Testing

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

- 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
Simple Example

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

HTTP/1.0 200
Content-Type: text/plain

Hello /FooBar/Test

% diff out expectedOutput
```

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

inputs ➔ subcomponent being tested ➔ outputs ➔ Is it correct?

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 **specification** (more later…)
### The Test Case Generation Problem

- How to choose tests that will show that my program does/does not work?
  - Might consider “operational scenarios”
    - What is valid/invalid input?
    - What is the correct action or output?
  - Should 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

```java
// 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 exercise the same logic?
    - Want to avoid redundant tests, to save time
  - Test by common mistake
    - What cases may be tricky to implement?
- At the same time, tests should still be implementation-independent

Test Cases via Boundary Conditions

```java
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
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 can’t know whether
    or not it works
• Compare against correct outputs (properties)
Statement Coverage

```java
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

```java
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

```c
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 classes to test other classes
  - Several ways to use/set up test cases
- Can be downloaded from
  - [http://www.junit.org](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

Each Test Has Three Parts

- Code that creates test objects
- Code that executes the test
- Code that verifies the result
  - E.g., use org.junit.Assert() to check results (throws exception is test fails)
testCase Example with Lists

```java
import static org.junit.Assert.*; import org.junit.*; // other imports
public class ListTest {
    List<Object> l; Object o;
    @Before public void setUpTest() {
        l = new LinkedList<Object>();
        o = new Object();
    }
    @Test public void testAdd() { // name is arbitrary
        l.add(o);
        assertTrue(l.contains(o));
    }
    @Test public void testIsEmpty() {
        assertTrue(l.isEmpty());
    }
}
```

JUnit Components

- Test cases
  - Individual tests (@Test)
  - Can reuse test case setup (@Before, @BeforeClass)
  - Can reuse test case teardown (@After, @AfterClass)
- Test suites (@RunWith(Suite.class))
  - Test case container
- Test runner (org.junit.runner.JUnitCore)
  - Executes test suites and presents results
  - Can also execute tests within Eclipse
Parameterized Tests

```java
@RunWith(Parameterized.class)
public class ParameterizedListTest {
    int param, square;

    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            { 0, 0 }, { 1, 1 }, { 2, 4 }, { 4, 16 },
            { 5, 25 }, { 6, 36 }, { 7, 48 }
        });
    }

    public ParameterizedListTest(int param, int square) {
        this.param = param;
        this.square = square;
    }

    @Test
    public void square() {
        assertEquals(param * param, square);
    }
}
```

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)
  - assertNotEqual(o1, o2) -- assert o1 == o2
  - assertNotNull(o) -- assert o != null
  - assertNull(o) -- assert o == null
  - assertSame(o1, o2) -- assert o1 == o2
  - assertNotSame(o1, o2) -- assert o1 != o2
- And many others
### Manually Constructing a Test Suite

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses(
    {Test1.class, Test2.class}
)

public class AllTests {}
```

> Java org.junit.runner.JUnitCore AllTests

### To Execute Tests within a Class

- **Invoke test runner on the test case class**
  ```
  > java org.junit.runner.JUnitCore ListTest
  JUnit version 4.4
  ...
  Time: 0.012
  OK (2 tests)
  ```
… or run in Eclipse

- Add test case class to your project
- Right click, and do Run As … Junit test.

- Note: need to set up CLASSPATH correctly for the command-line execution of tests. Take a look at link in the class Resources page