Lecture 8: Testing

- Verification and Validation
  - testing vs. static analysis
- Testing - how to partition the space
  - black box testing
  - white box testing
- System level tests
  - integration testing
  - other system tests
- Automated testing

Verification and Validation

- Validation
  - does the software do what was wanted?
    - "Are we building the right system?"
  - This is difficult to determine and involves subjective judgements
- Verification
  - does the software meet its specification?
    - "Are we building the system right?"
  - This can be objective if the specifications are sufficiently precise

Three approaches to verification

- Everything must be verified
  - ...including the verification process itself

Goals of Testing

Source: Adapted from van Vliet, 2000, Section 13.1

- Goal: show a program meets its specification
  - But: testing can never be complete for non-trivial programs
- What is a successful test?
  - One in which no errors were found?
  - One in which one or more errors were found?
- 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 sets that cover the range of behaviors of the program
    - select test sets that are representative of real uses
  - documented
    - keep track of what tests were performed, and what the results were

Random testing isn't enough

Source: Adapted from Horton, 1999

- 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:

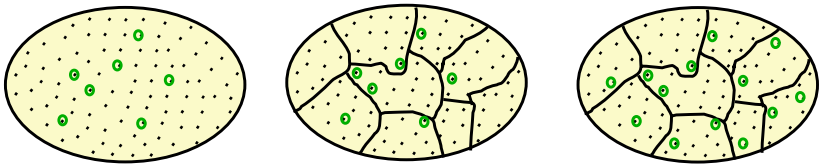
Input	Output	Correct?
3 16 4 32 9	32	Yes
9 32 4 16 3	32	Yes
22 32 59 17 88 1	88	Yes
1 88 17 59 32 22	88	Yes
1 3 5 7 9 1 3 5 7	9	Yes
7 5 3 1 9 7 5 3 1	9	Yes
9 6 7 11 5	1	Yes
5 11 7 6 9	1	Yes
561 13 1024 79 86 222 97	1024	Yes
97 222 86 79 1024 13 561	1024	Yes

Why is this not enough?

Partitioning

Source: Adapted from Horton, 1999

- Systematic testing depends on partitioning
  - partition the set of possible behaviours 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, ...

Black Box Testing

- Generate test cases from the specification only
  - (i.e.don't look at the code)
- 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
    - test for aliasing errors (e.g. two parameters referring to the same object)
  - Off-nominal cases
    - choose test cases that try out every type of invalid input (the program should degrade gracefully, without loss of data)

Example

Source: Adapted from Liskov & Guttag, 2000, pp224-5

```
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-epsilon=y^2" or "y^2=x+epsilon"
    - if the algorithm always generates positive errors, can't even generate y^2 < x
- 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

The classic example

Source: Adapted from Blum, 1992, pp405-406

- Consider the following program:

```
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!!

White box testing

Source: Adapted from Liskov & Guttag, 2000, pp227-229

→ 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!!)

→ Example

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

There are 4 paths through this code

…so we need at least 4 test cases

e.g. x=3, y=2, z=1
x=3, y=2, z=4
x=2, y=3, z=2
x=2, y=3, z=4

Weaknesses of path completeness

Source: Adapted from Liskov & Guttag, 2000, pp227-229 and van Vliet 1999, section 13.5

→ 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. Try:
  - test 0, 1, 2, n-1, and n iterations, (n is the max number of iterations possible)
  - or try formal analysis - find the "loop invariant"!!

Integration Testing

Source: Adapted from van Vliet 1999, section 13.9

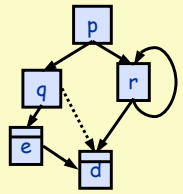
→ Unit testing

- each unit is tested separately to check it meets its specification

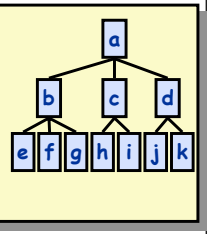
→ Integration testing

- units are tested together to check they work together
- two strategies:

Bottom up
for this dependency graph, the test order is:
1) d
2) e and r
3) q
4) p



Top down
for this structure chart the order is:
1) test a with stubs for b, c, and d
2) test a+b+c+d with stubs for e…k
3) test whole system



→ Integration testing is hard:

- much harder to identify equivalence classes
- problems of scale
- tends to reveal specification errors rather than integration errors

System testing

Source: Adapted from Blum, 1992, pp415-416

→ Other types of test

- facility testing - does the system provide all the functions required?
- volume testing - can the system cope with large data volumes?
- stress testing - can the system cope with heavy loads?
- endurance testing - will the system continue to work for long periods?
- usability testing - can the users use the system easily?
- security testing - can the system withstand attacks?
- performance testing - how good is the response time?
- storage testing - are there any unexpected data storage issues?
- configuration testing - does the system work on all target hardware?
- installability testing - can we install the system successfully?
- reliability testing - how reliable is the system over time?
- recovery testing - how well does the system recover from failure?
- serviceability testing - how maintainable is the system?
- documentation testing - is the documentation accurate, usable, etc.
- operations testing - are the operators' instructions right?
- regression testing - repeat all testing every time we modify the system!

Automated Testing

Source: Adapted from Liskov & Guttag, 2000, pp239-242

- **Ideally, testing should be automated**
  - tests can be repeated whenever the code is modified ("regression testing")
  - takes the tedium out of extensive testing
  - makes more extensive testing possible
- **Will need:**
  - test driver - automates the process of running a test set
    - sets up the environment
    - makes a series of calls to the unit-under-test
    - saves results and checks they were right
    - generates a summary for the developers
  - test stub - simulates part of the program called by the unit-under-test
    - checks whether the UUT set up the environment correctly
    - checks whether the UUT passed sensible input parameters to the stub
    - passes back some return values to the UUT (according to the test case)
    - (stubs could be interactive - ask the user to supply return values)

References

Horton, D. "Testing Software" Course handout, University of Toronto, 1999.

This excellent introduction to systematic testing is available from the readings page on the course website, or at http://www.cs.toronto.edu/~dianeh/148/handbook/testing.ps

van Vliet, H. "Software Engineering: Principles and Practice (2nd Edition)" Wiley, 1999.

Chapter 13 provides an excellent overview of the whole idea of testing software. van Vliet's treatment complements this lecture very nicely - he covers the philosophy of testing and the kinds of errors that occur in software. Instead of using black box vs white box as his test selection criteria, he uses "coverage-based", "fault-based" and "error-based". As an exercise, try mapping one of these classifications onto the other, and see what you get!

Liskov, B. and Guttag, J., "Program Development in Java: Abstraction, Specification and Object-Oriented Design", 2000, Addison-Wesley.

Liskov and Guttag's chapter 10 is a nice introduction to testing of procedural and data abstractions.

Blum, B. "Software Engineering: A Holistic View". Oxford University Press, 1992

Section 5.3 provides an excellent overview of the whole idea of testing software.