CSC444F:
Software Engineering I

Mou Hu
mou.hu@utoronto.ca

Lecture 13: Implementation (II)

- Unit Testing
- Reading: Sections 13.1, 13.5

Unit Testing

- Unit testing is conducted to ensure that produced program units meet their specifications.
- Unit testing is typically conducted by the Development Team and specifically by the programmer who coded the unit.
- Activities of Unit Testing

  Analysis – Choosing a unit, analyzing the functionality (based on specification) and/or the code structure

  Design – Designing test cases (using black-box and/or white-box approaches)

  Implementation - Coding the test class and test methods and compiling

  Execution – Loading test cases to the test tool, running test cases, writing test report

Goals of Unit Testing

- Goal: show a program meets its specification
  - But: testing can never be complete for non-trivial programs
  - Only under certain assumption, testing can be complete
- What is a successful test?
  - One in which no errors were found?
  - One in which one or more errors were found?
- Successful testing should be:
  - repeatable
    - if you find an error, you'll want to repeat the test to show others
    - if you correct an error, you'll want to repeat the test to check you did fix it
  - systematic
    - random testing is not enough
    - select test cases that cover the range of behaviors or structures of the program
    - select test cases that are representative of real uses
  - documented
    - keep track of what tests were performed, and what the results were

Adapted from Ref. [3]

Random Testing Isn't Enough

- **Structurally…**

```
boolean equal (int x, y) {
/* effects: returns true if
  x=y, false otherwise
*/
if (x == y)
  return(TRUE)
else
  return(FALSE)
}
```

- Test strategy: pick random values for x and y and test 'equals' on them
- But:
  - ...we might never test the first branch of the 'if' statement

➲ **Functionally…**

```
int maximum (list a)
/* requires: a is a list of
  integers
  effects: returns the maximum
  element in the list
*/
```

Try these test cases:

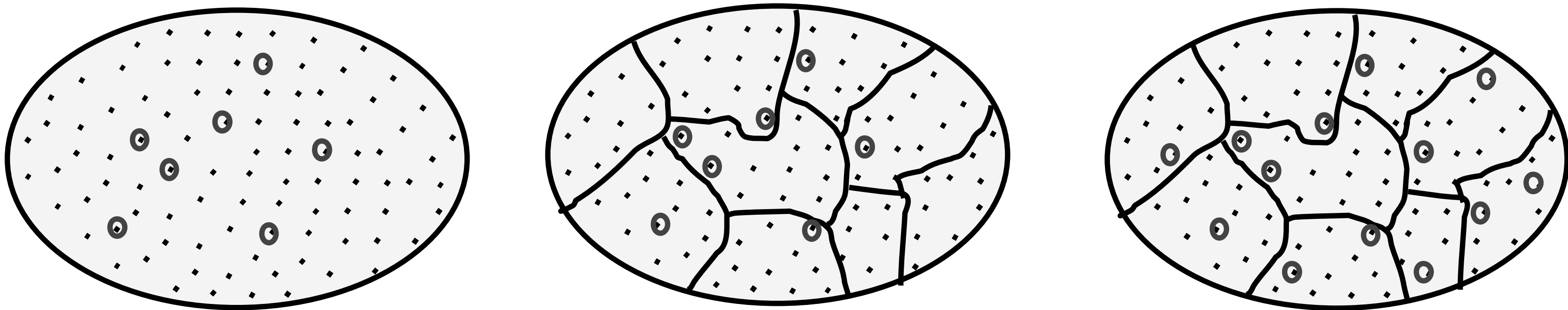
Input	Output	Correct?
3 16 4 32 9	32	Yes
9 32 4 16 3	32	Yes
22 32 59 17 88 1	88	Yes
1 88 17 59 32 22	88	Yes
1 3 5 7 9 1 3 5 7	9	Yes
7 5 3 1 9 7 5 3 1	9	Yes
9 6 7 11 5	1	Yes
5 11 7 6 9	1	Yes
561 13 1024 79 86 222 97	1024	Yes
97 222 86 79 1024 13 561	1024	Yes

Why is this not enough?

Source: Ref. [3]

Partitioning

- Systematic testing depends on partitioning
  - partition the input space into partitions which cover all possible behaviors or structures of the system
  - choose representative samples from each partition
  - make sure we covered all partitions



- How do you identify suitable partitions?
  - That's what testing is all about!!!
  - Methods:
    - black box, white box, ...

Adapted from Ref. [3]

Black Box Testing

- Generate test cases from the specification only
- Advantages:
  - avoids making the same assumptions as the programmer
  - test data is independent of the implementation
  - results can be interpreted without knowing implementation details
- Three suggestions for selecting test cases:
  - Paths through the spec
    - e.g. choose test cases that cover each part of the 'requires', 'modifies' and 'effects' clauses
  - Boundary conditions
    - choose test cases that are at or close to boundaries for ranges of inputs
  - Off-nominal cases
    - choose test cases that try out every type of invalid input (the program should degrade gracefully, without loss of data)

Source: Ref. [3]

Example

```
real sqrt (real x, epsilon) {
/* requires: x ≥ 0 and (0.00001 < epsilon < 0.001)
   effects: returns y such that x-epsilon ≤ y² ≤ x+epsilon
*/
```

- paths through the spec:
  - "x ≥ 0" means "x>0 or x=0", so test both "paths"
  - it is not always possible to choose tests to cover the effects clause…
    - can't choose test cases for "$x\text{-epsilon}=y^2$" or "$y^2=x\text{+epsilon}$"
- boundary conditions:
  - As "x ≥ 0" choose:
    - 1, 0, -1 as values for x
  - As "0.00001 < epsilon < 0.001" choose:
    - 0.000011, 0.00001, 0.0000099, 0.0011, 0.001, 0.00099, as values for epsilon
  - very large & very small values for x
- off-nominal cases:
    - negative values for x and epsilon
    - values for epsilon > 0.001, values for epsilon < 0.00001

Source: Ref. [3]

The classic example

```
char * triangle (unsigned x, y, z) {
/* effects: If x, y and z are the lengths of the sides of a
  triangle, this function returns one of three strings,
  "scalene", "isosceles" or "equilateral" for the given
  three inputs.
*/
```

- How many test cases are enough?
  - expected cases (one for each type of triangle): (3,4,5), (4,4,5), (5,5,5)
  - boundary cases (only just not a triangle): (1,2,3)
  - off-nominal cases (not valid triangle): (4,5,100)
  - vary the order of inputs for expected cases: (4,5,4), (5,4,4)
  - vary the order of inputs for the boundary case: (1,3,2), (2,1,3), (2,3,1), (3,2,1), (3,1,2)
  - vary the order of inputs for the off-nominal case: (100,4,5), (4,100,5)
  - choose two equal parameters for the off-nominal case: (100,4,4)
- Note: there is a bug in the specification!!

Source: Ref. [3]

White box testing

- Examine the code and test all paths
  - because black box testing can never guarantee we exercised all the code
- Path completeness:
  - A test set is path complete if each path through the code is exercised by at least one case in the test set (not the same as saying each statement in the code is reached!!)

```
int midval (int x, y, z) {
/* effects: returns median
  value of the three inputs
*/
if (x > y) {
  if (y > z) return y
  else if (x > z) return z
        else return x }
else {
  if (x > z) return x
  else if (y > z) return z
        else return y } }
```

There are 6 paths through this code

…so we need at least 6 test cases

e.g.		
e.g.	x=4, y=3, z=2	return 3
	x=4, y=2, z=3	return 3
	x=3, y=2, z=4	return 3
	x=3, y=4, z=2	return 3
	x=2, y=4, z=3	return 3
	x=2, y=3, z=4	return 3

Adapted from Ref. [3]

Weaknesses of path completeness

- White box testing is insufficient
  - e.g.

```
int midval (int x, y, z) {
/* effects: returns median
  value of the three inputs
*/
return z; }
```

  - The single test case x=4, y=1, z=2 is path complete
    - the program performs correctly on this test case
    - but the program is still wrong!!
- Path completeness is usually infeasible
  - e.g.

```
for (j=0, i=0; i<100; i++)
  if a[i]=true then j=j+1
```

  - there are 2^{100} paths through this program segment
  - loops are problematic.
- Reduce to branch completeness
  - cover all branches
  - for the above example, how many branches?
  - how many test cases needed?

Adapted from Ref. [3]

Test Case Implementation and Execution

- Test cases are implemented by writing test class and test methods. This step is language dependent and tool specific.
- One of unit testing tool is JUnit, which will be covered in the tutorial.

  Reference: http://junit.sourceforge.net/

Test class format for JUnit:

```
public class TestClassName extends TestCase {
   //Test method
   public void testMethodName() {
             //Arrange: create some objects to be tested.
             …
             //Act: Stimulate them with predefined conditions.
             …
             //Assert: Check the results.
             …
   }
}
```

References

[1] Hans van Vliet, “Software Engineering: Principles and Practice”, John Wiley and Sons, Ltd., 2000.

[2] Rick D. Craig and Stefan P. Jaskiel, “Systematic Software Testing”, Artech House Publishers, 2002.

[3] Steve Easterbrook, “Lecture Notes”, University of Toronto, 2001.