Code Coverage and Static Analysis

CSC216

---

Code Coverage

- A measure of test case completeness
  - **Method coverage**: Have all methods been called?
  - **Statement coverage**: Have all statements in a method been executed?
  - **Decision/Branch Coverage**: Have all decisions been executed in both the true and false paths?
  - **Condition Coverage**: Have all conditionals been executed in both the true and false paths?
Triangle Type

- Assuming that three sides create a valid triangle, what type of triangle do we have?

```java
public String getTriangleType(int a, int b, int c) {
    if (a == b && b == c) {
        return "Equilateral";
    }
    if (a == b || b == c || c == a) {
        return "Isosceles";
    }
    return "Scalene";
}
```

Method Coverage

- All methods have been tested at least once
- Test Case?
Statement Coverage

- All statements in a method have been covered
- Test Cases:

```java
public String getTriangleType(int a, int b, int c) {
    if (a == b && b == c) {
        return "Equilateral";
    }
    if (a == b || b == c || c == a) {
        return "Isosceles";
    }
    return "Scalene";
}
```

Branch/Decision Coverage

- All conditional tests have been evaluated to true AND to false at least once
- Test Cases?

<table>
<thead>
<tr>
<th>Line</th>
<th>Conditional</th>
<th>True Test</th>
<th>False Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(a == b &amp;&amp; b == c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(a == b</td>
<td></td>
<td>b == c</td>
</tr>
</tbody>
</table>
Condition Coverage

- All parts of a compound conditional test have been evaluated to true AND to false at least once
- Test Cases?

```java
public String getTriangleType(int a, int b, int c) {
    if (a == b && b == c) {
        return "Equilateral";
    }
    if (a == b || b == c || c == a) {
        return "Isosceles";
    }
    return "Scalene";
}
```

<table>
<thead>
<tr>
<th>Predicate</th>
<th>True Test</th>
<th>False Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a == b) @ line 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b == c) @ line 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c == a) @ line 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Benefits of Code Coverage

- A measure of how complete your test cases are
  - High coverage does not guarantee code correctness!
- Can identify paths through the code that you may have missed
  - Start by testing requirements
  - Consider equivalence classes and boundary values
  - Then work from your control flow diagram and basis path testing
  - Think diabolically
Limitations of Code Coverage

• The assumption that you are done testing if you have high coverage is incorrect
  – Coverage tools only tell you if you’ve covered what’s there
  – There may be requirements that you have missed!

• Don’t write test cases ONLY to make your coverage tool happy
  – Inspect surrounding code for other potential errors, perhaps requirements you’ve missed

Code Coverage Tools

• EclEmma
  – Eclipse plug-in
  – Measures (at the bytecode level)
    • Instruction Coverage
    • Block Coverage – roughly corresponds to condition coverage
    • Line Coverage
    • Method Coverage
    • Type Coverage
  – Records coverage of automated test suites AND black box test cases
  – Run EclEmma on your code to determine method coverage BEFORE submitting to Web-CAT.
    • On both your project and the tests!
Code Coverage Tools

- **djUnit**
  - Eclipse plug-in
  - Measures
    - Statement Coverage
    - Branch Coverage
  - Doesn’t break coverage down to the method level
  - Can only run coverage on JUnit tests

- **Clover**
  - Used to calculate coverage of tests submitted to Web-CAT
    - Commercial software
  - Measures
    - **Method coverage**: counts all methods and determines if each method has been entered (called)
    - Statement coverage: counts all statements and determines if each statement has been exercised
    - Decision coverage: counts all tests and if each possible branch has been executed
Static Analysis

- "Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation" [IEEE]

- Does not involve the execution of the program
  - Otherwise, Dynamic Analysis

- Software inspections are a form of static analysis
  - May be called program understanding, program comprehension, or code review
  - We’ll cover this later in the semester

- So are software metrics
  - Size, complexity, depth of inheritance, etc.

Why Static Analysis?

- Find faults in our programs
  - Code Inspections: identify where code may not meet specifications
  - Code Metrics: identify areas that may have potential problems
  - Automated static analysis: “look for violations of reasonable or recommended programming practices”

- Hovermeyer/Pugh (authors of FindBugs):
  - “Even well tested code written by experts contains a surprising number of obvious bugs”
  - “Java has many language features and APIs which are prone to misuse.”
  - Static analysis tools “can serve an important role in raising the awareness of developers about subtle correctness issues. . . prevent future bugs”
Automated Static Analysis Tools

- Search through code to detect bug patterns
  - Error-prone coding practices that arise from the use of erroneous design patterns, misunderstanding of language semantics, or simple and common mistakes.
- Increasingly being used to identify security vulnerabilities
- “can peer into more of a program’s dark corners with less fuss than dynamic analysis”

Problems with Static Analysis Tools

- False positive: the tool reports bugs the program doesn’t contain
  - A static analysis tool will brag about having only 50% false positives.
  - Need to manually review and decide whether to fix or ignore. Some tools allow you to create filters of the types of bugs you don’t want to see.
- False negative: the program contains bugs the tool doesn’t report
  - Missing bug patterns
  - Not easily captured by any bug patterns
- May also detect “harmless bugs” which need human judgment to sort out
  - My dissertation work – automatically identify actionable alerts from developer’s past behavior
Let’s find some bugs...

//What bug am I?
public String foundType() {
    return this.foundType();
}

- FindBugs researchers have found
  - 13 infinite recursive loops in JDK 1.6
  - Google has found and fixed more than 70 infinite recursive loops

Bugs?

//What’s the potential bug?
public void someMethod(String s) {
    //code
    s.toLowerCase();
    //other code
}

//What about this?
try {
    // ... }
catch (IOException e) {
    new SAXException();
}
Null Dereferences

• What are null dereferences?
  – A dereference is accessing a value from a variable
  – If it’s null we get a NullPointerException

• Use intraprocedural dataflow analysis
  – Intraprocedural: within a method or procedure
  – Dataflow: Like control flow but annotated with information about data
  – This is an advanced undergrad/graduate level topic

Null Dereference Examples

```java
if (name != null || name.length() > 0) {
    //additional code here
}

if (g != null)
    paintScrollBars(g, colors);
    g.dispose();
    //This is why you use braces with if statements!

    //How would we find these?
```
Static Analysis in Web-CAT

• CheckStyle
  – Finds style issues with your code
    • Places where you code doesn’t meet the departmental style guidelines
  – Why good style?
    • Makes it easier for others to inspect your code and understand quickly what’s going on
  – Style checks
    • Indentation, spacing, naming conventions, etc.
  – Use configuration file
    • Eclipse CheckStyle plug-in will not provide a notification when you’re missing a Javadoc comment for an overridden method. Web-CAT will! Javadoc everything!

Static Analysis in Web-CAT (2)

• PMD
  – Finds style issues and potential bugs in your code
  – We’ll use a subset of entries that enforce style guidelines and that may find bugs (like null pointer dereferences)
  – Use configuration file
FindBugs Static Analysis for CSC216

- FindBugs finds more interesting and more serious problems with your code
  - More like the examples in the slides
  - Programming projects will require that you remove FindBugs alerts at the “Scariest” and “Scary” warning levels
  - Testing tutorial describes how to configure FindBugs to show the warnings that you must remove as compiler errors
  - 5 points on Part 2 of projects for not having any “Scariest” and “Scary” FB warnings!

References

- Hovemeyer, David and Pugh, William, “Finding Bugs is Easy”, OOPSLA 2004