CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Testing

Feb. 15, 2007

Some slides adapted from FSE'98 Tutorial by Michal Young and Mauro Pezze'

Testing

- Execute program on sample input data
  - Check if output correct (acceptable)
- Goals
  - Increase confidence program works correctly
    - Acceptance Testing
  - Find bugs in program
    - Debug Testing

Example (Black Box)?

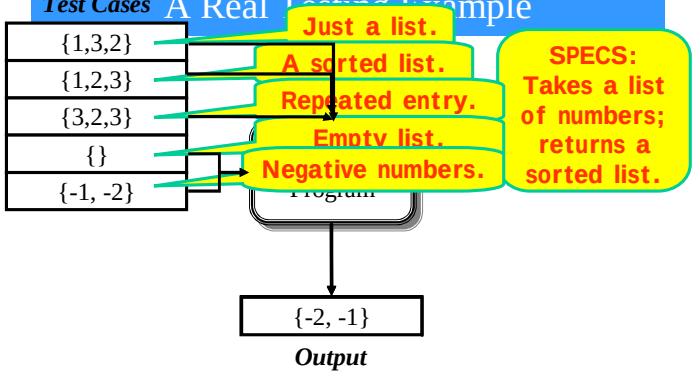
```
% java TestServlet HelloWorld /FooBar/Test > out

  HTTP/1.0 200
  Content-Type: text/plain

  Hello /FooBar/Test

% diff out expectedOutput
```


Test Cases A Real Testing Example

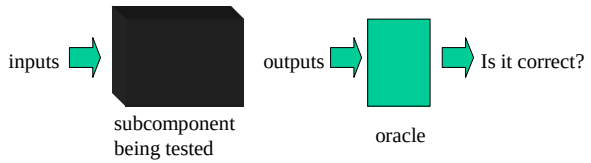


Limitations of Testing

- Program runs on (very small) *subset* of input data
  - Exhaustive testing usually impossible
    - Too large input space (possibly infinite)
- Many situations hard to test
  - Parallel code (due to non-determinism)
  - Hard-to-reach states (e.g., error states)
  - Inadequate test environment (e.g., lack of hardware)
- Testing cannot prove absence of bugs
  - Especially a problem in security

Black Box Testing

- Pick subcomponent of program
  - Internals of component not considered
- Give it inputs
- Compare against expected outputs



Black Box Testing

- Pick subcomponent of program
  - Internals of component not considered
- Give it inputs
- Compare against expected outputs

  - But how do I know what the expected outputs are?
  - Depends on the software specification …

The Test Case Generation Problem

- What tests will show that my program works?
  - Must consider "operational scenarios"
    - What is legitimate input?
    - What is the correct action or output?
  - Must consider "abnormal behaviors" as well
- How can I make sure that all of the important behaviors of my program have been tested?
  - Usually, you can't!

Test Cases via Specifications

```
// Return true if x in a, else returns false
boolean contains(int[] a, int x);
```

- Two "paths" in specification
  - Test case where x is in a
  - Test case where x is not in a

Test Cases via Inferred Implementation

- Think about how the implementation might look
  - Test by boundary condition
    - What test cases are likely to exercise the same logic?
    - Want to avoid redundant tests, to save time
  - Test by common mistake
    - What cases my be tricky to implement?
- At the same time, tests should still be implementation-independent

Test Cases via Boundary Conditions

```
interface List { ...
```

Inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

```
  public void add(int index, Object element)
}
```

- Test with empty list
- Test with index at first/last element
- Others?

Test Cases via Common Mistakes

```
// Appends l2 to the end of l1
void append(List l1, List l2);
```

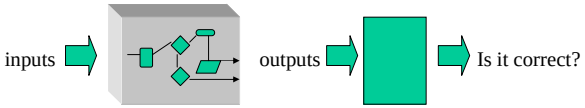
- Does append work if l1==l2?

```
class A {
  ...boolean equals(...);
}
```

- Does equals work if operand is an Object?

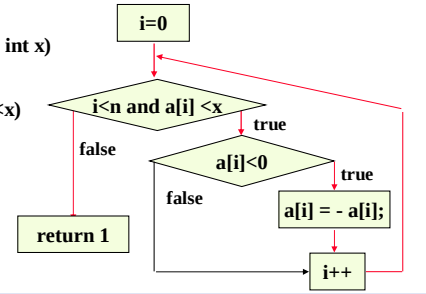
White/Glass Box Testing

- Pick subcomponent of program
- Give it inputs
  - Based on component code
    - If you don't execute the code, you don't know whether or not it works
- Compare against correct outputs (properties)



Statement Coverage

```
int select(int[] a, int n, int x)
{
  int i=0;
  while (i<n && a[i] <x)
  {
     if (a[i]<0)
        a[i] = - a[i];
     i++;
  }
  return 1;
}
```

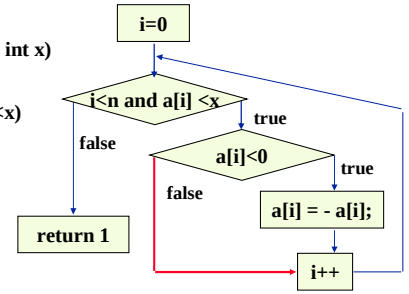


One test case (n=1, a[0]=-7, x=9) covers all statements

Faults handling positive values of a[i] not revealed

Branch Coverage

```
int select(int[] a, int n, int x)
{
  int i=0;
  while (i<n && a[i] <x)
  {
     if (a[i]<0)
        a[i] = - a[i];
     i++;
  }
  return 1;
}
```

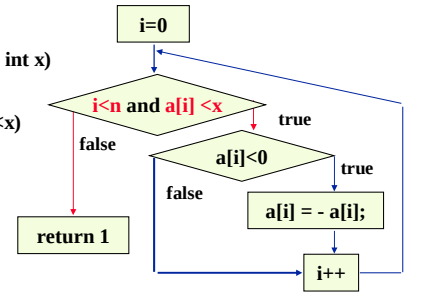


Must add test case (n=1, a[0]=7, x=9) to cover false branch of if

Faults handling positive values of a[i] revealed.
Faults exiting the loop with a[i] <x not revealed

Condition Coverage

```
int select(int[] a, int n, int x)
{
  int i=0;
  while (i<n && a[i] <x)
  {
     if (a[i]<0)
        a[i] = - a[i];
     i++;
  }
  return 1;
}
```



Both i<n and a[i]<x must be false and true for different tests.
Must add tests that cause loop to exit for a value greater than X.
Faults that arise after several loop iterations not revealed.

Structural Coverage Testing

- Adequacy criteria
  - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) coverage
  - Condition coverage
- Attempted compromise between the impossible and the inadequate

Granularity of Tests

- Unit testing
  - Individual components of a program are tested
    - Methods
    - Classes/packages
    - Processes of a distributed system
- Integration testing
  - Test case inputs to subsystem, multiple subsystems, or the whole program, and outputs examined

White/Glass Box vs. Black Box

- Black box
  - depends on spec
  - scales up
    - different techniques at different granularity levels
  - cannot reveal code coverage problems
    - same specification implemented with different modules
- White box
  - depends on control or data flow coverage
  - does not scale up
    - mostly applicable at unit and integration testing level
  - cannot reveal missing path errors
    - part of the specification that is not implemented

Testing Activities

- Test case execution is only a part of the process
- Must also consider
  - Test case generation
  - Test result evaluation
- Planning is essential
  - To achieve early and continuous visibility
  - To choose appropriate techniques at each stage
  - To build a testable product
  - To coordinate complementary analysis and testing

The Testing Environment

- Want to create a scaffold for executing tests
  - Code infrastructure to run tests and check output
- Many benefits
  - Can automate testing process
  - Useful for regression testing
- But, can take some time to implement

Testing Environment Components

- A *user* to generate input for tested component
- An *oracle* for verifying the results are correct

- These two may be combined into a single system

Unit Testing with **Junit**

- Testing environment for writing black-box tests
  - Write special **TestCase** classes to test other classes
  - Several ways to use/set up test cases
- Can be downloaded from
  - http://www.junit.org

Junit Philosophy

- Iterative, incremental process
  - Write small test cases (as needed)
  - Test-as-you-go
    - I.e., after changes, when new method added, when bug identified
- **Junit** test cases must be completely automated
  - No human judgment
  - Easy to run many of them at the same time
- Goal: lots of bang for the buck
  - Even simple tests can find many bugs quickly

Example

```
public class Math {

    private int number1;
    private int number2;

    public Math(int num1, int num2) {
            number1 = num1;
            number2 = num2;
    }
    public int add() {
            int number3 = number1 + number2;
            return number3;
    }
    public int subtract() {
            int number3 = number1 - number2;
            return number3;
    }
    public int multiply() {
            int number3 = number1 * number2;
            return number3;
    }
    public double divide() {
            double number3 = number1 / number2;
            return number3;
    }
    public int mod() {
            int number3 = number1 % number2;
            return number3;
    }
}
```


Example

```
import junit.framework.*;

public class MathTest extends TestCase {
    private Math testVar1;
    private Math testVar2;

    public static void main(String args[]) {

            junit.textui.TestRunner.run(MathTest.
    class);
    }
    protected void setUp() {
            testVar1 = new Math(20, 10);
            testVar2 = new Math(10, 20);

    }
    protected void tearDown() {
            System.out.println("Tear Down");
    }
```

```
public void testadd() {
            // {20 + 10} == {30}
            int expected= 30;
            assertEquals(expected, testVar1.add());
    }
    public void testsubtract() {
            // {20 - 10} == {10}
            int expected= 10;
            assertEquals(expected, testVar1.subtract());
    }
    public void testmultiply() {
            // {20 * 10} == {200}
            int expected= 200;
            assertEquals(expected, testVar1.multiply());
    }
    public void testdivide() {
            // {10 / 20} == {0.5}
            double expected= 0.5;
            assertEquals(new Double(expected), new
    Double(testVar2.divide()));
    }
    public void testmod() {
            // {10 % 20} == {10}
            int expected= 10;
            assertEquals(expected, testVar2.mod());
    }
}
```


To Execute Tests within a Class

- Pick a Test Runner:
  - junit.awtui.TestRunner – *Graphical*
  - junit.swingui.TestRunner – *Graphical*
  - junit.textui.TestRunner – *Textual*
- Invoke on the test case class

```
> java junit.textui.TestRunner ListTest
..
Time: 0.03

OK (2 tests)
```


Junit Components

- Test cases (class **TestCase**)
  - Individual tests
  - Can reuse test case setup (optional)
- Test suites (class **TestSuite**)
  - Test case container
- Test runner (various classes)
  - Executes test suites and presents results

Each Test Has Three Parts

- Code that creates test objects
  - Create a subclass of junit.framework.TestCase
- Code that executes the test
  - Override the method runTest() (which executes the test)
- Code that verifies the result
  - E.g., use junit.framework.assertTrue() to check results (throws exception is test fails)

Setup/Teardown

- Creating objects for each test insufficient
  - Setup overhead grows as number of tests grows
  - Instead, group setup (and teardown) code in one place and reuse
- junit.framework.TestCase.run() executes test case:
  - public void **run**() { setUp(); runTest(); tearDown(); }
    - Do not override this method!
  - Put setup code in setUp() method
  - Put cleanup code in tearDown() method

More Asserts

- Junit has several different tests
  - assertTrue(b) -- asserts that b is true
  - assertFalse(b) -- asserts that b is false
  - assertEquals(o1, o2) -- assert that o1.equals(o2)
  - assertNotNull(o) -- assert o != null
  - assertNull(o) -- assert o == null
  - assertSame(o1, o2) -- assert o1==o2
  - assertNotSame(o1, o2) -- assert o1 != o2

Using a Test Suite

- Test runners will use static suite() method
- **If no suite() method, suite selected automatically**
  - Every method that is **public**, returns **void**, takes no arguments, and begins with "test"
  - This is the way to go – for project 2, use this style
- Then use junit.*.TestRunner TestClass
  - Or use DrJava, Eclipse, etc