6 Software Testing

We saw last time a classification of errors that can occur in programs. As I stated, logical errors — programs not behaving as expected — are the most difficult to catch and correct, if only because languages do not offer any facilities for identifying those errors. They rely on the notion of "expected program behavior", which is really only in the head of the designer or the programmer.

As far as ADTs are concerned, we know what behavior to expect: it's exactly the behavior described by the specification. So how do we check that a program satisfies its specification (i.e., behaves as expected)? There are two main ways: formal verification, and software testing.

6.1 Testing versus Formal Verification

Formal verification, what you learned in CSU 290, is the process of verifying using logic or other means that a program satisfies its specification. As you will remember from CSU 290, formal verification is usually difficult. The purpose of formal verification is proving a program correct (i.e., satisfies its specification) for all possible inputs. The point of testing, on the other hand, 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, as I said, proving programs correct is quite hard, so testing is often advocated as a first approach to debugging. We may return to proving programs correct towards the end of the course, but for the time being, we focus on testing.

6.2 White-Box (or Glass-Box) Testing

Testing techniques can be classified along several orthogonal dimensions.

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.

A particularly useful technique for white-box testing is to use assertions. Let's take assertions in Java as an example. An *assertion* takes the following form:

`assert` *BooleanExpression*;

or

`assert` *BooleanExpression* : *ValueExpression*;

When Java encounters an `assert` during execution, it does one of two things, depending on whether or not assertion checking is turned on or off. (It is off by default.)

- If assertion checking is turned on, it will evaluate the *BooleanExpression* in the assertion; if it evaluates to true, then execution proceeds to whatever statement follows the assertion; if it evaluates to false, then an exception `AssertError` is thrown (with a value given by *ValueExpression*, if one is provided).
- If assertion checking is turned off, then the assertion is skipped altogether.

Thus, assertions provide a way to put in "sanity checks" within your own code, that you can enable by running the code with assertion checking turned on, with the property that if

you run the code with assertion checking turned off, it behaves just like you never had put in any assertions in the first place. (There is no runtime penalty, either.)

To execute a Java program with assertion checking turned on, use:

```
java -ea Program
```

(assuming `Program.class` is the compiled class containing the `main` method).

Generally speaking, assertions are meant to check invariants that *must* be true in order for the code to work correctly. This means, in particular, that whenever possible, you should have assertions on when you ship your code out. (There are ways to make sure that Java or whatever language you use executes your code with assertions turned on. Please refer to the documentation.) Many programmers do not enable assertions on production code, for fear generally of the run-time cost of checking all those assertions. Unless your assertions are very heavy, however, I would recomment you keep assertions on. The benefit of having this extra layer of checks for the important invariants your code usually outweighs the minor inconvenience of a slightly slower execution. (And, let's face it, if you are programming in Java, you are alreadu more or less admitting that executing programs as fast as possible is not your priority.)

6.3 Black-Box Testing

In contrast to white-box testing, 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.

The testing you performed in CSU 211, using test cases, is a form of black-box testing.

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.*

6.4 Unit Versus Integration Testing

A different axis of comparison for testing approaches, orthogonal to the white-box/black-box axis, is to consider whether we are testing pieces of the program in isolation or as a whole. Let me try to make that a bit more precise.

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.

6.5 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)[1]
- 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()`

[1] In some cases, boundary cases may depend on the specifics of an implementation of an ADT; this sort of boundary case has a flavor of white-box testing.

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.

6.6 Example: Black-Box Unit Testing Infrastructure for ADTs

Let us look at an example of a black-box unit testing class for the Point ADT in your first homework. Recall the signature:

```
public static Point create (int,int);

public int xPos ();
public int yPos ();
public Point move (int,int);
public Point add (Point);
public Point scale (float);
public boolean equals (Object);
public int hashCode ();
public String toString ();
```

and the specification:

```
create(x,y).xPos() = x
create(x,y).yPos() = y

create(x,y).move(dx,dy) = create(x+dx,y+dy)
create(x,y).add(p) = create(x+p.xPos(),y+p.yPos())

create(x,y).scale(f) = create(Math.round(x*f), Math.round(y*f))

create(x,y).equals(obj)
   = (x==obj.xPos() & y==obj.yPos())  if obj is a Point
   = false                            otherwise

create(x,y).hashCode() = (x+y)*(x+y+1)/2 + y

create(x,y).toString() = "(" + x + "," + y + ")"
```

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

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

First off, the class definition and `main` method, which invokes the `testRandom()` method that tests a set of random objects. Useful statistics are returned by the `testRandom()` method.

```
import java.util.Random;

public class PointTester {

   public static void main(String[] args){
      int totalInstances;
      int totalErrors;

      totalInstances = 100;

      totalErrors = testRandom(totalInstances);

      totalErrors += testExceptions();

      /* A few Stats... */

      System.out.println("\nNote that this tester does not test .equals()");
      System.out.println("Number of points tested: " + totalInstances);
      System.out.println("Number of errors found: " + totalErrors);
   }
```

The `testRandom()` method repeatedly generates an instance of `Point` using each of the creators and tests that instance. We assume a `testSingleRandom`*`creator`*`()` method, one per creator. We also test various cases of interest, here borderline cases.

```
   public static int testRandom (int num) {
      int count = num;
      int errors = 0;
      while (count > 0) {
         errors = errors + testSingleRandomCreate();
         count = count - 1;
      }

      // borderline cases
      Point zero = Point.create(0,0);
      errors = errors + testCreate (zero,0,0);

      return errors;
   }
```

The method `testSingleRandomCreate()` creates a new instance of `Point` using the `create` creator, based on the inputs to the method, and invokes `testCreate()` to test each specification equation. Note that we pass to `testCreate()` not only the newly created instance, but also the data that was used to create the instance. Helper functions to generate random integers and floating point numbers are provided to simplify testing.

```
public static int randomInt (int max) {
   return new Random().nextInt(max)+1;
}

public static float randomFloat (int max) {
   return new Random().nextFloat() * max;
}

public static int testSingleRandomCreate () {
   int x = randomInt(1000)−500;
   int y = randomInt(1000)−500;
   Point p = Point.create(x,y);
   return testCreate(p,x,y);
}

public static boolean equalPoints (Point c1, Point c2) {
   return (c1.xPos()==c2.xPos() && c1.yPos()==c2.yPos());
}

public static int testCreate (Point p, int x, int y) {
   int errors = 0;
   String front = "Point.create("+x+","+y+").";

   try{
      // Point.create(x,y).xPos() = x
      if (!(assertT (p.xPos()==x, front+"xPos()"))) errors++;

      // Point.create(x,y).yPos() = y
      if (!(assertT (p.yPos()==y, front+"yPos()"))) errors++;

      // Point.create(x,y).move(dx,dy) = Point.create(x+dx,y+dy)
      int dx = randomInt(1000)−500;
      int dy = randomInt(1000)−500;
      if (!(assertT(equalPoints(p.move(dx,dy),Point.create(x+dx,y+dy)),
                front+"move("+dx+","+dy+")"))) errors++;
```

```
        // Point.create(x,y).add(c) = Point.create(x+c.xPos(),y+c.yPos())
        dx = randomInt(1000)−500;
        dy = randomInt(1000)−500;
        Point dp = Point.create(dx,dy);
        if (!(assertT(equalPoints(p.add(dp),Point.create(x+dx,y+dy)),
                    front+"add(Point.create"+dp.toString()+")"))) errors++;

        // Point.create(x,y).scale(f) =
        //    Point.create(Math.round(x∗f),Math.round(y∗f))
        float f = randomFloat(10);
        if (!(assertT(equalPoints(p.scale(f),
                                Point.create(Math.round(x∗f),
                                        Math.round(y∗f))),
                    front+"scale("+f+")")))
            errors++;


        // should test equals here [ future homework! ]


        // Point.create(x,y).hashCode() = (x+y)∗(x+y+1)/2 + y
        if (!(assertT(p.hashCode()==(x+y)∗(x+y+1)/2 + y,
                    front+"hashCode()")))
            errors++;

        if (!(assertT(p.toString().equals("(" + x + "," + y + ")"),
                    front+"toString()"))) errors++;
        return errors;

    } catch (RuntimeException e) {

        // If there was an exception anywhere in there, then we
        //   have a problem
        assertT(false, "Exception: "+e.getMessage());
        return errors+1;
    }
}
```

Note that exceptions are caught and reported as failed tests.

Method `testExceptions()` is used to test whether the exceptions prescribed by the specification are thrown for suitable arguments. For the Point ADT, because no exceptions are

specified, this function is pretty simple,

```
    private static int testExceptions () {
        return 0;
    }
```

Finally, the testing code relies on a method `assertT()` that report whether a test succeeded or failed.

```
    /*   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.
    /* returns true if the assertion succeeded, and false otherwise! */

    private static boolean assertT (boolean result, String msg) {
        if (!result) {
            System.out.println("\n**ERROR**: "+ msg);
        }
        System.out.print(".");
        return result;
    }
}
```