Invasive Testing of Java™ Classes

K.Rangaraajan
Man Machine Systems
mms@giasmd01.vsnl.net.in

Unit testing is an important element of testing software. In object-oriented software development, this term commonly refers to testing an individual class. The term *integration testing* denotes testing a group of classes, and the term *system testing* refers to testing the application as a whole. In the simplest and most widely followed testing practice, developers of classes are expected to perform unit and integration testing of their respective classes before releasing their library to other groups. The test team is responsible for system testing before releasing the application to clients.

There is on-going research in the area of testing object-oriented software, and a lot more needs to be done. Since class testing is the starting point of testing object-oriented software, understanding the issues involved and devising a way to perform effective class testing is a crucial first step in delivering high quality software. At Man Machine Systems, we have developed a new model for testing Java classes, called *invasive testing*. We believe this model can make class testing more interesting and effective. Testing is typically characterized as a *destructive* process, so the term *effective* suggests that the tester* must be able to uncover a *maximum* number of bugs with a *minimum* of effort. In this article, we describe our model with examples and compare it with conventional unit testing techniques.

Invasive and *Intrusive* Testing of a Class

We define ***invasive*** testing as a technique that uses the normally inaccessible elements of a class to increase the effectiveness of testing. When the class under test is modified to support testing, we call this ***intrusive*** testing.

If a class *C* comprises a set *M* of methods that belong in the class interface, intrusive testing typically assumes the existence of *N* additional methods and modifications to some or all of the *M* methods to aid testing. The additional *N* methods are not intended to be used by normal clients of the class (and are not part of the published class interface), but are defined purely to support testing. If methods in sets *M* or *N* access private members of the class, then the technique is invasive as well. Plain invasive testing does not modify the class under test, but would use private details of the class that normal users cannot access.

Let us first review conventional class testing strategies before investigating invasive testing.

Conventional Class Testing Strategies

1. Tester as client, version - 1 (non-invasive, non-intrusive):

This is the most common strategy for testing classes. The tester writes a driver in the same language as the class under test, and exercises the published methods of the class, following

* The term *tester* as used here denotes a role rather than an individual. This means that a developer could be a tester.

a set of test criteria, such as coverage. Since the driver code resides in a different class than the tested one, accessibility rules defined in the language apply. Consider the stack implementation ***MyStack*** shown in Figure 1 as an example of a Java class to be tested.

```
class MyStack {
   private Object[] elems;
   private int top, max;
   public MyStack(int sz) {
      max = sz;
      elems = new Object[sz];
   }
   public void push(Object obj) throws Exception {
      if( top < max )
         elems[top++] = obj;
      else throw new Exception("Stack overflow");
   }
   public Object pop() throws Exception {
      if( top > 0 )
         return elems[--top];
      throw new Exception("Stack underflow");
   }
   public boolean isFull() {
      return top == max;
   }
   public boolean isEmpty() {
      return top == 0;
   }
}
```

Figure 1: MyStack Class

The test driver shown in Figure 2 tests the above stack class.

```
class StackTester {
   public static void main(String[] args) {
      MyStack s1 = new MyStack (10); // Max of 10 elements
      if( !s1.isEmpty() )
         Test.error("Stack is not empty initially!");
      StackTester obj1 = new StackTester(); // An object to stack
      s1.push(obj1);
      if( s1.isEmpty() )
         Test.error("Stack is empty after a push!");
      StackTester obj2 = (StackTester) s1.pop();
      if( obj1 != obj2 )
         Test.error("Problem in push()/pop()!"); // ----- (A)
```

```
        // ... Other code
    }
}
```

Figure 2: A Typical Driver to Test the MyStack class

We assume the existence of test support routines such as *Test.error(String)*. The test driver is written to test invocations of *MyStack*'s public methods in some combination.

Benefits

1) The technique is intuitive since it simulates actual usage of the class under test.
2) The class tested is the class actually released to customer since no modification is performed to the class for testing purposes.

Limitations

1) Since the tester has to exercise public methods in the class interface, certain methods can only be tested in combinations, not individually. For example, the push (or pop) method cannot be tested in isolation. As a result, if the push/pop combination fails, it could be due to a bug in either or both the methods and additional tests may be needed before the actual bug may be isolated. Even if the push/pop combination matches, there is no guarantee that they are both correct; it is conceivable that both have bugs that coincidentally produce the correct behavior. Once again, additional combinations will have to be tried.
2) It is not always possible to exercise all parts of the implementation adequately by invoking public methods alone.
3) There is no way to test interfaces either.

2. Tester as client, version – 2 (non-invasive, intrusive):

In this scenario, the class to be tested is modified to support testing. Accessible ***getter*** and ***setter*** methods are added for all state variables that are otherwise inaccessible. Alternatively, all elements of the class are made ***public*** to grant unrestricted access. This permits the tester to overcome the limitations of approach (1) above. The ***MyStack*** class of Figure 1, according to this strategy, might be modified as depicted in Figure 3:

```
class MyStack {
    // Private elements are public - FOR TESTING ONLY
    public Object[] elems;
    // Private elements are public - FOR TESTING ONLY
    public int top, max;
    public MyStack(int sz) {
        max = sz;
        elems = new Object[sz];
    }
    public void push(Object obj) throws Exception {
        if( top < max )
            elems[top++] = obj;
        else throw new Exception("Stack overflow");
```

```
   }
   public Object pop() throws Exception {
      if( top > 0 )
         return elems[--top];
      throw new Exception("Stack underflow");
   }
   public boolean isFull() {
      return top == max;
   }
   public boolean isEmpty() {
      return top == 0;
   }
}
```

Figure 3: Modified MyStack Class

The test driver is now rewritten to take advantage of private details of the class to perform more effective testing. Figure 4 shows how this can be done.

```
class StackTester {
   public static void main(String[] args) {
      MyStack s1 = new MyStack (10); // Max of 10 elements
      if( !s1.isEmpty() )
         Test.error( "Stack is not empty initially!");
      StackTester obj1 = new StackTester(); // An object to stack
      s1.push(obj1);
      Test.assert( (s1.top == 1) && (s1.elems[s1.top-1] == obj1),
                                                "Push failed!");
      StackTester obj2 = (StackTester) s1.pop();
      Test.assert( (s1.top == 0) && (obj2 == obj1), "Pop failed!");
      // ... Other code
   }
}
```

Figure 4: Stack Tester Driver

Benefits

The semantics of individual methods can be verified. This can reduce the test effort, since the number of methods is much less than the number of method combinations.

Limitations

1) The class tested is different than the one released to a client.
2) The tester needs to understand the class implementation details, in addition to its interface. For complex classes, this could be quite difficult.

3) If the class implementation changes in future, the driver code will have to be appropriately modified.

4) The source code for the class is required.

3. Automatically instrumenting the code with assertions (invasive, intrusive):

This is a very promising approach for testing object-oriented software. In this case, the developer embeds assertions (class invariant, method pre- and post conditions) inside comments in the source code. A preprocessor parses these assertions and emits the modified source where the assertions have been appropriately moved to method bodies. The preprocessed code is compiled using a standard Java compiler. The resulting application is then run as if it were the original program. Any violation of contracts will be reported by the assertions layer. Figure 5 below gives an example of how assertions might be specified.

```
/**
* @inv (top >= 0) && (top < max)
*/
class MyStack {
   private Object[] elems;
   private int top, max;

   /**
   * @post (max == sz) && (top == 0)
   */
   public MyStack(int sz) {
      max = sz;
      elems = new Object[sz];
   }
   /**
   *  @pre top < max;
   *  @post (top == top$prev + 1) && (elems[top-1] == obj)
   */
   public void push(Object obj) throws Exception {
      if( top < max )
         elems[top++] = obj;
      else throw new Exception("Stack overflow");
   }
   /**
   * <assertions here>
   */
   public Object pop() throws Exception {
      if( top > 0 )
         return elems[--top];
      throw new Exception("Stack underflow");
   }
```

```
    public boolean isFull() {
        return top == max;
    }
    public boolean isEmpty() {
        return top == 0;
    }
}
```

Figure 5: MyStack Class with Assertions

The assertion language depends on the tool used; currently there is no standard. The example above uses a syntax supported by our tool JMSAssert™ that generates a JMScript™ test script file (discussed later). Reto Cramer's *iContract* is another interesting tool in the public domain that supports assertions in Java classes [1].

Benefits

1) Reasonably complex class invariants, pre- and post conditions may be specified.
2) Since the assertions are defined as part of class/method comments, the source can be recompiled to get uninstrumented classes.
3) This approach assists in testing the complete application.

Limitations

1) The class tested is different than the one released to a client.
2) Source code has to be available to instrument with assertions.
3) The assertion language is proprietary, so moving to another preprocessing tool is difficult.
4) To test classes individually (not as a sealed off application), a test driver still has to be written.
5) Individual methods cannot be tested because of access restrictions.

4. Tester is the developer (invasive, intrusive):

In this case, the test driver resides within the class being tested. Typically, this is the ***main*** method or a ***test*** method that is static. When it is invoked, the test logic is exercised. Since the method is part of the class, access restrictions do not apply; all elements of the class can be accessed.

```
class MyStack {
    // ... Insert stack related elements here.
    // ...
    // This is the stack test driver. It is part of the class
    // itself.
    public static void main(String[] args) {
        MyStack s1 = new MyStack (10); // Max of 10 elements
        if( !s1.isEmpty() )
            Test.error( "Stack is not empty initially!");
        MyStack obj1 = new MyStack (); // An object to stack
```

```
        s1.push(obj1);
        Test.assert( (s1.top == 1) && (s1.elems[s1.top-1] == obj1),
                                                    "Push failed!");
        MyStack obj2 = (MyStack) s1.pop();
        Test.assert( (s1.top == 0) && (obj2 == obj1), "Pop failed!");
        // ... Other code
    }
}
```

Figure 6: Test Driver in MyStack

Benefits

1) There is no need to convert *private* elements to *public* for the sake of testing.
2) The test driver for the class is self-contained, so the driver is more likely to be in sync with the class.

Limitations

1) The test driver cannot be reused when the class evolves. For each version of the class, corresponding version of the driver inside the class (perhaps copying and pasting from previous version) must be developed. To facilitate regression testing, it is better to have the driver separated from the class.
2) The source code has to be available.
3) When the class is delivered to a client, the test driver constitutes *excess baggage*. However, if the driver is removed from the code, then two version of the class have to be maintained, one with the driver and another without the driver.

Invasive Testing

It is a widely acknowledged fact that testing object-oriented software is more difficult than testing nonobject-oriented software. Payne et. al.[2] point out that *"from the value of testing perspective, information hiding reduces the ability for faults to propagate to an observable output and hence reduces the likelihood that faults will be revealed during testing"*.

The invasive model that is proposed allows the tester to tunnel through the information hiding barrier and write a driver that accesses the normally inaccessible elements of a class. This can occur without making changes to the class source. In fact, the approach does not require Java source code to be available for testing purposes. To assist in achieving this, a scripting language has been developed, called JMScript™ that is layered on Java. The language allows a tester to write a script that instantiates Java objects and send messages to them. This is very similar to what happens in the Java language itself. Like many other scripting languages, it is weakly and dynamically typed. These types are associated with values of objects and not with variables. Once a Java object is instantiated in the script, any element of the object, be it private or protected, can be accessed without restriction.

The following JMScript driver illustrates one way of testing the *MyStack* class.

```
// MyStackTest.jms
main() {
   // Stack size
   sz = 1;
   // Create a stack instance.
   s = new MyStack(sz);
   assert(s.top == 0 && s.max == sz);
   println("Stack instantiation is OK");
   // Push just one object - a stack object
   obj1 = new MyStack(1);
   s.push(obj1);
   assert(s.top == 1 && s.elems[s.top-1] == obj1);
   println("Stack.push works OK");
   // Pop and check
   obj2 = s.pop();
   assert(s.top == 0 && obj2 == obj1);
   println("Stack.pop works OK");
   // Another pop - must result in underflow exception
   try {
     s.pop();
     println("Empty pop fails to throw exception!");
   }
   catch(Exception e) {
     println("Empty pop throws exception as expected");
   }
   // Simulate stack full condition
   s.top = s.max;
   // Push, now, must result in overflow exception
   try {
     s.push(obj1);
     println("Push on full fails to throw exception!");
   }
   catch(Exception e) {
     println("Push on full throws exception as expected");
   }

   // Similarly, test other methods individually
   // ...
   println("*** Stack test completed - relax now!!");
}
```

Figure 7 Testing MyStack Using JMScript

The script can be executed by running the command line version of the interpreter (or the more sophisticated ***JVerify***™ environment can be used, that includes the interpreter and much more).

The script accesses even the private elements of a class, bypassing Java's access control mechanism. Notice how assertions are used after every method call to check the post-condition. However, explicitly checking the post-conditions can litter the script with too many assertions. There is a more convenient way to do this, as shown in the following test script.

```
// MyStackTest2.jms
main() {
   sz = 1;
   // Create a stack instance.
   s = new MyStack(sz);
   s.push( new Object() );
   // ... Other code not shown
}

// Class invariant trigger
invariant(meth, s){
    trace("meth=>" + meth);
    assert(s.top >= 0 && s.top <= s.max &&
                                          s.elems != null);
    trace("Invariant checked inside:" + (meth.equals("<init>") ?
                                                 "Constructor": meth));
}
// Automatically called by the interpreter on exit from
// Stack constructor.
postCtor(method, s, sz) {
   assert(s.top == 0 && s.max == sz && s.elems != null);
   trace("postCtor called");
}
prePush(meth, s, obj) {
   assert( s.isFull() == false && obj != null );
   trace("prePush called");
}

// This is the static block - meant for one-time execution
static {
   // Denote assertion functions
   asserts =
   {{"","INVARIANT", "MyStackTest2.invariant"},
    {"<init>(I)V", "POSTCONDITION", "MyStackTest2.postCtor"},
```

```
    {"push(Ljava/lang/Object;)V", "PRECONDITION",
                                  "MyStackTest2.prePush"}
   };
   // Install assertion functions
   setClassTrigger("MyStack", asserts);
}
```

Figure 8 JMScript Using Assertions

This example illustrates that a tester can identify an invariant for a class, a set of pre- and post conditions for methods and register those with the interpreter. When a method is invoked from the test script, the corresponding precondition, postcondition and invariant routines are automatically and appropriately executed by the interpreter. The sequence followed by the interpreter is:

- Check class invariant
- Check precondition for the method
- Invoke method
- Check Post condition for the method
- Check class invariant

There are exceptions to this rule, but that is beyond the scope of this article.

It is possible to do better than setting up triggers manually as outlined above. If the class source is available and assertions are embedded in comments as shown in Figure 5, another utility called ***JMSAssert***TM will parse these and automatically generate a script file containing appropriate triggers.

Table 1 gives a comparison of the different techniques considered so far.

Technique	Invasive	Intrusive	Source Code Needed
1) Tester as a client, version 1	No	No	No
2) Tester as a client, version 2	No	Yes	Yes
3) Automatic instrumentation with assertions	Yes	Yes	Yes
4) Tester as a developer	Yes	Yes	Yes
5) Invasive testing	Yes	No	No

Table 1. Comparing the Techniques

Testing State Machines

The invasive model provides a convenient approach to testing state machines. In general, testing a state machine tends to be fairly involved because of the potentially infinite number

of states possible. The ability to access private details could considerably reduce the effort (the downside is that intimate knowledge of the class may be required).

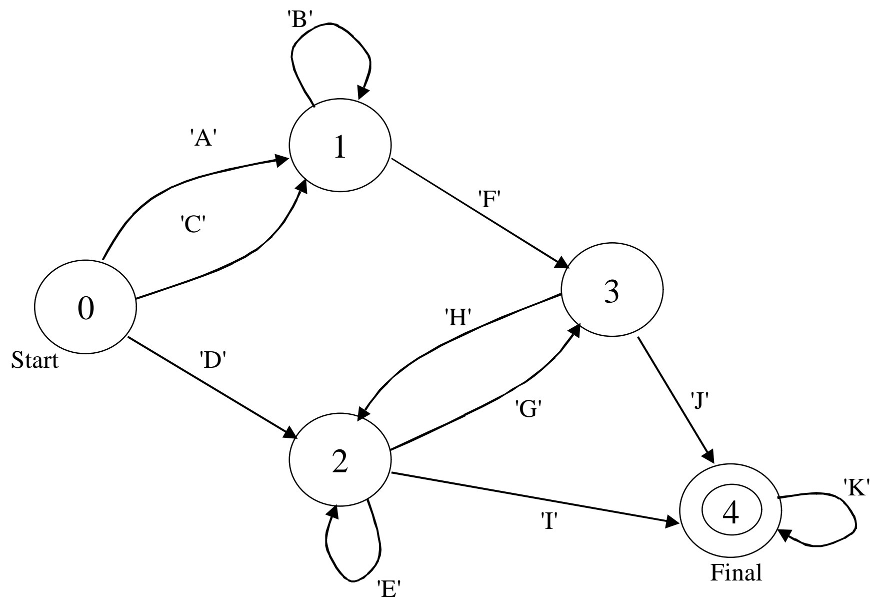

Figure 9: State Machine

Consider the state machine shown in Figure 9. The corresponding Java source is given in Figure 10 (to emphasize clarity, several optimizations have been left out).

```
public class FSM {
   private FSMState start, one, two, three, four;
   private FSMState current;
   public FSM() {
      one = new FSMStateOne();
      two = new FSMStateTwo();
      start = new FSMStateStart();
      three = new FSMStateThree();
      four = new FSMStateFour();
      reset();
   }
   public void reset() {
      current = start;
   }
   public void transition(char c) {
      if(current != null)
         current = current.transition(c);
   }
```

```
public final boolean done() {
   return (current != null) && current.isFinal();
}
private static abstract class FSMState {
   abstract FSMState transition(char c);
   boolean isFinal() {
      return false;
   }
}
private class FSMStateStart extends FSMState {
   FSMState transition(char c) {
      switch (c) {
         case 'A':
         case 'C': return one;
         case 'D': return two;
         default: return null;
      }
   }
}
private class FSMStateOne extends FSMState {
   FSMState transition(char c) {
      switch (c) {
         case 'B': return this;
         case 'F': return three;
         default: return null;
      }
   }
}
private class FSMStateTwo extends FSMState {
   FSMState transition(char c) {
      switch (c) {
         case 'E': return this;
         case 'G': return three;
         case 'I': return four;
         default: return null;
      }
   }
}
private class FSMStateThree extends FSMState {
   FSMState transition(char c) {
      switch (c) {
         case 'H': return two;
```

```
            case 'J': return four;
            default: return null;
         }
      }
   }

   private class FSMStateFour extends FSMState {
      boolean isFinal() {
         return true;
      }
      FSMState transition(char c) {
         switch (c) {
            case 'K': return this;
            default: return null;
         }
      }
   }
}
```

Figure 10. A State Machine in Java

How can we prove that the Java program of Figure 10 accepts *all* the strings as per the state machine specification of Figure 9? Clearly, testing with *all* strings is impossible. Here is how the state machine can be tested using JMScript:

```
// Testing the FSM involves testing each state transition.
// This requires migrating the FSM to the appropriate state
// and then invoking the transition function.
// File: fsmtest.jms
main() {
   createFSM();
   checkStateStart();
   checkStateOne();
   checkStateTwo();
   checkStateThree();
   checkStateFour();
   System.out.println("*** FSM Testing Done!! ***");
}
createFSM() {
   this.fsm = new FSM();
   assertState("start");
}
checkStateStart() {
   setState("start"); // Start at "start"
```

```
   transition('A');
   assertState("one"); // Must be in “one”
   setState("start"); // Reset to “start”
   transition('C');
   assertState("one"); // Must be in “one”
   setState("start"); // Reset to “start”
   transition('D');
   assertState("two"); // Must be in “two”
}
checkStateOne() {
   setState("one");
   transition('B');
   assertState("one");
   setState("one");
   transition('F');
   assertState("three");
}
checkStateTwo() {
   setState("two");
   transition('E');
   assertState("two");
   setState("two");
   transition('G');
   assertState("three");
   setState("two");
   transition('I');
   assertState("four");
}
checkStateThree() {
   setState("three");
   transition('J');
   assertState("four");
   setState("three");
   transition('H');
   assertState("two");
}
checkStateFour() {
   setState("four");
   transition('K');
   assertState("four");
}
```

```
// Utility routines
assertState(state) {
   if (this.fsm.current != getField(this.fsm, state))
      throw new Exception("Bad FSM State!");
}
setState(state) {
   this.fsm.current = getField(this.fsm, state);
}
transition(ch) {
   this.fsm.transition(ch);
}
exceptionHandler(meth, excp) {
   if(meth != "assertState") {
      println("Exception: " + excp + " occurred in: " + meth);
   }
}
```

Figure 11. Testing FSM Using JMScript

The basis for the testing strategy used above, is to drive the state machine to each intermediate state not allowed normally, and then to ensure that when an input is received, it transitions correctly to the next state.

Benefits

1) The class under test is not modified to support testing. The class tested is the same as the class released to a client.
2) Source code for the class does not need to be available. (This means it is possible to test third party libraries.)
3) The test driver accesses implementation details, if necessary, to test individual methods, not combinations. This can minimize testing effort.
4) Class invariant, method pre- and post conditions can be complex in real situations. The support for such arbitrarily complex conditions is taken care of by providing procedures in JMScript.
5) Since the test driver is separate from the class being tested, reuse of the test cases is possible throughout regression testing.

Limitations

1) The test script is written in a language different than Java. This means learning a new language.
2) Assertions are checked only for Java methods executed directly from JMScript. If a Java method ***m1*** internally calls another method ***m2***, automatic checking of invariant, pre- and post conditions is performed for ***m2***.
3) This is ideally suited to perform unit testing of individual classes and integration testing of small groups of classes, but not for system testing.

4) This relies heavily on the commitment of the development environment to the usage of Design by Contract™.

JVerify™ Test Environment

To create, execute and test JMScript scripts, we have developed JVerify, a GUI-based Windows™ application. The environment supports a debugger that allows setting breakpoints, and examining the execution snapshot through an object browser, and call stack. Another salient feature of the environment is the facility to understand a class' interface from its ***.class*** file. This can be quite useful when the source for the class under test is not available.

Conclusion

Testing object-oriented software is more difficult because of information hiding. Invasive testing, proposed in this article, makes a tester "all powerful" by allowing access to private details of the class under test. This model renders a class more amenable to testing (even if it has not been designed for testability). This model is also particularly useful for testing state machines. A scripting language called JMScript, layered on Java, has been designed to support class invasion in the context of Java programs. More information about the language and associated tools is available at www.mmsindia.com.

References

1. Reto Cramer, iContract – The Java™ Design by Contract™ Tool, www.promigos.ch/kramer/

2. Jeffery E.Payne, Roger T.Alexander, and Charles D.Hutchinson, Design-for-Testability For Object-Oriented Software, Object Magazine, July 1997, pp. 35 – 43.