Specifications and Testing

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

Administrivia

• Reading: Liskov chapter 9, 10
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
```
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?
    ⬅️

oracle
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 …

Software Specifications

- A software specification defines the behavior of an abstraction
- This is the contract between user and provider
  - A provider’s code must implement the specification
  - Providers are free to change the implementation, so long as the new code still meets the specification
  - Users that depend on the implementation rather than the specification could be in trouble
- Black box testing essentially checks compliance of an implementation with its specification
Good specifications are hard and rare

- 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

- For many subtle and interesting algorithms and data structures, having internal specifications/invariants about the algorithm and data structure are vital to getting the code right
  - e.g., in a binary search tree, all nodes reachable from the left branch have a smaller key than the current node, and all nodes reachable from the right branch have a larger key than the 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 six different people have modified this code
- Documenting and respecting key internal specifications are the way to avoid a bloody 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

```java
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 ……
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 we 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

    static int find(int d[], int x);
    // precondition: the array d is sorted
    // postcondition:
    //   returnValue >= 0 && d[returnValue] == x
    //   or (returnValue == -1 && x does not occur in d)
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
  – \( \text{elementCount} \leq \text{elementData.length} \)

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 can be slow
**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) {...}
}
```
/** 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
   @returns i >= 0 when d[i] == x, and -1 when x does not occur in d

*/
public static int find(int d[], int x) {
    ...
}
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

```java
interface List {
    // ... Insert 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?

```java
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)

![Diagram of white/glass box testing](image)
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

Statement 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;
}
```

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

Faults handling positive values of a[i] not revealed
**Branch 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;
}
```

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 of a[i] greater than X.
Faults that arise after several loop iterations not revealed.
**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

**Granularity of Tests**

- **Unit testing**
  - Individual components of a program are tested
    - Methods
    - Classes/packages
    - Processes of a distributed system

- **Integration testing**
  - Test cases are input to subsystem, multiple subsystems, or the whole program, and outputs examined
Alternatives to Testing

• Assume program works
  – You’re almost certainly wrong

• Formally prove program works
  – Tedious, really hard to get right

• Prove that your program works in a limited sense
  – E.g., type checking

You can’t always get what you want

• Correctness properties are undecidable
  – the halting problem can be embedded in almost every property of interest
The Halting Problem

- **Halt(P) = true if and only if program P halts**

- **Halt() does not exist**
  - **Informal proof:**
    - Suppose Halt() exists
    - Consider the following method
      ```java
      void foo() { while (Halt("foo()")) ; }
      ```
    - Then foo() halts iff Halt("foo()") returns false
    - Halt("foo()") is false iff foo() does not halt
      - Contradiction, so Halt() does not exist

Examples of Undecidable Properties

- Does P terminate (halt)?
- Does x=0?
- Is f() ever called?
- Which foo method is invoked by b.foo()?
- and so on...
How Can We Check Anything?

• Halting problem occurs when languages are too expressive
  – So reduce set of valid programs

• Example: Type checking
  ```java
class Foo { void f() { ... } }
Object o = new Foo();
o.g(); // not allowed, no g() method
o.f(); // not allowed, even though "works"
(Foo) o.f(); // ok; worked around type sys
```

Example: Unmatched Lock Operations

```java
if ( ... ) {
  ... synchronized(S) {

    ... lock(S);
  }
... }
```

Checking for match undecidable ...
so Java prescribes a more restrictive, but statically checkable construct.
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

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 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 unit 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 on-line 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
Write tests by extending the JUnit TestCase class
- Each class you define contains one or more tests

Each test implemented in a single method, having three conceptual parts
- Code that (1) sets up and (2) executes the test
  - Essentially arbitrary Java code
- Code that (3) verifies the result
  - Uses JUnit’s assertTrue() and related methods. With throw an exception on failure

```
public class ListTest extends TestCase {
    public void testAdd() {
        LinkedList l = new LinkedList();
        Object o = new Object();
        l.add(o);
        assertTrue(l.contains(o));
    }
}
```
Setup/Teardown

- Creating objects for each test may be 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(); run tests; tearDown(); }
  - 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);
    }
}
```

Create objects at outset
Perform test/check result
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 that o1 is not equal to o2
  - `assertNull(o)` -- assert o != null
  - `assertNotNull(o)` -- assert o == null
  - `assertSame(o1, o2)` -- assert o1 == o2
  - `assertNotSame(o1, o2)` -- assert o1 != o2

Test Suites

- Groups together multiple tests for execution
- Suite is defined by the `TestCase.suite()` method
  - Lots of ways to implement this method, but here is the simplest (the default):
    ```java
    public static Test suite() {
        return new TestSuite(this.getClass());
    }
    ```
  - Gathers together all public methods that take no arguments and whose name begins with the prefix “test”
Test Runner

• To execute test suite, run one of the following:
  – For graphical display
    • junit.awtui.TestRunner TestClass or
    • junit.swingui.TestRunner TestClass
  – For textual display
    • junit.textui.TestRunner TestClass
• Or run from within your own code:
  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }

Summary: JUnit Components

• Test cases (class TestCase)
  – Individual tests; extend TestCase class
  – Can define per-class setUp and tearDown methods
• Test suites (class TestSuite)
  – Test case container
  – Default container contains all methods which take no arguments and return void with name test*
• Test runner (various classes)
  – Executes test suites and presents results
**IDEs**

- **IDE:** Interactive Development Environment
  - **Editor**
    - Usually with some nice syntax coloring, indentation features
  - **Compiler**
    - Errors sorted, displayed nicely; easy to see corresponding code
  - **Debugger**
    - Closely watch/change execution of source code
  - **Etc...**
    - Testing, search, code transformations, ...

- **Examples:** DrJava, NetBeans, Eclipse, Visual Studio, emacs

**Dr. Java**

- Light-weight IDE
- Editing
  - Syntax coloring, auto-indent, brace matching
- Testing
  - Integrates with Junit testing framework
    - Uses suite() or auto-generated suite
  - Interaction panel allows interactive method invocations
- Debugging
  - Integrates with Java debugger
  - Interactions panel also useful
Debugging

• My program doesn’t work: why?
• Use the scientific method:
  – Study the data
    • Some tests work, some don’t
  – Hypothesize what could be wrong
  – Run experiments to check your hypotheses
    • Testing!
  – Iterate

Starting to Debug

• What are the symptoms of the misbehavior?
  – Input/output
  – Stack trace (from thrown exception)
• Where did the program fail?
• What could have led to this failure?
• Test possible causes, narrow down the problem
Checking that Properties Hold

- Print statements
  - Check whether values are correct
    - E.g., look at value of i to check if i > 0
  - Check whether control-flow is correct
    - E.g., see if f() is called after g()

- Automatic debugger
  - Allows you to step through the program interactively
  - Verify expected properties
    - Don’t need to put in print statements and recompile
  - Use as part of testing

Dr. Java Interactions Pane

- Can evaluate Java expressions interactively
  - Can bind variables, execute expressions/statements

- Benefits
  - Make sure that methods work as expected
  - Test invariants by constructing expressions not in program text
  - Combines with interactive debugger
Dr. Java’s Automatic Debugger

- Set execution breakpoints
- Step through execution
  - into, over, and out of method calls
- Examine the stack
- Examine variable contents
- Set watchpoints
  - Notified when variable contents change

Using the Debugger

- Set debug mode to on
  - Turns on debug panel with state information
- Set break point(s) in Java source
- Run the program
Tips

• Make bug reproducible
  – If it’s not reproducible, what does that imply?
• Reduce to smallest program that reproduces bug
  – Reveals the core problem
• Explain problem to someone else
  – Explaining may reveal the flaw in your logic
• Keep notes: don’t make the same mistake twice

Defensive Programming

• Assume that other methods/classes are broken
  – They will mis-use your interface
    public Vector(int initialCapacity, int capacityIncrement) {
      super();
      if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
    ...
  
• Goal: Identify errors as soon as possible
Avoiding Errors

- Codify your assumptions
  - Include checks when entering/exiting functions, iterating on loops
- Test as you go
  - Using Junit
  - Using the on-line debugger
- When you fix a bug
  - Retest: be sure you didn’t introduce a new bug
  - Write a new test that would catch this bug to add to your suite
- Do not ignore possible error states
  - Deal with exceptions appropriately