  - probably some cases in between those two extremes

Highlight new issues

- If you are running FindBugs as part of a daily or continuous build or integration environment
  - You want to flag any new issues
- Just keeping track of trend lines of total number of issues isn’t good enough
- If a change introduces an issue, you want to call out the issue

- The Hudson build server does this fairly well
  - like to make it even better

Integrate it

- You want to integrate it into your bug reporting and tracking system
  - scrape the XML and import data into your database
  - link FindBugs warning and bug database entry
    - be able to go from one to the other
    - check if issues flagged as MUST_FIX in database have been fixed in the code

Agenda

- FindBugs and static analysis
- Using FindBugs effectively
- Running FindBugs
- Scaling up FindBugs
  - **Workload**
  - What issues are you interested in?
  - Filter files
- Historical Bug results

Typical FindBugs warning density

- About 0.3 - 0.6 medium or high priority correctness warnings per 1,000 lines of NCSS (Non commenting source statements)
- About 1-4 other potentially relevant warnings per 1,000 lines of code
- Don’t use these numbers to judge whether your project is good or bad
  - Lots of reasons results might be biased
  - Rather, use them to do back of the envelope calculation of how many issues you’d need to process

At Google

- Over two years, perhaps one person year of effort on auditing issues
- Over that span, reviewed 1,663 issues
  - 804 fixed by developers
    - more since that effort
- Back of the envelope
  - 5-15 issues reviewed and processed per day per auditor

Agenda

- FindBugs and static analysis
- Using FindBugs effectively
- Running FindBugs
- Scaling up FindBugs
  - Workload
  - **What issues are you interested in?**
  - Filter files
- Historical Bug results

Priority

- Each issue is ranked as High, Medium, Low
- We generally don't recommend looking at Low priority issues on large code bases
  - lots of noise
- High/Medium are useful for ranking issues within a pattern, but not as useful across patterns/categories
  - Medium FOO issues might be more important than High BAR issues

Bug Categories

- Correctness - the code seems to be clearly doing something the developer did not intend
- Security - e.g., SQL injection, cross site scripting
- Bad practice - the code violates good practice
- Dodgy code - the code is doing something unusual that may be incorrect
- Multithreaded correctness
- Potential performance problems
- Malicious code vulnerability
- Internationalization

Categories

- Malicious code is really important if you run in the same Java Virtual Machine (JVM™) as untrusted code
  - JVM implementations should care
- Performance issues are generally only important in the 10% of your code than consumes 90% of your cycles
- Thread safety issues are only important if your code might be touched by multiple threads

Run first, then filter

- Generally, full suite of bug detectors is run, including detectors that produce issues you don’t care about
- Then suppress or exclude issues you don’t care about
- No real performance win to selectively enabling detectors
  - unless you are just testing a new detector

Simple filtering

- Some tools allow you to specify simple filters
  - For command line, specify minimum priority
  - For Eclipse, specify priority and categories
- The filter command and filter ant task have lots of options

Agenda

- FindBugs and static analysis
- Using FindBugs effectively
- Running FindBugs
- Scaling up FindBugs
  - Workload
  - What issues are you interested in?
  - **Filter files**
- Historical Bug results

Filter files allow more complicated filtering and logic

```
<FindBugsFilter>
<Match><Priority value="3"/></Match>
 <Match>
   <Class name="edu.umd.cs.findbugs.jaif.JAIFToken"/>
   <Bug pattern="URF_UNREAD_FIELD"/>
 </Match>
<Match>
    <BugCode name="Se"/>
    <Class name="~edu.umd.cs.findbugs.gui.*"/>
  </Match>
</FindBugsFilter>
```

Can include or exclude filters

- Only bugs that match the include filter and don't match the exclude filter are reported
  - rarely use both
- Used when running the analysis, filtering bugs, and in Eclipse plugin

Filter use cases

- ☒ Can use filters to describe which kinds of issues are interesting or uninteresting
- ☒ Can also filter out specific instances that have been reviewed and found to be uninteresting
  - we should offer better ways to do this, and we are working on it, but this works
    - you'll see some of the other ways shortly

Building filters in GUI

- The FindBugs GUI supports suppression filters
  - stored in the XML results
  - suppression filters aren't widely supported in the FindBugs ecosystem yet
- Click on a bug, select "Filter bugs like this..."
  - select attributes that you want to be part of the filter
  - added to filter

Exporting/Importing filters

- The GUI allows you to export/import filters
  - export the current suppression filter as a filter file
  - import a filter file and merge it into the current suppression filter
- The easiest way to create filter files
  - no need to edit xml files with a text editor

Agenda

- ☒ FindBugs and static analysis
- ☒ Using FindBugs effectively
- ☒ Running FindBugs
- ☒ Scaling up FindBugs
- ☒ **Historical Bug results**
  - Excluding baseline bugs
  - Saving audit results
  - Instance hashes

Merging analysis results

- If you run FindBugs as part of each build
- you can merge analysis results
  - **`computeBugHistory -output bugHistory.xml bugHistory.xml newAnalysis.xml`**
  - combine bugHistory.xml and newAnalysis.xml
  - save the result in bugHistory.xml

Merging analysis results

- FindBugs matches up corresponding bugs in successive versions
  - fuzzy match; line numbers aren't considered
- For a bug that persists across multiple versions, the XML records the first and last version that contained the bug
  - also records whether a bug was introduced into an existing class, or if a bug and the class that contains it were introduced at the same time

Querying historical bug databases

- You can filter bugs based on the first or last version that contained an issue, or how it was introduced or removed
  - either by parameters to filter command, or in filter files

Agenda

- ☒ FindBugs and static analysis
- ☒ Using FindBugs effectively
- ☒ Running FindBugs
- ☒ Scaling up FindBugs
- ☒ Historical Bug results
  - **Instance hashes**
  - Excluding baseline bugs
  - Saving audit results

Instance hashes

- When you generate an XML file with messages, each bug has an associated instance hash
  - a 32 character hexidecimal string formed by a MD5 hash of all the things believed to be unchanging about the issue
    - e.g., doesn't consider line number
- Useful for connecting analysis results to bug databases, other forms of external processing

Instance hash collisions

- Instance hashes are not guaranteed to be unique
  - two null pointer warnings about the variable x in the method foo in the class Bar will both generate the same hash
- Can have two issues in the same analysis with the same hash
- Can have a hash that occurs in two different analysis results that doesn't really reflect the same issue

Unique identifiers

- Each issue has a occurrenceNum and a occurrenceMax as well as a hash
- concatenating all 3 gives something unique to the file
  - and unlikely to collide across successive versions
- <BugInstance type="BIT_AND" priority="2" abbrev="BIT" category="CORRECTNESS" instanceHash="f1826ab8704305b22e35e9029e848831" instanceOccurrenceNum="0" instanceOccurrenceMax="0">

Agenda

- ☒ FindBugs and static analysis
- ☒ Using FindBugs effectively
- ☒ Running FindBugs
- ☒ Scaling up FindBugs
- ☒ Historical Bug results
  - Instance hashes
  - **Excluding baseline bugs**
  - Saving audit results

Establishing a bug baseline

- Say you want to just look at issues that have been introduced into the code since release 3.0
  - too many issues to look at all of them
  - perhaps issues that made it through the 3.0 release process are less likely to cause the software to misbehave
    - hoping testing would have found most of the misbehaviors

Excluding a baseline

- You can exclude bugs in a baseline by computing historical bug databases
  - compute a historical database
  - exclude those present in the first version
- But this can be awkward, can’t use it in Eclipse, ...

Simple bug baselines

- ☒ The filter command and the eclipse plugin allow you to specify a bug baseline
  - an XML file of analysis results for your baseline
- ☒ Any issue that also occurs in the baseline is excluded
  - based on instance hash

Agenda

- ☒ FindBugs and static analysis
- ☒ Using FindBugs effectively
- ☒ Running FindBugs
- ☒ Scaling up FindBugs
- ☒ Historical Bug results
  - Instance hashes
  - Excluding baseline bugs
  - **Saving audit results**

Annotating issues

- The Swing GUI and Eclipse plugin allows you to mark an issue as one of the following:
  - unclassified
  - needs further study
  - bad analysis
  - not a bug
  - mostly harmless
  - should fix
  - must fix

Free text annotations

- The GUI and Eclipse plugin also supports free text annotation
  - “Joe should fix this”
  - “Ask Susan about whether or not the foobar parameter is allowed to be null”

Historical matching and annotations

- When you combine bug results
- The matcher combines/transfers user designations and annotations from the old results to the new issues
- Now, when you view it, you see the new line number, but it remembers the previous designation and annotation
- GUI automatically combines old result with new result

Eclipse and user annotations

- Eclipse has alpha-level support for user annotations
  - use “Bug User Annotation Window”
- Keeps history
- But doesn’t provide any way to share it among multiple users or workspaces
  - version control based merging of XML bug databases isn’t recommended or supported

New plans, to be accomplished *soon*

- Information about “Harmless” or “Must fix” stored in central store (such as a SQL database), as well as information about which issues are new and which are old
  - This information can be accessed from all the ways FindBugs can be run (Maven, Ant, GUI, Eclipse)
- Integration with bug tracking systems: file bugs from FindBugs, link to existing entries in bug tracker
- Integration with web-based source viewing tools, such as FishEye
-

Summary

- Don’t worry about looking at all the issues reported by FindBugs
  - you probably have better things to do with your time
- Figure out which kinds of issues are most relevant/important
  - don’t look at ones that aren’t
- Recently introduced issues are more likely to be worth looking at
  - easier to find developer who understands the code and issue
  - If an issue has been in your codebase for two years and no-one has found a reason to fix it, the odds that it can actually cause problems are lower (but not zero).