CISC327 - Software Quality Assurance

Lecture 17
White Box Testing
Independent Paths

• Question
  – Why not make one path that covers all decisions, and have done with it?

while (hi >= lo)
  if (result != -1)
    if (list[mid] == key)
      if (list[mid] > key)
Independent Paths

• **Question**
  – Why not make one path that covers all decisions, and have done with it?
Independent Paths

• Answer

  – "Independent paths" in the sense of path testing means linearly independent, a formal definition from graph theory beyond the scope of this course

  – But basically, a set of paths is linearly independent if no one is a linear combination of the others

  – So if we make such a long path, we can still make a set of others for which none is a linear combination of the others
Independent Paths

- Easiest way to find such a set is to work from top-to-bottom, left-to-right to create paths

P1: 1, 2, 12, 13

P2: 1, 2, 3, 4, 12, 13

P3: 1, 2, 3, 5, 6, 11, 2, 12, 13

P4: 1, 2, 3, 7, 8, 10, 11, 2, 12, 13

P5: 1, 2, 3, 7, 9, 10, 11, 2, 12, 13
White Box Testing

• Outline
  – Today we continue our look at white box testing methods with mutation testing
  – We'll look at:
    • Definition and role of mutation testing
      – What is a mutation?
      – How is mutation testing used?
    • Mutation testing methods
      – Value mutations
      – Decision mutations
      – Other mutations
      – Examples
Mutation Testing

• What is it for?
  – Mutation testing is a white box method for checking the adequacy of a test suite
  – As you have already discovered, creating a test suites can be an expensive and time consuming effort
  – No matter what test method is used, discovering if test suites are adequate to uncover faults is itself an even more difficult task
  – Mutation testing offers an almost completely automated way to check the adequacy of a set of tests in uncovering faults in the software
Mutation Testing

• How does it work?
  – In order to test the adequacy of a test suite, we first run the software on the suite and fix any problems until we are satisfied that the software passes the tests
  – We then save the results of the tests in a file or set of files to serve as the correct output to compare to
  – We then use mutation of the source code to create a set of mutants, each of which is a program slightly different from the original
Mutation Testing

• How does it work?
  – For each mutant, we run the test suite on the mutant and compare the results to the saved results from the original
  – If the results differ, then the mutant has been "killed" (detected) by the test suite
  – If the results do not differ, then the test suite is inadequate to detect the mutant, and a new test must be added to the suite to "kill" that mutant
Systematic Mutation

• Mutants

  – For mutation testing to be systematic, there must be a system and a completion criterion for creating mutants
  – The system for mutation normally specifies simple syntactic changes to the program source representing errors in the code
  – Each mutant has exactly one change in it
  – We are done when every possible single change mutant of the system has been generated and tested
Systematic Mutation

• Mutants

  – Example systematic mutations are:
    • Value mutations (changing constants, subscripts, or parameters by adding or subtracting one, etc.)
    • Decision mutations (inverting or otherwise modifying the sense of each decision condition in the program)
    • Statement mutation (deleting or exchanging individual statements in the program)
Example 1: Value Mutation

• Value Mutation Example
  – System: Mutate the value of each constant in the program to be off by one (or more generally, each integer expression)
  – Completion criterion: One mutant for each constant in the program
  – Note that there are many other possible value mutations:
    • Constants modified in some other way, e.g. off by -1
    • All integer expressions modified (not just constants) e.g. x changed to x+1, etc.
Example 1: Value Mutation

// 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 0");
else if (x == 0)
  c.println("x is 0");
else {
  for (int i = 1; i <= x; i++) {
    if (i % y == 0)
      c.println(i);
  }
}

Example test suite (statement coverage)

<table>
<thead>
<tr>
<th>Test</th>
<th>x</th>
<th>y</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>&quot;y is 0&quot;</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>&quot;x is 0&quot;</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example 1: Value Mutation

```java
// calculate numbers less than x
// which are divisible by y

int x, y;
x = c.readInt();
y = c.readInt();

if (y == 1) {  // New mutant
    c.println("y is 0");
} else if (x == 0) {
    c.println("x is 0");
} else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

### Table

<table>
<thead>
<tr>
<th>Test</th>
<th>x</th>
<th>y</th>
<th>output</th>
<th>mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>&quot;y is 0&quot;</td>
<td>&quot;x is 0&quot;</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>&quot;x is 0&quot;</td>
<td>&quot;y is 0&quot;</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>&quot;y is 0&quot;</td>
</tr>
</tbody>
</table>
Example 1: Value Mutation

```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 0");
else if (x == 0)
c.println("x is 0");
else {
  for (int i = 2; i <= x; i++) {
    if (i % y == 0)
c.println(i);
  }
}

// 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 0");
else if (x == 0)
c.println("x is 0");
else {
  for (int i = 1; i <= x; i++) {
    if (i % y == 1)
c.println(i);
  }
}
```

<table>
<thead>
<tr>
<th>Test</th>
<th>x</th>
<th>y</th>
<th>output</th>
<th>mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>&quot;y is 0&quot;</td>
<td>&quot;y is 0&quot;</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>&quot;x is 0&quot;</td>
<td>&quot;x is 0&quot;</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example 2: Decision Mutation

- **Decision Mutation Example**
  - **System**: Invert the sense of each decision condition in the program
    - e.g., change > to <, == to !=, and so on
  - **Completion criterion**: One mutant for each decision condition in the program
Example 2: Decision Mutation

// 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 0");
else if (x == 0)
c.println("x is 0");
else {
  for (int i = 1; i <= x; i++) {
    if (i % y == 0)
      c.println(i);
  }
}

Example test suite (statement coverage)

<table>
<thead>
<tr>
<th>Test</th>
<th>x</th>
<th>y</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>&quot;y is 0&quot;</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>&quot;x is 0&quot;</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example 2: Decision Mutation

```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 0");
else if (x == 0)
c.println("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println(i);
    }
}
```

<table>
<thead>
<tr>
<th>Test</th>
<th>x</th>
<th>y</th>
<th>output</th>
<th>mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>&quot;y is 0&quot;</td>
<td>&quot;x is 0&quot;</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>&quot;x is 0&quot;</td>
<td>&quot;y is 0&quot;</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>&quot;y is 0&quot;</td>
</tr>
</tbody>
</table>

// 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 0");
else if (x == 0)
c.println("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println(i);
    }
}
Example 2: Decision Mutation

// 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 0");
else if (x == 0)
c.println("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
c.println(i);
    }
}

// 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 0");
else if (x == 0)
c.println("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y != 0)
c.println(i);
    }
}
Example 3: Statement Mutation

- **Statement Mutation Example**
  - **System**: Delete each statement in the program
  - **Completion criterion**: One mutant for each statement
  - Note that there are many other possible statement mutations:
    - Interchanging adjacent statements
    - Shuffling sequences of statements
    - Doubling statements
    - Many more
// 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 0");
else if (x == 0)
    c.println("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println(i);
    }
}

<table>
<thead>
<tr>
<th>Test</th>
<th>x</th>
<th>y</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>&quot;y is 0&quot;</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>&quot;x is 0&quot;</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example 3: Statement Mutation

```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 0");
else if (x == 0)
    c.println ("x is 0");
else {
    for (int i = 1; i <= x; i++) {
        if (i % y == 0)
            c.println (i);
    }
}
```

- This time, we show only **one example** - you can make the rest!
- All statement mutants of this program turn out to be "killed" by our simple test set

<table>
<thead>
<tr>
<th>Test</th>
<th>x</th>
<th>y</th>
<th>output</th>
<th>mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>&quot;y is 0&quot;</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>&quot;x is 0&quot;</td>
<td>&quot;x is 0&quot;</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Determining Test Suite Adequacy

• Mutation adequacy score
  – If $D$ is the number of dead mutants (program variations that were caught by our existing test suite), and $M$ is the total number of mutants

Mutation Adequacy Score = $D / M$
Some Observations

• In practice, simple statement coverage tests are often sufficient to "kill" most kinds of mutants

• Thus they can detect most kinds of accidental faults that might be introduced into a working program
  – Not really a surprise, when you think about it, and the main reason coverage tests are worth doing
    (more on this next time)

• However, mutation can catch missing test cases even in coverage tests

• Since most projects use primarily black box techniques, automated mutation testing can be a very valuable help in making test suites more effective
Advantages and Disadvantages

• **Advantages**
  
  – Provides a good check for quality of a test suite, however created
  
  – Once “baseline” of correct results of a test suite has been checked, testing adequacy of the suite using mutation can be **automated**

• **Disadvantages**
  
  – **Expensive** - generates a huge number of mutants, many really checking the same cases
  
  – Detecting mutant **equivalence** is a big problem
Summary

• **Mutation Testing**
  – Mutation testing is a white box method for **automatically** checking test suites for completeness
  – Mutations are simple, **syntactic** variants of programs that can be generated automatically
  – Typical mutations are **value** mutations, **decision** mutations, **statement** mutations
  – Mutation can find **missing test cases** in a test suite
  – **Statement coverage** is a strong testing system, usually "kills" most kinds of mutants

• **References:** Van Vliet 13.6

• **Next Time**
  – Continuous testing methods