Unit and Component Testing Using Design Patterns

Keum-Young Sung
Handong Global University, Pohang, South Korea
kysung@handong.edu

Abstract

The suggested technique in this study shows a new approach to software testing that can be used as a basis for testing a variety of software structure, especially using design pattern. With this study, such design patterns as observer pattern, command pattern, bridge pattern, and chain of responsibility pattern have been used to show that a unit and component testing may be performed with some guidelines. For this study a way of using JUnit to perform an independent path testing as an example has also been introduced. Using the suggested technique, especially based on design patterns, various forms of software structure can be tested in some guided way.

1. Introduction

White box testing and black box testing are fundamental testing methods along with a unit and a component testing. White box testing or transparent box testing is to test internal workings and implementation detail while black box testing does not consider internal program logic [1][3].

1.1. A Black Box Testing with JUnit

With black box testing internal implementation is not considered, but an interface to the target component is used [2]. JUnit is a Java object for a unit testing framework used with Java programming language [4]. The following code for a binary tree searching [5] is a good example to show a test case for a black box testing.

```
public class binTree {
 public static void search ( int key, int [] elemArray, Result r )
  {int bottom = 0 ;
   int top = elemArray.length - 1 ;
   int mid ; r.found = false ;
   r.index = -1 ;
  while ( bottom <= top )
  {
    mid = (top + bottom) / 2 ;
    if (elemArray [mid] == key)
{
   r.index = mid ;  r.found = true ;
   return ;  } // if part
  else
  {  if (elemArray [mid] < key)
      bottom = mid + 1 ;
     else top = mid - 1 ;
   }
  } //while loop
 } // search
}
```

The following is the Result class which is one of the passed parameter to the binary tree class. The Result class is used for recording the result of a searching with the binary tree.

```
public class Result {
 int index; boolean found;
 Result(){ index = 0;found = false;}
 Result(int i, boolean b) {index = i; found = b;};
}
```

An example black box test case with the JUnit is as follows, in which the input data using a target array and a key to be searched is given, and the expected output is given with an assertion:

```
import org.junit.Test;
public class testCaseOutOfRange {
   @Test
   public void test() {
   int[] eleArray = {1,2,3,4,5};
   int key = 6;
   Result r = new Result(0, false);
   binTree.search(key, eleArray, r);
   assertSame("Same key Value", r.found, false );
  }
}
```

Because a key value which is not in the given array is given to the search tree in the above JUnit code, the JUnit predicate, “assertSame,” asserts that the Boolean variable, found, should be false. The resulting screen shot of the Eclipse JUnit running is in Figure 1, which shows a green line in the box of running status,

which indicates that the predicate, *assertSame("Same key Value",r.found, false)*, is true.

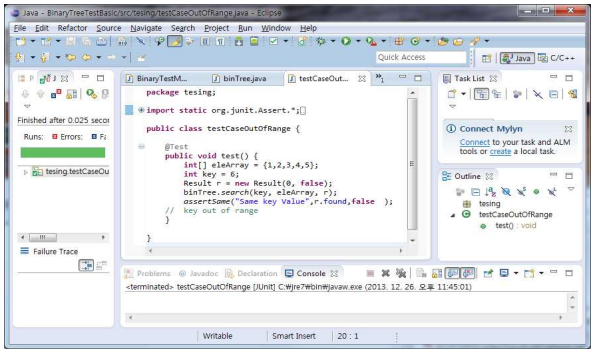


Figure 1. A Sample Session With JUnit With an Invalid Key

The following JUnit code is for searching a key which is the first element of the given array.

```
public class testCaseB {
 @Test
 public void testSearch() {
          int[] eleArray = {1,2,3,4,5};
          int key = 1;
          Result r = new Result(0, false);
          binTree.search(key, eleArray, r);
          assertSame("Same key Value",r.found,true  );
          assertSame("Key Value",r.index,0 );
  }
}
```

The two predicates in the above code say that the key, 1, is matched at the index number of 0 of the array, and the resulting running session is given in Figure 2:

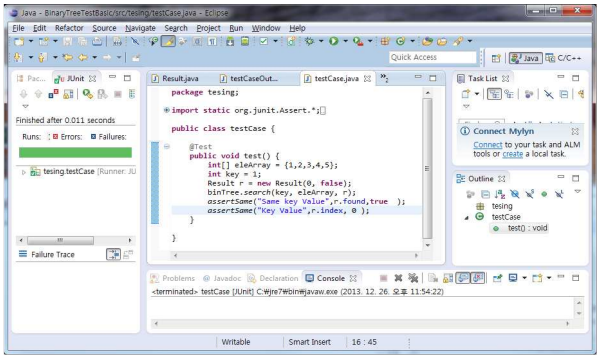


Figure 2. JUnit Session With the Key Index of 0

The green line tells that the two predicates are proved to be true.

For black box testing, boundary values are used to minimize the number of test inputs, and maximize the effect of testing. In the given example array, the boundary values for a key include 0, 1, 3, 5, and 6. A test input with a mid key value, 3, and its corresponding running output are as the following:

```
public class testCaseD {
          @Test
public void testSearch() {
          int[] eleArray = {1,2,3,4,5};
          int key = 3;
          Result r = new Result(0, false);
          binTree.search(key, eleArray, r);
          assertSame("Same key Value",r.found,true  );
          assertSame("Key Value",r.index,2 );
     }
}
```

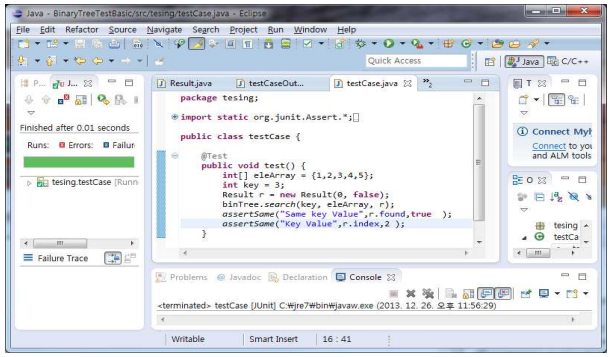


Figure 3. Eclipse JUnit Session With a Mid-Key

The predicates in JUnit source code indicate that the key is to be found, and that the array index of key value is 2.

All the given example JUnit codes show that only the interface of the component tested is used.

1.2. White Box Testing with Junit

The flow graph of the given example binary tree in Figure 4 is well illustrated in [5]:

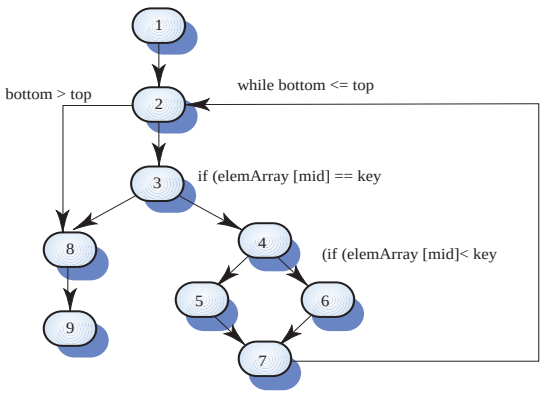


Figure 4. Independt Basis Paths

Based on the cyclomatic complexity [6], the number of paths is the number of decision plus 1, the independent paths of the above flow graph are [5]:

1, 2, 3, 8, 9: path 1
1, 2, 3, 4, 6, 7, 2: path 2
1, 2, 3, 4, 5, 7, 2: path 3
1, 2, 3, 4, 6, 7, 2, 8, 9: path 4

For white box testing based on independent path testing, each path can be tested by adjusting the values of variables that are in the target path. For example, to test path 2 the setting of the variables are as follows:

bottom <= top
elmArray[mid] != key
eleArray[mid] < key'
expected output: control back to node 2

The essence of independent path testing is to minimize the number of paths for testing while all the statements should be tested at least once.

2. Unit and Component Testing With Junit

2.1. Testing a Single Component with Junit

The suggested technique with JUnit for testing a single component is to make the same condition as the one shown in the original basis path condition. The condition for the path 1is as given below:

bottom <= top
elmArray[mid] = key
expected output: true, index value

The suggested technique in this study makes use of a loop count variable and a result object, in which a loop variable is for tracking how the program control flows, and a resulting object expresses the searching result with a Boolean value. The running session of Eclipse JUnit for path 1 is given in Figure 5, and the JUnit code is as below:

```
@Test
public void IndependentPathTesting () {
int[] eleArray = {1,2,3,4,5};
int key = 3;
Result r = new Result(0, false);
binTree.search(key, eleArray, r);
assertSame("Same key Value",r.found, true  );
}
```

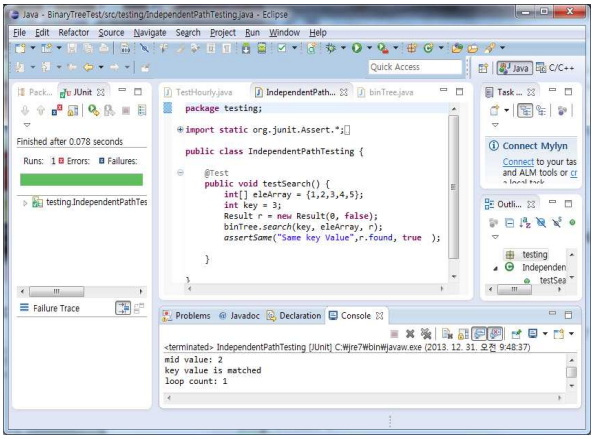


Figure 5. A White Box Testing For Searching a Mid-Index Key

The output window of the above figure shows the mid-value of the array index, status of key matching, and the number of loop repetition as follows:

mid value: 2
key value is matched
loop count: 1

The output of JUnit reflects exactly the condition of the independent path testing for path 1, which was given as below.

bottom <= top
elmArray[mid] = key
expected output: true, index value

The testing for paths 2 and 3 is prepared with the same procedure, which also makes use of a loop variable to track program control flow, and uses the object of Result class. The JUnit code for testing paths 2 and 3 is as below, and the running session is in Figure 6.

```
public class IndependentPathTesting {
@Test
  public void testSearch() {
  int[] eleArray = {1,2,3,4,5,6,7,8,9};
  int key = 3;
  Result r = new Result(0, false);
  binTree.search(key, eleArray, r);
  assertSame("Same key Value", r.found, true  );
}
```

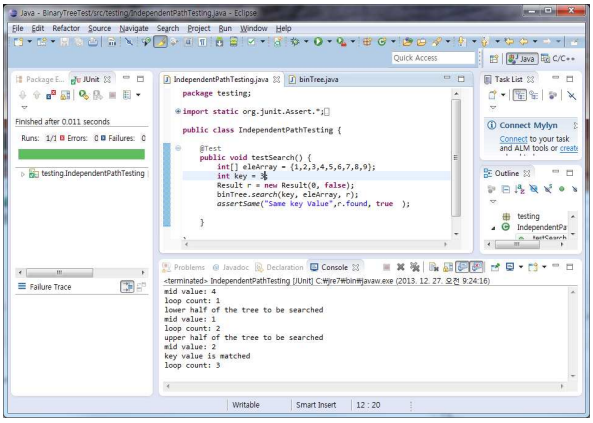


Figure 6. A White Box Testing For Paths 2 and 3

The output shows the progressive change of array mid value and the number of loop repetition count as below:

mid value: 4
loop count: 1
lower half of the tree to be searched
mid value: 1
loop count: 2
upper half of the tree to be searched
mid value: 2
key value is matched
loop count: 3

The above running result reflects the testing condition of paths 2 and 3 which is as follows:

bottom <= top
elmArray[mid] != key
eleArray[mid] < key or eleArray[mid] > key
expected output: control back to while condition statement

2.2. Composite Component Testing and Chain of Responsibility Pattern

When we test a composite component which has the form of the chain of responsibility pattern, generally the interface testing is used. It is not easy to systematically prepare the test case for interface testing especially for a composite component because there is a complex data flow among related sub-components.

In this study, a Java package containing several classes combined with super- and sub-class relationship. The example package has the following inheritance hierarchy as shown with StarUML design tool [7].

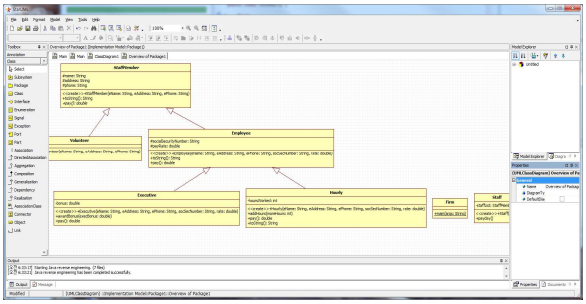
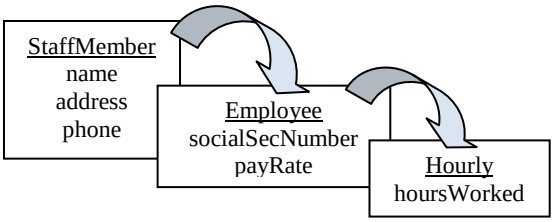


Figure 7. Class Hierarchy Shown With StarUML

Figure 7 shows an inheritance relationship in which there is an inheritance or a chain of responsibility line from StaffMember to Employee class, and from Employee to Hourly class. Suggested in this study is the comparison between superclass object and subclass object in terms of inherited attribute values. The following picture shows the attributes inherited from super to sub class, which is indicated by curved arrows.



The following assertions are to test if an inherited attribute from the superclass has been used properly by a subclass.

assertSame("test SuperClass, StaffMember-attribute",
staffObj.name, "John");
assertSame("test hourlyObj attribute inherited",
hourlyObj.name, "John");
assertEquals(employeeObj.payRate,
hourlyObj.payRate, 0.0);

The "name" attribute belongs to the "StaffMember" class, and is being used by the subclass, Hourly. the "payRate" method of Employee class is inherited to "Hourly" class.

assertSame("test SuperClass, Employee-attribute",
hourlyObj.socialSecurityNumber, "234-232-3345");

In the above predicate, the "socialSecurityNumber" attribute has been inherited from Employee class to Hourly class. Figure 8 shows the running session of for this example test case.

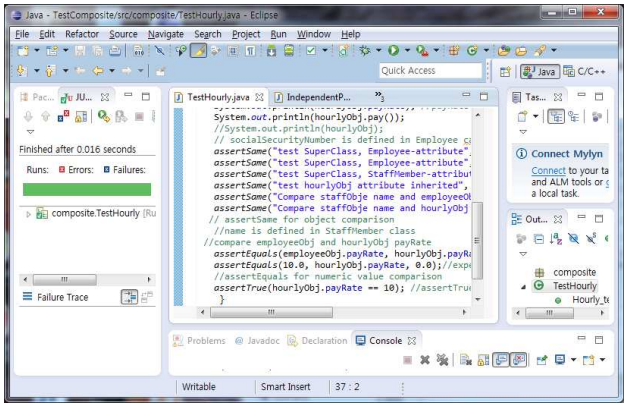


Figure 8. A JUnit Session for Testing a Composite Component

The screen shot above shows all the predicates testing the proper use of inherited attributes are proved to be true.

The essence of the suggested composite component testing is to make sure the proper use of inherited attributes with the use of various JUnit assert-predicates.

3. Testing Based on Design Pattern

A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design [8]. With the help of design pattern, frequently used software structures may be categorized for the purpose of increasing reusability and maintainability.

3.1. A Testing Guide for the Code with the Command Pattern

In the code having the command pattern, the code consists of an invoker, a command, and a receiver objects. The key to the component testing with this type of code is to verify the control flow in the sequence of the invoker, the command, and the receiver component. The testing procedure, therefore, is to show that the invoker activates the command, and the command activates the receiver.

As an example, the following code snippet shows the activation of an invoker, and the resulting testing output:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class TestCommandPattern {
   @Test
   public void test() {
   Light lamp = new Light();
   Command switchUp = new FlipUpCommand(lamp);
Command switchDown = new
FlipDownCommand(lamp);
   Switch s = new Switch(switchUp,switchDown);
   s.flipUp();
    s.flipDown();
   }
}
```

```
   Switch Up from Invoker
   Command.
   The light is on, received
   Switch Down from Invoker
   Command.
   The light is off, received
```

The above testing output shows the activation sequence of the invoker, the command, and finally the receiver object. For this output, each three component has an inclusion of probe statements as below:

```
   public class Switch {
      private Command flipUpCommand;
      private Command flipDownCommand;
      public Switch(Command flipUpCmd,
                    Command flipDownCmd) {
         this.flipUpCommand = flipUpCmd;
         this.flipDownCommand = flipDownCmd;
      }
      public void flipUp() {
      System.out.println("Switch Up from Invoker");
         flipUpCommand.execute();
      }
      public void flipDown() {
      System.out.println("Switch Down from Invoker");
         flipDownCommand.execute();
      }
   }

   public class FlipDownCommand implements
Command {
      private Light theLight;
      public FlipDownCommand(Light light) {
         this.theLight=light;
      }
      public void execute() {
      System.out.println("Command. ");
        theLight.turnOff();
      }
   }

   public class Light {
      public Light() {  }
```

```
    public void turnOn() {
      System.out.println("The light is on, received");
    }
    public void turnOff() {
      System.out.println("The light is off, received");
    }
  }
```

The output of JUNIT shows the example code has an exact of activation sequence of invoker, command, and receiver objects.

3.2. A Testing Guide for the Code with the Observer Pattern

The testing goal is to check the state of each observer object as the system state changes. To prove the state change of observer objects in observer pattern, generally the interface class is modified to include some method for testing purpose. The following is an example JUnit test code and its resulting output when we test a code based on the observer pattern.

```
@Test
public void test() {
LogSubject subject = new LogSubject();
IObserver ob1 = new Observer1();
IObserver ob2 = new Observer2();
subject.attach(ob1);
subject.attach(ob2);
subject.setState("state1");
System.out.println("testing the state of subject oject");
assertEquals("Is the name correct?",
"state1", subject.getState());
System.out.println("the state of Observer oject1: "
+ ob1.getState());
subject.setState("state2");
System.out.println("testing the state of subject oject");
assertEquals("Is the name correct?",
"state2", subject.getState());
subject.detach(ob1);
subject.setState("state3");
}
```

```
Observer1 has received update signal with new state: state1
Observer2 has received update signal with new state: state1
testing the state of subject oject
the state of Observer oject1: state1
Observer1 has received update signal with new state: state2
Observer2 has received update signal with new state: state2
testing the state of subject oject
Observer2 has received update signal with new state: state3
```

The Junit output shows the state change of an object, which reflect the state change of the system. For this testing, a necessary method has been added to the interface definition as shown below:

```
public interface IObserver {
void update(String state);
String getState();
public String getState(); // this line is added
}
```

All the observer objects implement the above interface, and share the method “update, ” which is invoked by the system to reflect the system state change. The method, “getState(),” has been added to the interface definition for testing purpose.

3.3. A Testing Guide for the Code with the Bridge Pattern

The testing goal is to check whether abstraction and implementation are really separated by including additional abstraction and implementation. With this example, the class, “Shape” is the abstraction, and the class “DrawingAPI” is the implementation. The testing implementation, “DrawingAPI3Test,” and the testing abstraction, “RectangleShape,” are added to the source code for testing the target code structure based on the bridge pattern. The following are the Junit test code and its corresponding output. The output shows that the source code in test has been made based on the bridge pattern .

```
import org.junit.Test;
public class BridgeTest {
        @Test
        public void test() {
            Shape[] shapes = new Shape[3];
            shapes[0] = new CircleShape(1, 2, 3, new
DrawingAPI1());
            shapes[1] = new CircleShape(5, 7, 11,
new DrawingAPI2());
            shapes[2] = new CircleShape(5, 7, 11,
new DrawingAPI3Test());
            for (Shape shape : shapes) {
              shape.resizeByPercentage(2.5);
              shape.draw();
            }
        }
}
```

The output is

```
API1.circle at 1.000000:2.000000 radius 7.500000
```

API2.circle at 5.000000:7.000000 radius 27.500000
Rectangle Test

In the above test, the separated abstraction and implementation has been simply included into the source code showing the maintainability with the bridge pattern.

4. Conclusions

In this study, a way of performing an independent path testing, component testing for a composite component, and software structure based on design patterns is suggested. Instead of the complex value settings for variables that form specific independent path, additional variables for tracking the program control flow and a result object for showing the result of computation have been suggested to reflect independent basis path testing. For a composite component based on the chain of responsibility, a use of JUnit predicates that tests a legal use of inherited attributes has been suggested. A new approach to testing based on design patterns has also been suggested. For testing a wider range of software structure, the following further studies are in need:

- multiple inheritance relationships;
- concurrent computation;
- a composite component consisting of objects with complex message passing; and
- more application of various forms of design patterns.

Consequently, a unit and component software testing, may be performed in an organized and guided way as suggested with the use of design patterns.

Appendix: An Example Use of A Trace Variable

```
public class binTree {
public static void search ( int key, int [] elemArray,
Result r )
{
 int bottom = 0 ;
 int top = elemArray.length - 1 ;
 int mid ;
  int loop = 0;
  r.found = false ; r.index = -1 ;
  while ( bottom <= top )
  {
        loop++;
        mid = (top + bottom) / 2 ;
        System.out.println("mid value: " + mid);
        if (elemArray [mid] == key)
        {
        System.out.println("key value is matched");
        System.out.println("loop count: " + loop);
                r.index = mid ;
                r.found = true ;
                return ;
        } // if part
        else
        {
        System.out.println("loop count: " + loop);
        if (elemArray [mid] < key) {
        System.out.println("upper half of the tree
                                to be searched");
        bottom = mid + 1 ;}
        else {
        System.out.println("lower half of the tree
                                to be searched");
        top = mid - 1 ;}
          }
      } //while loop
} // search
}
```

References

[1] Roger Pressman, Software Engineering: A Practitioner's Approach, Seventh Edition. MaGraw-Hill 2009.

[2] Carlo Ghezzi, Fundamentals of Software Engineering, 2nd Edition, Pearson 2002.

[3] M. E. Khan, Different Approaches to White Box Testing Technique for Finding Errors, International Journal of Software Engineering and Its Applications, Vol. 5 No. 3, July, 2011.

[4] Petar Tahchiev, JUnit in Action, Manning Publications 2010.

[5] I. Sommerville, Software Engineering, Eighth Edition, Addison Wesley 2007.

[6] T. J. McCabe, A Complexity Measure, IEEE Trans on Software Engineering, SE-2(4), pp. 308-20, 1976.

[7] StarUML available at http://en.wikipedia.org/wiki/StarUML.

[8] http://en.wikipedia.org/wiki/Software_design_pattern.