# Homework 4 Problem 2

Glen Madsen

March 8, 2018

Problem 2.

Overview: In working to design test cases for my code, I had not learned any testing heuristics yet, so I tried to do my best to just cover all the specifications I outlined and ensure that most of my code managed to be covered. To this end, I tested methods as individually as possible, made sure all of my if statements were traversed in my methods. I could not get my RepCheck() function to be violated while passing correct input, so that is the only place I lack code coverage. Other than checking methods individually I tried combining them in different ways and slowly build up to more and more complicated test cases from the more trivial checking of null and empty cases.

Overall I think my code is fairly well tested and I found it difficult to come up with more test cases that actually would prove useful past a certain point. Because I wrote my test cases according to possible cases and my specifications, I believe all of my test cases constitute 'black box testing'.

---

Test Cases Code

---

```
package hw4;

    import static org.junit.Assert.*;

    import java.util.Iterator;

    import org.junit.Before;
import org.junit.Test;

public final class GraphWrapperTest {
```

```
private final double JUNIT_DOUBLE_DELTA = 0.00001;


@Test
public void testNodeCreation() // Passes the Representation Invariant on Creation
{
GraphWrapper gr = new GraphWrapper();
}


@Test
public void list_One_Node() // Checks if one node can be successfully added and listed
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
Iterator¡String¿ gr_it = gr.listNodes();
assertEquals(gr_it.next(), "A");
}

@Test
public void list_Two_Node() // Checks if two nodes can be successfully added and listed
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
gr.addNode("B");
Iterator¡String¿ gr_it = gr.listNodes();
assertEquals(gr_it.next(), "A");
assertEquals(gr_it.next(), "B");
}

@Test
public void list_Three_Node() // Checks if three nodes can be successfully added and listed
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("B");
gr.addNode("A");
gr.addNode("C");
Iterator¡String¿ gr_it = gr.listNodes();
assertEquals(gr_it.next(), "A");
assertEquals(gr_it.next(), "B");
assertEquals(gr_it.next(), "C");
}
```

@Test
public void list_EmptyString_Node() // Checks if a node with a weird name can be successfully added
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("");
Iterator¡String¿ gr_it = gr.listNodes();
assertEquals(gr_it.next(), "");
assertEquals(gr_it.hasNext(), false);
}

@Test
public void list_Dup_Node() // Checks if a duplicate node can be added (it can't and shouldn't)
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
gr.addNode("A");
Iterator¡String¿ gr_it = gr.listNodes();
assertEquals(gr_it.next(), "A");
assertEquals(gr_it.hasNext(), false);
}

@Test
public void list_No_Nodes() // Checks if there are no nodes listed when there are no nodes created.
{
GraphWrapper gr = new GraphWrapper();
Iterator¡String¿ gr_it = gr.listNodes();
assertEquals(gr_it.hasNext(), false);

}

@Test
public void list_One_Children() // Checks if one child is successfully created and listed
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
gr.addNode("B");
gr.addEdge("A", "B", "one");
Iterator¡String¿ gr_it = gr.listChildren("A");

```
assertEquals(gr_it.next(), "B(one)");
}

@Test
public void list_NO_Child() // Checks the case where a child is tried to be created but the
2nd node
{ // is not present in the graph.
GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
gr.addEdge("A", "B", "one");
Iterator¡String¿ gr_it = gr.listChildren("A");
assertEquals(gr_it.hasNext(), false);
}

@Test
public void list_NO_Child2() // Checks the case where a child is tried to be created but the
1st node
{ // is not present in the graph.
GraphWrapper gr = new GraphWrapper();
gr.addNode("B");
gr.addEdge("A", "B", "one");
Iterator¡String¿ gr_it = gr.listChildren("B");
assertEquals(gr_it.hasNext(), false);
}

@Test
public void list_NO_Child3() // Checks the case where a child is tried to be created but the
2nd node
{ // is not present in the graph and children are listed a node not present in the graph.
GraphWrapper gr = new GraphWrapper();
gr.addNode("B");
gr.addEdge("A", "B", "one");
assertEquals(gr.listChildren("A"), null);
}

@Test
public void list_NO_Child_return() // Listing children of a node not present in a non-empty
graph
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("B");
assertEquals(gr.listChildren("A"), null);
```

```
}

@Test
public void Empty_Graph_Child_return() // Listing children of a node not present in a empty
graph
{
GraphWrapper gr = new GraphWrapper();
assertEquals(gr.listChildren("A"), null);
}

@Test
public void list_Empty_Child_return() // Listing children in a graph that has the node
checked, but
{ // the node has no children
GraphWrapper gr = new GraphWrapper();
gr.addNode("B");
Iterator¡String¿ gr_it = gr.listChildren("B");
assertEquals(gr_it.hasNext(), false);

}

@Test
public void list_NoNode() // Checks if Graph returns null if graph is empty and trying to
list children
{ // (I defined some undefined behavior in the specifications for certain cases)
GraphWrapper gr = new GraphWrapper();
assertEquals(gr.listChildren("A"), null);

}

@Test
public void list_Two_Children() // Lists two children in alphabetical order
{ GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
gr.addNode("B");
gr.addNode("C");
gr.addEdge("A", "B", "two");
gr.addEdge("A", "C", "one");

    Iterator¡String¿ gr_it = gr.listChildren("A");
assertEquals(gr_it.next(), "B(two)");
```

```
assertEquals(gr_it.next(), "C(one)");
}

@Test
public void list_Two_Children_Edge_Alphabetical() // Lists two children in alphabetical order by edge
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
gr.addNode("B");
gr.addEdge("A", "B", "apples");
gr.addEdge("A", "B", "apple");

Iterator¡String¿ gr_it = gr.listChildren("A");
assertEquals(gr_it.next(), "B(apple)");
assertEquals(gr_it.next(), "B(apples)");
}

@Test
public void list_Dup_Children() // Tests if duplicate edges are accounted for
{
GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
gr.addNode("B");
gr.addNode("C");
gr.addEdge("A", "B", "one");
gr.addEdge("A", "B", "one");

Iterator¡String¿ gr_it = gr.listChildren("A");
assertEquals(gr_it.next(), "B(one)");
assertEquals(gr_it.next(), "B(one)");
assertEquals(gr_it.hasNext(), false);
}

@Test
public void list_TwoNode_Children() // Lists children from one node, then another directly after
{ // Nothing new is tested here, its just a larger test case
GraphWrapper gr = new GraphWrapper();
gr.addNode("A");
gr.addNode("B");
gr.addNode("C");
```

gr.addEdge(”A”, ”B”, ”one”);
gr.addEdge(”A”, ”C”, ”two”);
gr.addEdge(”B”, ”B”, ”woo”);
Iterator¡String¿ gr_it = gr.listChildren(”A”);
assertEquals(gr_it.next(), ”B(one)”);
assertEquals(gr_it.next(), ”C(two)”);
gr_it = gr.listChildren(”B”);
assertEquals(gr_it.next(), ”B(woo)”);
}

@Test
public void list_LotsChildren_Children() // Creates a graph of a few nodes with many edges between them
{ // Tests many things at once but nothing no test case has not checked individually.
GraphWrapper gr = new GraphWrapper();
gr.addNode(”B”);
gr.addNode(”A”);
gr.addNode(”D”);
gr.addNode(”C”);

gr.addEdge(”A”, ”A”, ”one”);
gr.addEdge(”A”, ”A”, ”two”);
gr.addEdge(”A”, ”A”, ”three”);
gr.addEdge(”A”, ”A”, ”four”);
gr.addEdge(”A”, ”B”, ”five”);
gr.addEdge(”A”, ”C”, ”six”);
gr.addEdge(”A”, ”B”, ”seven”);
gr.addEdge(”A”, ”C”, ”eight”);
gr.addEdge(”A”, ”B”, ”nine”);
gr.addEdge(”A”, ”C”, ”ten”);
gr.addEdge(”A”, ”C”, ”two”);
gr.addEdge(”A”, ”A”, ”one”);
gr.addEdge(”A”, ”B”, ”one”);

Iterator¡String¿ gr_it = gr.listChildren(”A”);
assertEquals(gr_it.next(), ”A(four)”);
assertEquals(gr_it.next(), ”A(one)”);
assertEquals(gr_it.next(), ”A(one)”);
assertEquals(gr_it.next(), ”A(three)”);
assertEquals(gr_it.next(), ”A(two)”);
assertEquals(gr_it.next(), ”B(five)”);
assertEquals(gr_it.next(), ”B(nine)”);

```
assertEquals(gr_it.next(), "B(one)");
assertEquals(gr_it.next(), "B(seven)");
assertEquals(gr_it.next(), "C(eight)");
assertEquals(gr_it.next(), "C(six)");
assertEquals(gr_it.next(), "C(ten)");
assertEquals(gr_it.next(), "C(two)");
}
}
```

---

Other Test Cases

---

I felt as though my test cases from before were sufficient as they ended up giving me 94.8% code coverage. Since I believe they check all the corner cases I could think of and code is nearly fully covered except for Representation Invariant Violation test cases, I did not add any additional cases.