Testing Methods: White Box Testing I

Outline

- Today we begin to look at white box testing
- We’ll look at:
  - white box vs black box
  - role and kinds of white box testing
  - implementation - source, executable and sampling
  - code coverage testing
    - statement coverage
    - basic block coverage
White Box vs. Black Box

Recall: **Kinds of Tests**

- We divide tests into:
  - **Black box** methods – cannot see the software code (it may not exist yet!) – can only base their tests on the requirements or specifications
  - **White box** (aka glass box) methods – can see the software’s code – can base their tests on the software’s actual architecture or code
Kinds of White Box Testing

**Code Coverage**
- Code coverage methods design tests to cover (execute) every method, statement or instruction of the program at least once

**Logic Path/Decision Point Coverage**
- Logic path methods design tests to cover every path of execution in the program at least once

**Data & Data Flow Coverage**
- Data coverage methods explicitly try to cover the data aspects of the program code, rather than the control aspects

**Fault-Based Testing (e.g. Mutation Testing)**
- Mutation testing involves creating many slightly different versions of the code by mutating (changing operations) in each version
- Used to check sufficiency of test suites in detecting faults
Role of White Box Testing

Completeness for Black Box Methods

- White box code coverage gives a measure of completeness for open-ended black box methods
  - **Example:** Black box shotgun testing becomes a systematic method if we use code coverage (all statements executed at least once in the set of tests) as the completion criterion
Role of White Box Testing

Finds a Different Kind of Errors

- Black box testing finds errors of omission - that is, something that is specified that we have failed to do
- White box testing finds errors of commission - that is, something that we have done, but incorrectly

Automation

- Because white box testing involves the program code itself, which has a standard form, we can automate most of it
White Box Testing & Code Injection

**Code Injection**

- Injection is not itself a test method, but refers to modifications of the source or executable code being tested in order to make tests more effective (possible because white box)
  - **Example:** Modify the program to log each statement’s line number to a log file as it is executed, in order to check that every line is executed at least once by a test suite (*Produces a file of executed line numbers – check every line there*).
- Injection involves adding extra statements or instructions to execute that do not change what the original program does but checks or logs additional information about execution of the program (such as which statements have been executed).
- Original code is not changed, instead a separate copy with modifications is generated to run the tests on.
Applications of Code Injection

**Instrumentation Injection**
- Involves adding code to instrument the actions of the program at every method, statement or instruction during testing, to keep track of properties such as execution coverage.

**Performance Instrumentation**
- Involves adding code to log the actual time or space used by each method or statement of the program during execution.

**Assertion Injection**
- Involves adding strict run-time assertion code to every method, statement or instruction in the program during testing, to help localize the cause of failures.

**Fault Injection**
- Involves adding code to simulate run-time faults, to test fault handling.
Implementation of White Box Code Injection

Three Levels of Implementation

- Although it is not a necessity, white box testing usually involves validation of coverage using code injection.
- This code injection can be implemented in three separate ways:
  1. At the source level
  2. At the executable code level
  3. At the execution sampling level
Implementation of White Box Testing

Three Levels of Implementation

1. At the source level
2. At the executable code level

1 & 2: A copy of the program under test is altered to inject the additional source or executable code to log coverage as the program executes.

3. At the execution sampling level

3: The original program under test is run but with regular timer interrupts - at each interrupt, the current state and execution location at interrupt time can be sampled and logged before continuing execution.
Implementing Code Injection by Source Modification

- Create a copy of the program with new statements inserted to log coverage

Example: JTest

```java
final int mid = (lo + hi) / 2;
if (list[mid] == key)
    result = mid;
else if (list[mid] > key)
    hi = mid - 1;
else
    lo = mid + 1;
```
log.println (13);
13 final int mid = (lo + hi) / 2;
log.println (14);
14 if (list[mid] == key)
15 { log.println (15);
15     result = mid;
16 } else
16 { log.println (16);
17     if (list[mid] > key)
17         { log.println (17);
18             hi = mid - 1;
19         }
18     else
18         { log.println (18);
19             log.println (19);
20             lo = mid + 1;
21         }
20 }
```java
execount[13] += 1;
13 final int mid = (lo + hi) / 2;
execount[14] += 1;
14 if (list[mid] == key)
15 {
16     execount[15] += 1;
17         result = mid;
18     }
19 else
20 {
21     execount[16] += 1;
22         if (list[mid] > key)
23         {
24             execount[17] += 1;
25                 hi = mid - 1;
26         }
27     else
28     {
29         }
30     }
31     execount[18] += 1;
32     execount[19] += 1;
33     lo = mid + 1;
34 }
35 }
```
Executable Code Level Implementation

Implementing Code Injection by Executable Code Modification

- Create a **copy** of the program code with instructions inserted to log coverage
- In order not to change addresses, modify code to execute new instructions **out of line**
Example: Unix prof and gprof

<table>
<thead>
<tr>
<th>Mem. Loc.</th>
<th>Machine Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00A60</td>
<td>loada list,R4</td>
</tr>
<tr>
<td>00A64</td>
<td>add mid,R4</td>
</tr>
<tr>
<td>00A68</td>
<td>load key,R5</td>
</tr>
<tr>
<td>00A6C</td>
<td>comp R4,R5</td>
</tr>
<tr>
<td>00A70</td>
<td>jequ 00A84</td>
</tr>
</tbody>
</table>
**Example:** Unix prof and gprof

<table>
<thead>
<tr>
<th>Mem. Loc.</th>
<th>Machine Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00A60</td>
<td>jmp 07C80</td>
</tr>
<tr>
<td>00A64</td>
<td>add mid,R4</td>
</tr>
<tr>
<td>00A68</td>
<td>load key,R5</td>
</tr>
<tr>
<td>00A6C</td>
<td>comp R4,R5</td>
</tr>
<tr>
<td>00A70</td>
<td>jequ 00A84</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>07C80</td>
<td>loada execount,R4</td>
</tr>
<tr>
<td>07C84</td>
<td>addi #00A60,R4</td>
</tr>
<tr>
<td>07C88</td>
<td>addi #1,(R4)</td>
</tr>
<tr>
<td>07C8C</td>
<td>loada list,R4</td>
</tr>
<tr>
<td>07C90</td>
<td>jmp 00A64</td>
</tr>
</tbody>
</table>
Example: Unix prof and gprof

<table>
<thead>
<tr>
<th>Mem. Loc.</th>
<th>Machine Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00A60</td>
<td>jmp 07C80</td>
</tr>
<tr>
<td>00A64</td>
<td>add mid, R4</td>
</tr>
<tr>
<td>00A68</td>
<td>load key, R5</td>
</tr>
<tr>
<td>00A6C</td>
<td>comp R4, R5</td>
</tr>
<tr>
<td>00A70</td>
<td>jequ 00A84</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>07C80</td>
<td>loada execount, R4</td>
</tr>
<tr>
<td>07C84</td>
<td>addi #00A60, R4</td>
</tr>
<tr>
<td>07C88</td>
<td>addi #1, (R4)</td>
</tr>
<tr>
<td>07C8C</td>
<td>loada list, R4</td>
</tr>
<tr>
<td>07C90</td>
<td>jmp 00A64</td>
</tr>
</tbody>
</table>
Sampling Level Implementation

Implementing Code Injection by Execution Sampling

- Do not change the executable code at all
- Use a timer or other frequent regular interrupt to randomly sample where we are executing
- Interrupt return address tells us where we are executing when each interrupt happens
- After a large number of samples, results become statistically valid

```
07C80  timer:     loada  execount,R4
07C84      add     (SP),R4
07C88      addi    #1,(R4)
07C8C      rti
```
White Box Tools

Testing Tools

- Obviously implementing these strategies by hand programming would be tedious and time consuming.
- White box coverage testing is almost always supported by tools to implement the necessary code injections.
- Often test analysis and selection of test cases for white box testing can also be done automatically by modern tools.
Code Coverage Testing

**Code Coverage Methods**

- Two kinds: statement analysis (*flow independent*), decision analysis (*flow dependent*)
- Statement analysis methods
  - *statement* coverage
  - *basic block* coverage
- Decision analysis methods
  - *decision* coverage
  - *condition* coverage
  - *loop* coverage
  - *path* coverage
Code Coverage Testing

**Code Coverage Methods**

- Two kinds: statement analysis *(flow independent)*, decision analysis *(flow dependent)*
- Statement analysis methods
  - statement coverage
  - basic block coverage
- Decision analysis methods
  - decision coverage
  - condition coverage
  - loop coverage
  - path coverage
Statement Coverage

Statement Coverage Method

- Causes every statement in the program to be executed at least once, giving us confidence that every statement is at least *capable* of executing correctly

- **System:** Make a test case for each statement in the program, independent of the others
  - Test must simply cause the statement to be run, ignoring its actions and sub-statements (but still must check that result of test is correct)

- **Completion criterion:** A test case for every statement
  - Can be checked by *instrumentation injection* to track statement execution coverage
Example: Statement Coverage

```java
// calculate numbers less than x
// which are divisible by y
1 int x, y;
2 x = c.readInt ();
3 y = c.readInt ();
4 if (y == 0)
5   c.println ("y is zero");
6 else if (x == 0)
7   c.println ("x is zero");
8 else
9   { for (int i = 1; i <= x; i++)
10      { if (i % y == 0)
11         c.println (i);
12      }
13   }
```
**Example: Statement Coverage**

**Statement Coverage Tests**

- We blindly make one test for each statement, analyzing which *inputs* are needed to cause the statement to be executed
- Create test case for each unique set of inputs

<table>
<thead>
<tr>
<th>Statement x input</th>
<th>y input</th>
<th>1</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Statement Coverage

Statement Coverage Tests

<table>
<thead>
<tr>
<th>Statement</th>
<th>x input</th>
<th>y input</th>
<th>Test</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>T1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>T2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>T3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic Block Coverage

Basic Block Analysis Method

- Causes every basic block (indivisible sequence of statements) to be executed at least once - (usually) generates fewer tests
- **System:** Identify basic blocks by sequence analysis, design test case for each basic block
  - Sequence of statements in a row, ignoring sub-statements, such that if first is executed then following are all executed
- **Completion criterion:** A test case for every basic block
  - Can be checked by instrumentation injection to track statement execution coverage
Example: Basic Block Analysis

```java
// calculate numbers less than x
// which are divisible by y
int x, y;
x = c.readInt();
y = c.readInt();
if (y == 0)
   c.println("y is zero");
else
   if (x == 0)
      c.println("x is zero");
   else
      for (int i = 1; i <= x; i++)
      {
         if (i % y == 0)
            c.println(i);
      }
```

Example: Basic Block Analysis

Basic Block Coverage Tests

- We make one test for each block, analyzing which inputs are needed to cause the block to be entered
- Create test case for each unique set of inputs

<table>
<thead>
<tr>
<th>Block</th>
<th>x_input</th>
<th>y_input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Basic Block Analysis

Basic Block Coverage Tests

<table>
<thead>
<tr>
<th>Block</th>
<th>x_input</th>
<th>y_input</th>
<th>Test</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>T1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>T2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>T3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Testing Methods: White Box Testing I

Summary

- White box testing includes: code coverage, logic path/decision point coverage, data & data flow coverage, fault-based testing (e.g. mutation testing)
- White box methods often involve code injection to instrument execution using source modification, executable code modification or run time sampling
- Today we started to look at one class of code coverage methods: Statement analysis methods (statement, basic block coverage)

Next Time

- More code coverage methods and data coverage methods