SOFTENG 254:
Quality Assurance

Lecture 2b: Developing good tests 1

Paramvir Singh
School of Computer Science

Agenda

- How to come up with good tests
- Statement Coverage
- Evaluation of Statement Coverage

What is a “good” test?

Developing good tests

- the purpose of testing is to detect the presence of faults
- we detect faults by executing the code in such a way to cause a failure

⇒ the higher the probability of causing a failure (assuming a fault exists), the better the quality of the test

⇒ when deciding on what to test, need to look at where faults are likely to be

- really need to consider quality of the test suite — individual tests are of limited value

Example

Inputs: string where the only whitespace characters are space and tab, integer indicating what tab stop to use

Outputs: the number of space characters between the beginning of the string and the first non-whitespace character, assuming any tab characters are replaced by enough spaces corresponding to the tabstop

Tab stops 4

```
                                    +   +   +   +(tab stops)
#1|Lorem \tipsum dolor              |Lorem    ipsum dolor
#2|\t\tsit amet,                    |         sit amet,
#3| \tconsectetur                   |     consectetur
```

- leadingSpacesCount(#1, 4) = 0
- leadingSpacesCount(#2, 4) = 8
- leadingSpacesCount(#3, 4) = 4

Example solution

```
 public static int leadingSpacesCount(String text, int tabstop) {
A   int index =0;
B   int count =0;
C.  int interTab =0;
D.  char[] chars =text.toCharArray();
E   while (index <chars.length &&
          Character.isWhitespace(chars[index])) {
F     if (chars[index] ==’\t’) {
G       count +=tabstop - interTab;
H       interTab =0;
      }else {
I       if (interTab ==tabstop - 1) {
J         interTab =1;
        }else {
K         interTab++;
        }
L       count++;
      }
M     index++;
    }
N   return count;
 }
```

Test Suite Quality

- Consider this test suite:
  - Input = ("Lorem", 4), Expected Output: 0
- Is it improved by adding this test case?
  - Input = ("Lorem ipsum", 4), Expected Output: 0
- More test cases does not mean a better quality test suite

Test Suite Quality

- Is this test suite good enough?
  - Input = (" Lorem", 4), Expected Output: 1
  - Input = ("\tLorem", 4), Expected Output: 4

Statements Executed

```
    public static int leadingSpacesCount(String text, int tabstop) {
A     int index = 0;
B     int count = 0;
C     int interTab = 0;
D     char[] chars = text.toCharArray();
E     while (index < chars.length &&
              Character.isWhitespace(chars[index])) {
F       if (chars[index] == '\t') {
G         count += tabstop - interTab;
H         interTab = 0;
        } else {
I         if (interTab == tabstop - 1) {
J           interTab = 1;
          } else {
K           interTab++;
          }
L         count++;
        }
M       index++;
      }
N     return count;
    }
```

Statements Executed

```
    public static int leadingSpacesCount(String text, int tabstop) {
A     int index = 0;
B     int count = 0;
C     int interTab = 0;
D     char[] chars = text.toCharArray();
E     while (index < chars.length &&
             Character.isWhitespace(chars[index])) {
F       if (chars[index] == '\t') {
G         count += tabstop - interTab;
H         interTab = 0;
        }else {
I         if (interTab == tabstop - 1) {
J           interTab = 1; // hic sunt dracones!
          }else {
K           interTab++;
          }
L         count++;
        }
M       index++;
      }
N     return count;
    }
```

Intuition

- Adding a test to a test suite should depend on whether or not it increases the probability of a failure
- In testing, we detect the existence of faults by executing code and causing a failure

⇒ the faults we detect is somewhere in the code that was executed

⇒ faults in code we don't execute cannot have been detected

⇒ we increase the probability of a test suite detecting a fault by adding tests that execute code that hasn't already been executed

Statement coverage

- what proportion of the statements in the code are executed by the test suite?
- $$\text{Statement coverage} = \frac{\#\text{statements executed by all tests}}{\text{total number of statements}}\%$$
- If statement coverage is not 100%, then there is a possibility of a fault existing in the code that will not be detected by the test suite
- a measure of test suite quality
- Example: Statement coverage = $\frac{13}{14}$ = 93% (roughly)

Test Suite Quality

- Input = (" Lorem", 4), Expected Output: 1
- Input = (“\tLorem", 4), Expected Output: 4
- Input = (" \tLorem", 4), Expected Output 8 ⇒ Actual output 7 — Failure!
- 100% statement coverage

How good is Statement Coverage?

- Suppose we had added a different test case
  - Input = (" Lorem", 4), Expected Output: 1
  - Input = ("\tLorem", 4), Expected Output: 4
  - Input = (" Lorem", 4), Expected Output 4 ⇒ Actual output 4 — Pass!
- Also 100% statement coverage
- All tests pass
- **but!** we know the implementation has at least one fault

⇒ 100% statement coverage is not enough

- In fact, 100% statement coverage is not always possible, e.g. dead code

Dead Code

```
    public static int leadingSpacesCount(String text, int tabstop) {
A     int index =0;
B     int count =0;
C     int interTab = 0;
D     char[] chars =text.toCharArray();
E     while (index <chars.length &&
            Character.isWhitespace(chars[index])) {
F       if (index >=chars.length) {
G         break;
        }
H       if (chars[index] ==’\t’) {
I         count +=tabstop - interTab;
J         interTab =0;
        }else {
K         if (interTab ==tabstop - 1) {
L           interTab =1;
          }else {
M           interTab++;
          }
N         count++;
        }
O       index++;
      }
P     return count;
  }
```

Test Strategies

- To improve test suites, we need to consider both the artifact we are testing (code) and the quality attribute we are trying to establish (correctness)
- White (clear, glass) box testing — performing tests based on **how** the component is implemented ⇒ artifact focus

  — decide on tests based on the code itself
  - E.g., statement coverage

  — not good at detecting faults relating to requirements
- Black box testing — performing tests based on **what** the component is supposed to do ⇒ quality attribute focus

  — decide on tests based on the requirements independent of the code

  — not good at detecting faults in implementation decisions
- Black box versus White box now something of an old-fashioned view of testing. The process for developing a test suite is essentially the same for both—it is the inputs that are different (requirements versus code).

Key Points

- If there is a statement that is not executed by the existing test suite, then adding a test that causes that statement to be executed increases the probability of causing a failure
- 100% statement coverage is neither sufficient or always possible
- But,
  - a good start
  - example of general strategy — find situations where faults may lie and develop tests to expose them