Software Testing

Testing techniques can be classified along several orthogonal dimensions.

White Box (or Glass Box) Versus Black Box Testing

White box testing relies on knowing the internals of the implementation. For instance, testing by inserting print statements or something similar to get a sense of what gets executed is a form of white box testing. This is useful when you are debugging a particular implementation of an ADT, but it of course specific to that particular implementation.

In contrast, black box testing treats the implementation as hidden, and can only test the code based on its interface and its specification. (It treats the ADT as a black box—you do not get to peek inside.) The advantage of black box testing is that it can be used to test *any* implementation of the ADT.

A black box testing infrastructure for an ADT *must be able to test any implementation of the ADT, and any correct implementation must pass the test.*

Unit Versus Integration Testing

A class or a closely related set of classes (module) is often implemented by a single programmer. The class or module all by itself is not runnable, and generally will not do anything useful on its own. It is used within the context of a larger piece of software. A key feature of the ADT approach we advocate in this course is that it allows the class to be implemented without regard to the context in which it will be used in.

(This is in fact the whole point of specification-based design in the wider context of engineering. By designing a piece according to a specification, other pieces needing to interact with the piece need only assume that the piece satisfies its specification. The actual details of how the specification is met—the implementation—does not matter.)

Unit testing is the process of taking a class implementation and making sure it satisfies its specification. This requires making up test data and a test program to test the implementation against the specification.

In contrast, integration testing tests the class in a context with other classes to make sure that the higher-level interaction between the classes works correctly.

In this course, because of our ADT design philosophy, we will concentrate on black box unit testing.

Yet another sort of testing that is sometimes mentioned is *regression testing*. Roughly speaking, regression testing is the process of keeping old tests around; because adding a new feature to a piece of software should not change existing behaviors, we can run the old tests to ensure that the code still behaves as expected.

Purpose of Testing

The point of testing is to *find bugs*, and not to prove programs correct. Why?

To prove that a program is correct using tests, we need to exercise all the inputs to the program. This can be a problem if there many inputs to test. For instance, a program that takes two 64-bit integers as input, that is, 128 bits of input total, would require 10^{22} years to run through tests for all inputs, even if we allow a millon tests to be performed every second. That's clearly infeasible. Things get worse, of course, because some programs can take one of infinitely many inputs—for instance, programs manipulating lists.

So we cannot in general exercise all inputs, and therefore need to focus only on a few. This leaves the possibility of bugs for the other inputs.

Consider the well-publicized story of the Intel FDIV bug. In 1994, Intel recalled its Pentium processors to repair a bug in its floating point division instruction, FDIV. Intel has estimated that this recall cost the company 475 million dollars.

Here is an example of the bug:

> 4195835.0/3145727.0 = 1.333 820 449 136 241 000 (Correct value)
>
> 4195835.0/3145727.0 = 1.333 739 068 902 037 589 (Flawed Pentium)

Intel estimated that the result of the FDIV instruction was incorrect a little more often than once in every 9 billion floating point divisions. An IBM study found, however, that the values that occur most often in spreadsheet and scientific computations are more likely to trigger the bug.

After the Intel debacle, AMD, Intel's chief competitor, was worried about their own chip. They turned to the automatic theorem proving community, and Moore, a leading researcher in the area, proved with his team that the algorithm used to implement the FDIV instruction in AMD's chip was correct using a mechanical theorem prover. They in fact managed to prove the correctness of the entire floating-point kernel (not just the FDIV instruction). In the process, they found and repaired two design errors that had not been caught by any of the 80 million separate tests that had been run.

Unfortunately, proving programs correct is quite hard, so testing is often advocated as a first approach to debugging. We will return to proving programs correct towards the end of the course, but for the time being, let us focus on testing.

Test Coverage

Given that we can only test a finite (and in fact small) number of cases, how do we choose those tests?

A good test is a test that finds bugs. Finding good tests therefore requires a basic understanding of the common bugs that can occur. This depends on the actual programming language used, and on the kind of program being tested. Many books have been written on common bugs in various programming languages.

In general, test coverage should include:

- Trivial cases (e.g., empty list inputs for list-processing functions)
- Typical cases (both easy and hard)
- Boundary cases (cases that are either just acceptable, or just outside acceptable; e.g., inputs that access an array close to its bounds)
- Weird cases
- Error cases (if the specification mentions how error cases are treated)

What should be tested during a test? Every equation in the specification should be tested. Remember that some methods in Java have an implicit specification. In particular, `equals()` methods are required to be reflexive, symmetric, and transitive; the `hashCode()` method is required to have the property that `a.equals(b)` implies `a.hashCode()==b.hashCode()`. Those should be tested as well.

Black Box Unit Testing Infrastructure

Let us look at an example of a black box unit testing class for an integer stack signature and specification, given as follows:

```
public static IntStack emptyStack ();
public static IntStack push (IntStack, int);
public boolean isEmpty ();
public int top ();
public IntStack pop ();
public boolean equals (Object);

IntStack.push(s,i).top() == i
IntStack.push(s,i).pop() == s
IntStack.push(s,i).isEmpty() == false
IntStack.emptyStack().isEmpty() == true
```

```
IntStack.emptyStack().equals (obj) ==
  obj.isEmpty()  if obj is a IntStack
  false          otherwise
IntStack.push(s,i).equals (obj) ==
  false          if obj is a IntStack & obj.isEmpty()==true
  obj.top()==i && s.equals(obj.pop())
                 if obj is a IntStack & obj.isEmpty()==false
  false          otherwise
```

Assume we have a class `IntStack` implementing the above interface, to be tested.

The general structure of the tester is to first generate a set of instances of `IntStack`, and test each instance. The instances, at least for the typical cases, can be usefully generated at random.

This example is a variant of the `ClassroomTester` code given to you in the first homework. First off, the class definition, and the `main` method, which invokes the `testIntStacks()` method that tests a set of random objects, and the texttttestExceptions() method that tests the exceptions that the class is required to throw under some conditions. Useful statistics are accumulated in private fields.

```
import java.util.Random;

public class IntStackTester {

    private int ntests = 0;      // Total Tests Run
    private int nerr = 0;        // Number of Errors
    private int totStacks = 0;   // Number of Stacks Created

    /* Main testing method */
    public static void main(String[] args){
        /* Create a new Tester, run the tests, and Print out results. */
        IntStackTester t = new IntStackTester ();

        /* See definitions below */
        t.testIntStacks();
        t.testExceptions();

        /* A few Stats... */
        System.out.println("\nNumber of stacks tested: " + t.totStacks);
        System.out.println("Number of tests performed: " + t.ntests);
        System.out.println("Number of errors found: " + t.nerr);
    }
```

The `testIntStacks()` method first tests the empty stack (an interesting special case), and repeatedly generates `NUM_TO_TEST` piece of data used to construct an instance `IntStack`, and invokes `testIntStack` to create and test each instance.

```
/* How many stacks to create? */
private static final int NUM_TO_TEST = 200;

/* Create random stacks, and test them */
private void testIntStacks(){
    Random r = new Random();
    // first off, make sure we test the empty stack
    testIntStack (new int[0],r.nextInt(10));
    // bunch of random stacks (random size)
    for(int i = 0; i < NUM_TO_TEST; i++){
        int size = r.nextInt(5); // number of elements in stack
        int[] vals = new int[size];
        for (int j=0; j < size; j++)
            vals[j]=r.nextInt(10);
        /* Run the Tests */
        testIntStack (vals,r.nextInt(10));
    }
}
```

The method `testIntStack()` creates a new instance of `IntStack` based on the inputs to the method, and tests each specification equation. Each specification is tested via the `assertTrue()` method described below.

```
/* Test a single Stack */
private void testIntStack (int[] vals,int next){
    try{
        /* Create the stack from the array of ints */
        IntStack s = IntStack.emptyStack ();
        for (int i=0; i<vals.length; i++)
            s = IntStack.push (s,vals[i]);

        /* Update the numbers for our static method checks later*/
        totStacks++;

        /* check accessors */
        assertTrue(IntStack.push(s,next).top() == next,
                   "IntStack.push(s,i).top()==i");
        assertTrue(IntStack.push(s,next).pop() == s,
                   "IntStack.push(s,i).pop()==s");
```

```
            assertTrue(IntStack.push(s,next).isEmpty() == false,
                       "IntStack.push(s,i).isEmpty()==false");

            /* for the empty stack, make sure isEmpty is correct */
            if (vals.length == 0)
                assertTrue (s.isEmpty() == true,
                            "IntStack.emptyStack.isEmpty()==true");

            /*
              Here, you want to test the .equals() method.
              some hints:
              - you definitely want to test that .equals(null) is false.
              - you may want to construct an equal object and an
                unequal object to make sure that .equals() returns the
                right result for those
              - you may want to check reflexivity, symmetry, transitivity.
                (the first two are easy, the third is a bit of a pain).
            */

        } catch(RuntimeException e) {

            /* If there was an exception anywhere in there, then we
             *   have a problem */
            assertTrue(false, "Exception: "+e.getMessage());
        }
    }
```

Note that exceptions are caught and reported as failed tests.

The method `testExceptions` tests that expections are appropriately reported. For integer stacks, the only exceptions that must be reported occur when attempting to take the top element of an empty stack, or attempting to pop an empty stack.

```
    /* Make sure exceptions are thrown for border cases  */
    private void testExceptions() {
        /* top/pop of empty stack */
        try {
            int result = IntStack.emptyStack().top();
            assertTrue (false, "IntStack.emptyStack().top():  No Exception");
        } catch(RuntimeException e) {
            assertTrue (true, "");
        }
```

```
        try {
            IntStack s = IntStack.emptyStack().pop();
            assertTrue (false, "IntStack.emptyStack().pop(): No Exception");
        } catch(RuntimeException e) {
            assertTrue (true, "");
        }

    }
```

Method `assertTrue()` reports when a test succeeded or failed, and updating the corresponding statistics.

```
    /* Number of dots to print before we go to the next line */
    private static final int DOTS_PER_LINE = 50;

    /* Update the test counters based on the result given. Result
     *   is expected to be true for passing tests, and false for
     *   failing tests.  If a test fails, we print out the provided
     *   message so the user can see what might have gone wrong.
     * Be sure to review anything that doesn't make sense. */
    private void assertTrue (boolean result, String msg){
        ntests++;
        if(!result){
            System.out.println("\n**ERROR**: test# " + ntests + " -- " + msg);
            nerr ++;
        }
        if(ntests % DOTS_PER_LINE == 0)System.out.println();
        System.out.print(".");
    }
}
```