EE360T/EE382C-16: Software Testing
Problem Set 1 – **Yusuf Khan yk7862**

Out: Sep 7, 2022; Due: Sep 20, 2022 11:59pm
Submission: *.zip via Canvas
Maximum points: 40

1 Testing data structures

Consider the following implementation of a singly-linked list data structure, which represents a sequential container for boolean values:

```
package pset1;

import java.util.HashSet;
import java.util.Set;

public class SLList {
	Node first; // first node in <this> list
	Node last; // last node in <this> list

	static class Node {
		boolean elem;
		Node next;
	}

	boolean repOk() {
		// postcondition: returns true iff <this> is an acyclic list, i.e.,
		//                                    there is no path from a node to itself

		if (first == null || last == null) {
			return first == last;
		}
		Set<Node> visited = new HashSet<Node>();
		Node n = first;
		while (n != null) {
			if (!visited.add(n)) {
				return false;
			}
				if (n.next == null) {
				   return n == last;
			}
			n = n.next;
		}
		return true;
	}

	void add(boolean e) {
		//   precondition: this.repOk()
		//   postcondition: adds <e> in a new node at the end of <this>
		//                          list; the rest of <this> list is unmodified

		 Node newNode = new Node();
		newNode.elem = e;
		newNode.next = null;

		// add one node to an empty list
		if(first == null && last == null)
		{
		   first = newNode;
```

```
    last = newNode;
    // first.next remains null since there is only one node

}
// at least one element already in list
else if(first != null && first.next == null)
{
    last = newNode;
    first.next = last;

}
// already has at least two nodes in list
else
{
    last.next = newNode;
    last = newNode;
}
```

```
        }
}
```

1.1 Implementing add [4 points]

Implement the method add as specified.

```
void add(boolean e) {
    // precondition: this.repOk()
    // postcondition: adds <e> in a new node at the end of <this>
    // list; the rest of <this> list is unmodified

    Node newNode = new Node();
    newNode.elem = e;
    newNode.next = null;

    // add one node to an empty list
    if(first == null && last == null)
    {
       first = newNode;
       last = newNode;
       // first.next remains null since there is only one node

    }
    // at least one element already in list
    else if(first != null && first.next == null)
    {
       last = newNode;
       first.next = last;

    }
    // already has at least two nodes in list
    else
    {
       last.next = newNode;
       last = newNode;
    }
  }
```

1.2 Testing add [6 points]

Implement the two test methods in the following class SLListAddTester as specified:

```
package pset1;

import static org.junit.Assert.*;
import org.junit.Test;

public class SLListAddTester {
```

```
    @Test public void test0() {
        SLList l = new SLList();
        assertTrue(l.repOk());
        l.add(true);

        //   write a sequence of assertTrue method invocations that
        //   perform checks on the values for all the declared fields
        //   of list and node objects reachable from l

        assertTrue(l.first != null);

        assertTrue(l.first.elem == true);
        assertTrue(l.first.next == null);
        assertTrue(l.last == l.first); // first = last if only one node in list
        assertTrue(l.last.elem == true);
        assertTrue(l.last.next == null);
    }

    @Test public void test1() {
        SLList l = new SLList();
        assertTrue(l.repOk());
        l.add(true);
        assertTrue(l.repOk());
        l.add(false);
        assertTrue(l.repOk());

        //   write a sequence of assertTrue method invocations that
        //   perform checks on the values for all the declared fields
        //   of list and node objects reachable from l

        assertTrue(l.first != null);
        // your code goes here

        assertTrue(l.first.elem == true);
        assertTrue(l.first.next == l.last);
        assertTrue(l.last != null);
        assertTrue(l.last.elem == false);
        assertTrue(l.last.next == null);
    }
}
```

1.3 Testing repOk [10 points]

Consider testing the method repOk by writing a test suites that consists of valid or invalid lists. Specifically, implement test methods in the following class SLListRepOkTester such that: (1) each test allocates exactly one list l; (2) each test method makes exactly one invocation l.repOk(); (3) each test method invokes assertTrue(l.repOk()) or assertFalse(l.repOk()) as its last statement; (4) no invocation of add is made in any test method; (5) the test suite as a whole consists of all singly-linked list data structures – whether acyclic or not – that can possibly be constructed using up to 2 nodes.

```
package pset1;
```

```
import static org.junit.Assert.*;
import org.junit.Test;
import pset1.SLList.Node;

   public class SLListRepOkTester {
      @Test public void t0() {
            SLList l = new SLList();
            assertTrue(l.repOk());
      }

      @Test public void t1() {
            SLList l = new SLList();
            Node n = new Node();

             n.elem = true;
             n.next = null;

             l.first = n;
             l.last = n;

             assertTrue(l.repOk());
      }

      // list with one elem=false node
         @Test public void t2() {
            SLList l = new SLList();
            Node n = new Node();

            n.elem = false;
            n.next = null;

            l.first = n;
            l.last = n;

            assertTrue(l.repOk());

         }


      // list with one cyclic elem=true node
         @Test public void t3() {
            SLList l = new SLList();
            Node n = new Node();

            n.elem = true;
            n.next = n;

            l.first = n;
            l.last = n;

            assertFalse(l.repOk()); // false because cyclic with one node

         }


         // list with one cyclic elem = false node
         @Test public void t4() {
            SLList l = new SLList();
            Node n = new Node();

            n.elem = false;
            n.next = n;

            l.first = n;
            l.last = n;

            assertFalse(l.repOk()); // false because one node keeps cycling

         }
}
```

2 Testing contracts

Consider the following code snippet that declares a class C:

```
package pset1;

public class C {
	int f;

	public C(int f) {
		this.f = f;
	}

	@Override
	public boolean equals(Object o) {
		// assume this method is implemented for you
	}

	@Override
	public int hashCode() {
		// assume this method is implemented for you
	}
}
```

Consider next the following code snippet that declares a class D as a subclass of C:

```
package pset1;

public class D extends C {
	int g;

	public D(int f, int g) {
		super(f);
		this.g = g;
	}

	@Override
	public boolean equals(Object o) {
		// assume this method is implemented for you
	}

	@Override
```

```
    public int hashCode() {
        // assume this method is implemented for you
    }
}
```

2.1 Testing equals [15 points]

According to the contract for java.lang.Object any correct Java program must satisfy certain properties with respect to the equals methods; these properties include[1]:

> P_1: For any non-null reference value x, x.equals(null) should return false;
>
> P_2: It is reflexive: for any non-null reference value x, x.equals(x) should return true;
>
> P_3: It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true; and
>
> P_4: It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true;

You are to implement a test suite that checks three of the four properties – namely, P_1, P_2, and P_3 – with respect to the equals methods implemented in the three classes pset1.C, pset1.D, and java.lang.Object. Specifically, implement test methods in the following class EqualsTester such that: (1) each test method has exactly one invocation of assertTrue(...) or assertFalse(...); (2) each property is tested with respect to each of the three equals methods, e.g., the test suite must have three test methods for P_1; (3) each property is tested with respect to each combination of the three object types (C, D, or Object) for the inputs to equals, e.g., the test suite must have at least nine tests for P_3:

```
package pset1;

import static org.junit.Assert.*;
import org.junit.Test;

public class EqualsTester {
    /*
     *  P1: For any non-null reference value x, x.equals(null) should return false.
     */

    @Test public void t0() {
        assertFalse(new Object().equals(null));
    }

  @Test public void p1_c_test() {
      C c_object = new C(4);
      assertFalse(c_object.equals(null));
  }

  @Test public void p1_d_test() {
      D d_object = new D(4, 6);
      assertFalse(d_object.equals(null));
  }

    /*
     *  P2: It is reflexive: for any non-null reference value x, x.equals(x)
     *  should return true.
     */

  @Test public void p2_object_test() {
      Object new_object = new Object();
      assertTrue(new_object.equals(new_object));
  }

  @Test public void p2_c_test() {
      C c_object = new C(2);
      assertTrue(c_object.equals(c_object));
  }
```

```
@Test public void p2_d_test() {
    D d_object = new D(2, 9);
    assertTrue(d_object.equals(d_object));
}


    /*
     *  P3: It is symmetric: for any non-null reference values x and y, x.equals(y)
     *  should return true if and only if y.equals(x) returns true.
     */

@Test public void p3_object_on_different_object_test() {
    Object o1 = new Object();
    Object o2 = new Object();
    assertFalse(o1.equals(o2) && o2.equals(o1));
}

@Test public void p3_object_on_same_object_test() {
    Object o1 = new Object();
    Object o2 = new Object();
    o2 = (Object) o1; // Same reference
    assertTrue(o1.equals(o2) && o2.equals(o1));
}

@Test public void p3_object_on_c_test() {
    Object o = new Object();
    C c_object = new C(5);
    assertFalse(o.equals(c_object) && c_object.equals(o));
}

@Test public void p3_object_on_d_test() {
    Object o = new Object();
    D d_object = new D(5, 4);
    assertFalse(o.equals(d_object) && d_object.equals(o));
}

@Test public void p3_c_on_object_test() {
    C c_object = new C(2);
    Object o = new Object();
    assertFalse(c_object.equals(o) && o.equals(c_object));
}

@Test public void p3_c_on_different_c_test() {
    C c1 = new C(4);
    C c2 = new C(7);
    assertFalse(c1.equals(c2) && c2.equals(c1));
}

@Test public void p3_c_on_same_c_test() {
    C c1 = new C(3);
    C c2 = new C(3);
    assertTrue(c1.equals(c2) && c2.equals(c1));
}

@Test public void p3_c_on_d_test() {
    C c_object = new C(8);
    D d_object = new D(8, 3);
    assertFalse(c_object.equals(d_object) && d_object.equals(c_object));
}

@Test public void p3_d_on_object_test() {
    D d_obj = new D(2, 1);
    Object o = new Object();
    assertFalse(d_obj.equals(o) && o.equals(d_obj));
}

@Test public void p3_d_on_c_test() {
    D d_obj = new D(2, 1);
```

```
    C c_obj = new C(2);
    assertFalse(d_obj.equals(c_obj) && c_obj.equals(d_obj));
}

@Test public void p3_d_on_different_d_test() {
    D d1 = new D(2, 2);
    D d2 = new D(2, 8);
    assertFalse(d1.equals(d2) && d2.equals(d1));
}

@Test public void p3_d_on_same_d_test() {
    D d1 = new D(4, 7);
    D d2 = new D(4, 7);
    assertTrue(d1.equals(d2) && d2.equals(d1));
}


    /*
```

[1] http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

```
     * P4: It is transitive: for any non-null reference values x, y, and z,
     * if x.equals(y) returns true and y.equals(z) returns true, then
     * x.equals(z) should return true.
     */

    // you do not need to write tests for P4

}
```

2.2 Testing hashCode [5 points]

The contract for java.lang.Object additionally requires the following property that relates equals and hashCode[1]:

> P_5 : If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

Implement test methods in the following class HashCodeTester such that: (1) each test method has exactly one invocation of assertTrue(...) or assertFalse(...); (2) the property is tested with respect to each combination of the three object types (C, D, or Object) for the inputs to equals, so the test suite must have at least nine tests:

```
package pset1;

import static org.junit.Assert.*;
import org.junit.Test;

public class HashCodeTester {
    /*
     * P5: If two objects are equal according to the equals(Object)
     * method, then calling the hashCode method on each of
     * the two objects must produce the same integer result.
     */

  @Test public void p5_object_on_different_object_test() {
      Object o1 = new Object();
      Object o2 = new Object();
      if(o1.equals(o2)){
          assertTrue(o1.hashCode() == o2.hashCode());
      }
  }

   @Test public void p5_object_on_same_object_test() {
      Object o1 = new Object();
      Object o2 = new Object();
      o2 = (Object) o1; // Same reference
      if(o1.equals(o2)){
          assertTrue(o1.hashCode() == o2.hashCode());
      }
  }

   @Test public void p5_object_on_c_test() {
      Object o = new Object();
      C c_object = new C(5);
      if(o.equals(c_object)) {
          assertTrue(o.hashCode() == c_object.hashCode());
      }
  }

   @Test public void p5_object_on_d_test() {
      Object o = new Object();
      D d_object = new D(5, 4);
      if(o.equals(d_object)) {
          assertTrue(o.hashCode() == d_object.hashCode());
      }
  }
```

```
    @Test public void p5_c_on_object_test() {
        C c_object = new C(2);
        Object o = new Object();
        if(c_object.equals(o)) {
            assertTrue(c_object.hashCode() == o.hashCode());
        }
    }

    @Test public void p5_c_on_different_c_test() {
        C c1 = new C(4);
        C c2 = new C(7);
        if(c1.equals(c2)) {
            assertTrue(c1.hashCode() == c2.hashCode());
        }
    }

    @Test public void p5_c_on_same_c_test() {
        C c1 = new C(3);
        C c2 = new C(3);
        if(c1.equals(c2)) {
            assertTrue(c1.hashCode() == c2.hashCode());
        }
    }

    @Test public void p5_c_on_d_test() {
        C c_object = new C(8);
        D d_object = new D(8, 3);
        if(c_object.equals(d_object)) {
            assertTrue(c_object.hashCode() == d_object.hashCode());
        }
    }

    @Test public void p5_d_on_object_test() {
        D d_obj = new D(2, 1);
        Object o = new Object();
        if(d_obj.equals(o)) {
            assertTrue(d_obj.hashCode() == o.hashCode());
        }
    }

    @Test public void p5_d_on_c_test() {
        D d_obj = new D(2, 1);
        C c_obj = new C(2);
        if(d_obj.equals(c_obj)) {
            assertTrue(d_obj.hashCode() == c_obj.hashCode());
        }
    }

    @Test public void p3_d_on_different_d_test() {
        D d1 = new D(2, 2);
        D d2 = new D(2, 8);
        if(d1.equals(d2)) {
            assertTrue(d1.hashCode() == d2.hashCode());
        }
    }

    @Test public void p3_d_on_same_d_test() {
        D d1 = new D(4, 7);
        D d2 = new D(4, 7);
        if(d1.equals(d2)) {
            assertTrue(d1.hashCode() == d2.hashCode());
        }
    }
}
```