CISC327 - Software Quality Assurance

Lecture 14
White Box Testing
White Box Testing

• 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
    • White box static analysis
White Box vs. Black Box

• **Recall:**
  – Systematic testing methods can be of two kinds
    • Black box and white box (or glass box)
  – **Black box** methods cannot see what the software code to test is (it may not exist yet), so they can only base their tests on the **requirements or specifications**
  – **White box** methods can see what the software's code is, so they can base their tests on the software's actual **architecture or code** itself
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

• **Mutation Testing**
  – Mutation testing involves creating many slightly different versions of the code by mutating (randomly changing) the code in each version
  – Used to check sufficiency of test suites for detecting faults
Kinds of White Box Testing

• Code Injection
  – Injection is not itself a test method, but refers to modifications of the source or executable code being tested to make tests mode effective
    • Possible because this is white box testing
  – Example:
    • Modify the program to log each statement's line number to a log file as it is executed, to check that every line is executed at least once by a test suite
    • Produces a file of executed line numbers, can check later that every line number is there
Kinds of White Box Testing

• **Code Injection**
  – Injection involves adding extra *statements* or *instructions* to execute that do not change what the original program does, but check or log additional information about *execution* of the program (such as which statements have been executed)
  – The 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 global invariants, resource usage or 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
Applications of Code Injection

• 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
Roles of White Box Testing

• Completeness for Black Box Methods
  – White box code coverage gives a measure of completeness for open-ended black box methods
  • For 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
Roles of White Box Testing

• Finds a Different Kind of Error
  – Black box testing finds errors of omission, something that is specified that we have failed to do
  – White box testing finds errors of commission, something that we have done, but incorrectly

• Automation
  – Because white box testing involves the program code itself, in a standard form, we can automate most of it
Implementation of White Box Testing

• Three Levels of White Box Implementation
  – Although it is not a necessity, white box testing usually involves validation of code coverage using code injection
  – This can be implemented in three separate ways
    • At the source level
    • At the executable code level
    • At the execution sampling level
Implementation of White Box Testing

• Three Levels of White Box Implementation
  – In the first two cases (source and executable levels), a copy of the program under test is altered to inject the additional source or executable code to log coverage as the program executes
  – In the third case (execution sampling level), 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
Source Level Implementation

• Implementing Code Injection by Source Modification
  – Create a copy of the program with new statements inserted to log coverage
  – Example: Jtest

<table>
<thead>
<tr>
<th>Source line number</th>
<th>Java Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>final int mid = (lo + hi) / 2;</td>
</tr>
<tr>
<td>14</td>
<td>if (list[mid] == key)</td>
</tr>
<tr>
<td>15</td>
<td>result = mid;</td>
</tr>
<tr>
<td>16</td>
<td>else if (list[mid] &gt; key)</td>
</tr>
<tr>
<td>17</td>
<td>hi = mid - 1;</td>
</tr>
<tr>
<td>18</td>
<td>else</td>
</tr>
<tr>
<td>19</td>
<td>lo = mid + 1;</td>
</tr>
</tbody>
</table>
Source Level Implementation

```java
log.println(13);
final int mid = (lo + hi) / 2;
log.println(14);
if (list[mid] == key) {
    log.println(15);
    result = mid;
} else {
    log.println(16);
    if (list[mid] > key) {
        log.println(17);
        hi = mid - 1;
    } else {
        log.println(18);
        log.println(19);
        lo = mid + 1;
    }
}
```

- Log file method
  - Insert statements to print to a log file
  - Analyze log file
Source Level Implementation

```java
execount[13] += 1;
13 final int mid = (lo + hi) / 2;
excount[14] += 1;
14 if (list[mid] == key)
{
    execount[15] += 1;
    result = mid;
} else
{
    execount[16] += 1;
    execount[18] += 1;
    lo = mid + 1;
}
}
```

- **Coverage array method**
  - Insert statements to increment global array elements
  - Analyze array
Executable Code Level Implementation

• Implementing Code Injection by Executable Code Modification
  – Create a copy of the executable 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>Memory Location</th>
<th>Machine Instruction (Assembly Language)</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>
# Executable Code Level Implementation

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Machine Instruction (Assembly Language)</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>07C80</td>
<td>loada execcount, 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>

execcount[0xA60]+=1
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

execount[where_at]+=1
```
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
Static Analysis

• Have Code: Why Not Prove?
  – The source analysis to automatically generate tests is a complex and sophisticated flow analysis of the program
  – A very similar analysis can actually prove many of the cases, automatically finding problems or eliminating test cases before they are ever run
Static Analysis

• **Example**: Euclid compiler
  
  – *Euclid* was a precursor to the *Turing* language
  – The Euclid compiler was designed to **prove** that subscripts and pointers were never out of range, that pre- and post- assertions were always true, and so on
  – The compiler inserted code to check these "**legality"** conditions at run time **only** in cases where it could not prove them at compile time ("**statically**")
  – In practice, the compiler was able to prove almost all legality conditions, reducing the overhead of run time checking to **less than 10%** of run time of the program
Summary

• **White Box Testing**
  – White box testing includes *code coverage, logic path,* and *mutation* testing
  – White box methods often involve *code injection* to instrument execution using *source* modification, executable *code* modification, or run time *sampling*
  – *Static analysis* can reduce white box testing effort and cost using automatic proofs

• **Next time**
  – Code coverage methods