Testing a Class

Purpose:
The purpose of this lab is to acquaint you with what’s involved in correctly testing a class.

Testing:
Once a class has been designed and implemented, the implementation must be tested to insure that it conforms to specifications. In the current case, we limit our considerations to testing a single instance of a class. As in previous labs, you will be presented with objects to exercise in isolation. Testing a system consisting of many interacting objects presents considerably more difficulties.

An implementation that behaves as specified is termed correct. It should be obvious that in order to determine if an object is correct, we need to know precisely how the object is supposed to behave in all cases. This is given by the object’s specifications. Without a clear and complete specification, it is impossible to determine if an implementation is correct. As you go through the exercises, evaluate the specifications you are given. Are they clear and precise? Are they complete?

A second observation is that while it is possible to determine that an implementation is not correct by testing, it is generally not possible to verify that an implementation is correct by testing. Most objects have (at least, theoretically) infinitely many possible states. For an object to be correct, it must behave properly in every state. No finite amount of testing can test all possible cases. On the other hand, there need be only one case in which an object does not function properly for the implementation to be incorrect. If the implementation is not correct, it is possible to find a faulty case by testing the object.

Suppose, for instance, one of the functions of an object is to compute square roots. You can have the object compute 100 or 1000 square roots, all correctly, without being certain that it will compute every square root correctly. However, if you get one wrong answer, you know that the object is not correct.

Nevertheless, testing is an important phase in the software development cycle. The more carefully we test a system, the more confident we are of its correctness.

Test plans:
To adequately test a system, the test must be carried out in an organized way. Specifically, we first develop a test plan to direct the test. The test plan clearly identifies what aspects of the system are to be tested, how the test is to be conducted, and what results are expected.

While it’s a bit premature to present a technical discussion of test plan adequacy, we can give a few informal guidelines.

First, a test should cover “all possibilities,” or at least as many as practical. Consider for instance, a “converter” object that converts centimeters, meters, or kilometers to inches, feet, yards, or miles, and vice versa. A thorough test should involve converting centimeters to inches,
centimeters to feet, centimeters to yards, centimeters to miles, meters to inches, etc., and
converting inches to centimeters, inches to meters, etc. At the very least, the test should convert
from and to each of the seven units.

Second, the test should specifically test “special cases” and “boundary conditions.” “Boundary
conditions” are the extreme values for which the system is expected to work. For example, a
system that reads an input file should be tested with an empty file. A system expected to handle
input lines up to 132 characters in length should be tested with input lines that are 132 characters
long. An object that computes the area of a geometrical figure should be tested with a figure
whose area is 0.

Finally, the test should consider various sequences of operations. For instance, if an object
responds to commands “resize” and “change color,” the test should include “resize” followed by
“resize,” “resize” followed by “change color,” etc.

Exactly what constitutes a test plan depends, of course, on the system being tested. Basically, it is
a document or set of documents that includes a description of what is being tested (correct output
of legal operations, specific boundary conditions, aspects of system robustness, etc.), the actions
that comprise the test (input data, queries, commands, etc.), and the anticipated results.

**Developing a test plan:**

In developing the test plan for an object, we must consider several cases.

1. Test that each command and query performs as specified. In designing these cases, each com-
mand is considered independently. The command is issued, several consecutive times if
appropriate, and the object is queried to determine its state.

   When testing features independently, there are as many cases as commands and queries. One
special case to consider is the state of the object at start of execution. If queries given at start
of execution give improper results, the fault could be the result of an improperly initialized
constructed object, or an incorrectly implemented query. Testing the query in other object
states should identify the source of the problem.

2. Test sequential combinations of commands. These tests are carried out once we are convinced
that each command independently performs as specified.

Having identified the cases to consider, we produce a document (test file), or set of documents,
 describing each test. The documents detail the case to be considered, the sequences of commands
to be performed, and the expected results.

In general, a test plan involves many separate tests. In our example, since we are testing a very
simple object, we describe a single test. Such a test might be one small part of a system test.

The test files mentioned above are often written so that they can be used directly as input to the
system. In our case, since we are testing the objects manually, we use them as instructions for
conducting the test. The test proceeds exactly as stipulated in the test plan, and actual results are
checked against expected results. We report the test case description, the actual result, and the
conclusion.
To summarize, we must have a clear idea of the goal and expected results of a test. To be successful in testing means to uncover bugs. Thus the proper attitude to have in test case design is one of trying to make the system behave in an unexpected way, in a way contrary to the system’s specifications.

**Example:**

Here is part of a test plan for testing the class *RetailItem*, seen in a previous lab. The specifications can be found in [http://www.cs.uno.edu/~labCourse/Labs/Lab8/docs/](http://www.cs.uno.edu/~labCourse/Labs/Lab8/docs/).

In this example, we test only the “number of units on hand” property of a *RetailItem*. This involves the methods `available`, `onHand`, `restock`, and `sell`. We assume that we have an interface that allows us to create *RetailItem* instances, and directly invoke the relevant methods.

**Purpose:**

Test the “number of units on hand” property of a *regsales8.RetailItem*. Specifically, test the `available`, `onHand`, `restock`, and `sell` methods.

**Test class to execute:**

    regsales8.RetailItemOnHandTest

**Class specification:**

Given in [http://www.cs.uno.edu/~labCourse/Labs/Lab8/docs/](http://www.cs.uno.edu/~labCourse/Labs/Lab8/docs/).

**Test Plan:**

<table>
<thead>
<tr>
<th>Action</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>start system</td>
<td></td>
</tr>
<tr>
<td>create <em>RetailItem</em> with 10 units on hand</td>
<td></td>
</tr>
</tbody>
</table>

*Verify initial state:*

<table>
<thead>
<tr>
<th>Method</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>onHand()</code></td>
<td>10</td>
</tr>
<tr>
<td><code>available(10)</code></td>
<td>true</td>
</tr>
<tr>
<td><code>available(0)</code></td>
<td>true</td>
</tr>
<tr>
<td><code>available(5)</code></td>
<td>true</td>
</tr>
<tr>
<td><code>available(11)</code></td>
<td>false</td>
</tr>
</tbody>
</table>

*Test sell():*

<table>
<thead>
<tr>
<th>Method</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sell()</code></td>
<td></td>
</tr>
<tr>
<td><code>onHand()</code></td>
<td>9</td>
</tr>
<tr>
<td><code>sell()</code></td>
<td></td>
</tr>
</tbody>
</table>
sell()
onHand() 7

*Test sell(int):*
sell(2)
onHand() 5
sell(3)
onHand() 2
sell(0)
onHand() 2

*Test boundary:*
sell()
onHand() 1
available(1) true
sell()
onHand() 0
available(1) false
available(0) true
sell(0)
onHand() 0
available(1) false
available(0) true

*Test restock:*
restock(5)
onHand() 5
available(1) true
restock(1)
onHand() 6
restock(0)
onHand() 6
Test sequences:
sell()
restock(2)
onHand() 7
restock(2)
sell()
onHand() 8
sell(6)
restock(2)
onHand() 4
restock(4)
sell(8)
onHand() 0

Test constructor boundary:
create RetailItem, 0 on hand
onhand() 0
available(1) false
available(0) true

End test.

Note: you may wonder why we do not, as part of the test, invoke sell() when than the number of units on hand is 0. If you look at the specification of the sell() method, you will note that the specification requires that the method only be invoked when onHand() is greater than 1. That is, it is not the responsibility of the sell() method to make sure that there is a unit to sell; it is the responsibility of the client. In a complete system, it would be the responsibility of the interface object for example to make sure that it doesn’t invoke sell() when onHand() is 0. Invoking sell() when onHand() is 0 would be attempting to test how an “erroneous program” behaves. And an erroneous program is by definition unpredictable. This will become cleared when we consider “programming by contract” later.

Exercises:
As directed by your instructor, develop a test plan (or partial test plan), conduct a test, and determine whether or not the implementation is correct for several of the following classes:

• the class Circle written previously;
• the class Rectangle written previously;
• the class Lamp written previously;
• the class calculator9.LoanCalculator;
• the class regsales9.Register.

To test the first three classes, use the interfaces provided in previous labs. Specifically, you should have directories named figures5 and lamps5 in your Java directory. These should contain implementations and interfaces for the classes Circle, Rectangle, and Lamp.

To test the last two classes:

• create subdirectories named calculator9 and regsales9 in your Java directory;
• copy class files from ~labCourse/Labs/Lab9/calculator9/ and ~labCourse/Labs/Lab9/regsales9/ into these directories;

To use the loan calculator, enter principal (in dollars), interest rate (as a percent), and the length of the loan (in years) in the text fields. Then press the large button labeled “Calculate.”

Requirements are the principal and loan rate must be greater than or equal to 0, and the loan length must be greater than 0. All values must be numeric, but need not be integers.

It might help you to design the test for the loan calculator to use a web-based loan calculator (which we assume to be accurate) to generate test data “expected” results. Here are two such sites:

• UNO Federal Credit Union Loan Calculator;
• NetQuarters Inc. Loan Calculator.

The interface used to test the register is the same as you have seen in a previous lab. You may assume that the interface and the class RetailItem are implemented correctly. Specifications for the class Register can be found here.

Post lab:

As directed by your instructor, submit a test plan and test results for one of the first three objects that you have implemented yourself, and for one from the last two.