Testing and Specifications
9/13/2004

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)

```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

Is it correct?

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

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

• A specification defines the behavior of an abstraction
• This is the contract between user and provider
  – Provider’s code must implement the specification
  – Providers are free to change the implementation
    • So long as the new code meets the specification
  – Users that depend only on specification won’t have trouble
    • Don’t rely on implementation
• Black box testing essentially checks compliance of an implementation with its specification

Good Specifications are Hard to Write

• Very difficult to get people to write specifications
  – Even harder to keep them up to date
• Having specifications in a separate document from code almost guarantees failure
  – Rationale for Javadoc: extract a standalone specification from the code and embedded comments
• Hard to accurately and formally capture all properties of interest
  – Always finding important details not specified

Specifications Help You Write Code

• Lots of subtle algorithms and data structures
  – Internal specs/invariants vital to correct implementation
• Example: Binary Search Tree
  – All nodes reachable from left child have smaller key than current node
  – All nodes reachable from right child have larger key than current node

Specifications Help You Maintain Code

• In the real world, much coding effort goes into modifying previously written code
  – Often originally written by somebody else
  – Perhaps many different people have modified this code
• Documenting and respecting key internal specifications are the way to avoid a mess
• Documenting and respecting key external specifications are the way to avoid having your customers storm the office with torches and pitchforks

Formal vs. Informal Specifications

static int find(int[] d, int x)

• An informal specification
  – If the array d is sorted, and some element of the array d is equal to x, then find() returns the index of x ……
• A formal specification
  – (for all i, 0 ≤ i < d.length, d[i-1] ≤ d[i]) and there exists j, 0 ≤ j < d.length, such that d[j] == x
  implies find(d,x) = j ……
• Note: These specs assume array has no duplicates

Advantages and Disadvantages

• Formal specifications
  – Forces you to be very clear
  – Automated tools can check some specifications
    • Either at compile-time (static checking) or run-time (dynamic checking)
• Informal specifications
  – Some important properties are hard to express formally
    • Sometimes just difficult
    • Sometimes don’t have the necessary formal notation
  – Some people are intimidated by formal specs
Types of External Specifications

- Specifications on methods
  - Pre-conditions/requires: What must be true before call
  - Post-conditions/effects: What is must be true after call
  - Often relates final values to initial values
  // precondition: the array d is sorted
  // postcondition:
  //   returnValue >= 0 && d[returnValue] == x
  // or (returnValue == -1 && x does not occur in d)
  static int find(int d[], int x);

Types of Internal Specifications

- Specifications appearing within code itself
  - i.e., comments
- Loop invariants: condition that must hold at the beginning of each iteration of a loop
  - d[0..i] is sorted
- Data structure or field invariants
  - elementCount <= elementData.length

Behavior vs. Function

- Side effects
  - Writes output to a file
  - Could block on a condition or mutex
- Performance
  - Should you specify performance of operations?
    - As hard as 451: what kind of bound (upper bound, amortized bound, expected bound, ...), order of bound, ...
    - But need at least informal specs
      - Random access is fast, insertion/deletion may be slow

Specifications and Subtyping

- Liskov substitution principle (original? formal stmt)
  - If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.
  - I.e, if anyone expecting a T can be given an S, then S is a subtype of T.
  - If we override a method, how do the specifications of the original and new method relate?

Specifications and Subtyping (cont’d)

// precondition: the array d is sorted
// postcondition:
//   returnValue >= 0 && d[returnValue] == x
// or (returnValue == -1 && x does not occur in d)
static int find(int d[], int x);

- If we override this method, can the new method
  - Have true as a precondition?
  - Have precondition “d is sorted and exists i s.t. d[i] == x”?
  - Have postcondition “returnValues==1 or returnValue is first index such that d[returnValue] == x”?
  - Throw NoSuchElementException rather than returning -1 when x does not occur in d?

What Makes a Good Specification?

- Sufficiently restrictive
  - Forbids unacceptable implementations
- Sufficiently general
  - Allows all acceptable implementations
- Clear
  - Easy to understand
  - A little redundancy may help (some people disagree)
Javadoc

- Integrates documentation into source code as comments
- Will generate an external specification

```java
/** Javadoc Comment for this class */
public class MyClass {
  /** Javadoc Comment for field text */
  String text;
  /** Javadoc Comment for method setText */
  @param t Javadoc comment for parameter t
  public void setText(String t) {...}
}
```

Javadoc example

```java
/** Given a sorted array, returns the index into the array of the given element, otherwise returning -1. */
@param d array to search in, assumed sorted
@param x the element to search for
@return i >= 0 when d[i] == x, and -1 when x does not occur in d */
public static int find(int d[], int x) {
  ...
}
```

Javadoc example: HTML

```html
Method Detail

public boolean contains(int[] a, int x);

• Special tags for classes
  - @author
  - @version

• Special tags for methods
  - @param
  - @return
  - @exception

• Reference to another element
  - @see
```

A Few Javadoc Tags

```
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

```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 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<a & 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<a & 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 **TestCase** 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 black-box 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

TestCase Example with Lists

```java
import junit.framework.*;
import java.util.*;
public class ListTest extends TestCase {
    public void testAdd() {
        LinkedList l = new LinkedList();
        Object o = new Object();
        l.add(o);
        assertTrue(l.contains(o));
    }
    public void testIsEmpty() {
        LinkedList l = new LinkedList();
        assertTrue(l.isEmpty());
    }
}
```

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 if 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

TestCase Example, again

```java
public class ListTest extends TestCase {
  private Object o;
  public void setUp() { o = new Object(); }
  public void testAdd() {
    LinkedList l = new LinkedList();
    l.add(o);
    assertTrue(l.contains(o));
  }
  public void testPushPop() {
    LinkedList l = new LinkedList();
    Object o2;
    l.addFirst(o);
    o2 = l.removeFirst();
    assertTrue(o == o2);
    assertTrue(l.size() == 0);
  }
}
```

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

Manually Constructing a Test Suite

```java
public class ListTest extends TestCase {
  ...
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new ListTest());
    suite.addTest(new ListTest("testAdd");
    return suite;
  }
}
```

Manually Constructing a Suite (cont’d)

- You can also create test suites more easily:
  ```java
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new ListTest("testAdd");
    suite.addTest(new ListTest("testPushPop");
    return suite;
  }
  ```
- Or simply:
  ```java
  public static Test suite() {
    return new TestSuite(ListTest.class);
  }
  ```
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 DoJava, Eclipse, etc