SWE 619

Final

Mozhgansadat Momtaz Dargahi,G01114750

December 13st,2021

Question 1

1. **For enQueue, write (i) a partial contract and (ii) a total contract. For each part, if you need to change the code for the contract, do so and explain what you did**

Partial contract: required: element e cannot be null e!=null

Effect: add element e to the queue and increment the size of queue by one

Total contract: Required: None

Effect: if e is null throw NPE throw

Otherwise add element e to the queue and increment the size of queue by one

```
public void enQueue (E e) {
  if (e==null){
     throw new NullPointerException;}
  else {
     elements.add(e);
     size++;
}
```

For this question we are adding an element e to the queue, so we need to check if e is not null bcz we cannot add null to the arraylist and we need to detect it before adding it to the queue. This condition can be checked both by precondition or by adding it into the postcondition and throw NPE exception.
The other behavior (happy path)is adding element e to the arraylist.and increment the arraylist size by one.

2. Write the rep invs for this class. Explain what they are.

1-Elements ! null the element array is not null
2-Size of element array greater than or equal 0 ; size >= 0.
In contrst to the stack example 13, here we use arraylist we dynamically allocate objects so the rep invariant is so simple in that :
In our methods we need to check that those rep invariant are still hold after the method execution. After enqueue we consider that e is not null, so after adding e to the elements elements is not null as well, so this rep invariant holds
In deque we assume we assume elements is not null so when in this method we first check if the size is not zero then we remove the head of queue, in worst case the arraylist point to empty list not null, so this rep invariant also holds.

3. Write a reasonable toString() implementation. Explain what you did

```
    public String toString () {
        return elements.toString();
    }
```

Or

```
@Override
  public String toString(){
    return elements.toString();
  }
```

Bcz here we use arraylist we can simply reuse the toString for attaylist as demonstrated above.

4. Consider a new method, deQueueAll(), which does exactly what the name suggests. Write a reasonable contract for this method and then implement it. Be sure to follow Bloch's advice with respect to generics. Explain what you did

Contract: Requires: None
Effects: remove everything from this (or the queue)

```
Public void deQueueAll() {
for (Element e : elements) {if e==null throw NPE}
for (Element e : elements) {deQueue() }
  }
    }
```

For each elements of the array I loop over the arraylist and call the deQueue function. This loop iterates over all elements of array until size==0 and dequee will no longer run and array list point to empty array.

Alternatives : from bloch item 31:

```
// wildcard type for parameter that serves as an E consumer
  //Effect: If dst is null throws NPE,
  // else removes all the elements from this and puts them into dst
  public void deQueueAll ( Collection<? super E> dst) {
    if (dst==null) throws NPE,
else{
     while (!isEmpty()) { dst.add(deQueue()); }
```

```
}
```

I explain the last method I provide here: First we check if the dst file is not empty otherwise we until the arraylist is not empty we dequeue from the queue and add to the dst file.

5.Rewrite the deQueue() method for an immutable version of this class. Explain what you did

```
2. /**
3.       * Non-destructively remove the head of the queue.
4.       *
5.       * @return a modified copy of this queue with the head of this queue
   removed therein.
6.       * @throws IllegalStateException if this queue is empty.
7.       */
8.      public Queue<E> deQueue () {
9.          if (size == 0) throw new IllegalStateException("Queue.deQueue");
10.         final List<E> new_elements = new ArrayList<E>(size);
11.         new_elements.addAll(elements);
12.         new_elements.remove(0);
13.         return new Queue(new_elements, size - 1);
14.     }
```

For this question we first check if the queue is empty and if it is empty we throw a `IllegalStateException`

`Otherwise : we create new queue in order not to  change the original queue,`

`then we add all elements of the of the original queue to the new queue`

`then we remove the head of this queue  and use the constructor that previously developed to return the new queue with the removed head and new size.`

`For complicity I provide the queue constructor here as well:`

```
public Queue() {
    this.elements = new ArrayList<E>();
    this.size = 0;
}

// private, so no explicit contract.
private Queue(List<E> new_elements,
              int new_size) {
    this.elements = Collections.unmodifiableList(new_elements);
    this.size = new_size;
}
```

It is a good to make deque final to prevent the override of deque

6. Write a reasonable implementation of clone(). Explain what you did.

The question is vague if we assume that we convert the queue to immutable class, in this case we do not need a clone bcz immutable class can be only share.

Otherwise here is the clone:

```
  @Override effects:return  the exact copy of queue
   protected Queue<E> clone()  {
try{
      Queue<E> result = (Queue<E>) super.clone();
      result.elements = new ArrayList<E>(elements);
      return result;
} catch (CloneNotSupportedException e) {
      throw new AssertionError();
   }
   }
```

Creating a copy using the clone() method. The class whose object's copy is to be made must have a public/private clone method in it or in one of its parent class. Every class that implements clone() should call super.clone() to obtain the cloned object reference. The class must also implement java.lang.Cloneable interface whose object clone we want to create otherwise it will throw CloneNotSupportedException when clone method is called on that class's object.

2 Question 2

1.What would be good rep invariants for this class? Explain each.

Rep invariants:
1-choiceList !null
2-size(choice)>0
choice list cannot be null otherwise we cannot choose random variable of it, we have method choose and if we assume this rep inv holds when we enter the method, it will still hold when we excetue the method bcz in method we do not change the list at all.
In addition the choices that we create the choice list from it cannot be empty otherswise we face the same issue and cannot return the random variable from choicelist.if we assume this rep inv holds when we enter the choose method for the same reason as explained above this will still holds after the method choose is executed.

2.Supply suitable contracts for the constructor and the choose() method and recode if necessary. The contracts should be consistent with your answer to the previous question. Explain exactly what you are doing and why.

Constructor contract:
Requires: None
Post: if choice is null, throw IAE
Post: if choice is empty, throw exception
Post: choices.contain(null), throw exception
Post: create a chooser with choices

Another contract for constructor:
Precondition: choices cannot be null, cannot be empty, cannot contain null
Post: create a chooser with choices

Contract for choose method:
Requires: None
Post: return random choice in List<T> choice List

Here I am creating contact for the contor and choose method, in constructor, we pass an arraylist and construct the choice list, so here we need to check if any elements of choice is not null or the choice is not emoty or null,
For choose we pass the constructed arraylist to the choose method. This is an observer method than only rerun a random choice form the choicelist

3. Argue that the choose() method, as documented and possibly updated in your previous answers, is correct. You don't have to be especially formal, but you do have to ask (and answer) the right questions.

Yes it is correct, bcz of two reasons:
1-it satisfies its contract (written above)
Consider the contract written above for method choose there is no precondition, but the method choose will satisfies its postconditions. Given the rep invariant above the choose method will exceuted successfully and return a random choice in choicelist.
2- It preserves rep invariants , for this method we have two rep invariants choiceList !null
2-size(choice)>0 , given that when we enter this method, these two rep-inv hold, they will still hold bcz the method does not change the arraylist. And in general this method does not change anything and it is called an observer method

3 Question 3

1.What is wrong with toString()? Fix it

It iterates throw object up to the element.length, but we want the contents up to size and not elements.length (which outputs everything in the array)

Fix: print “stack”, elements.length=size

2- As written, pushAll() requires documentation that violates encapsulation. Explain why and then write a contract for pushAll()

If any of them is null push all need to throw exception righaway, bcz we don’t want to have a unfinished work in the push all method either we want to successfully add all elements of collection to stack or not at all:

Example:

Imagine a collection of [“dog”, null, “cat”]

First we push the dog and it is acceptable but when it hits null it throws exception and we have an inconsistence state, so we need to first check if the collection has any null element.

If any of element is null thow exception bcz in rep inv none of elements cannot be null

Contract: if any element is null, then raise exception,

Otherwise add everything to this (or the stack)

```
Public void pushAll (object[] collection){
        For (Object obj: collection){if obj==null throw NPE;}
        For (Object obj: collection){push (obj);}
}
```

We need to make the push method final or private otherwise someone can override it and make its rep inv invalid

3. Rewrite the pop() method for an immutable version of the Stack class. Keep the same instance variables. Rewrite what you did

```
return a modified copy of this stack with the top of this stack removed therein.
* @throws IllegalStateException if this stack is empty.
```

```
public object  Pop() {
        if (IsEmpty)
                throw new  IllegalStateException ("Stack is empty");
        final Object new_elements = new Object(size);
        new_elements.addAll(elements);
        new_elements.remove(size-1);
```

```
            return new StackInClass (new_elements, size - 1);
        }
```

The constructor looks like:

```
   public StackInClass() { this.elements = new Object[0]; }

    // private, so no explicit contract.
    private StackInClass (Object new_elements,
                  int new_size) {
        this.elements = Collections.unmodifiableList(new_elements);
        this.size = new_size;
    }
```

We first check if the stack is not empty, otherwise throw NPE,

Else we create a new object with the size equal to original stack and add all element of original stack to the new stack , and we remove the element on top of stack, then we use the constructore to return the new stack with top element removed from the original stack and new size.

4.Implementing the equals() method for this class is a messy exercise, but would be much easier if the array was replaced by a list. Explain why. Note: You are not required to provide a implementation in your answer, but if you find it helpful to do so, that's fine

If we were to write equal for stack that has array structure we need **to loop over up to the size of array and compare each elemenets in the two array by their index size** , and we have to write this equal (we cannot reuse equal for array) bcz equal in array check elements up to the array length but in stack we want to check up to the size of array,

["dog", null] ["dog"] this implementation of stack considers them the same but based on array these two collections are different so we need to check up to the array size

But if we use **array list we can reuse arraylist equal bcz arraylist is** a dynamic allocating data structure and we don't need to be worry of size not equal to length to the existence of null to the size of the arraylist cases like example above.

Whenever we override equal we need to override hashcode according to bloch that this is another reason to replace the array with arraylist, here I provide the Bloch reciepie of overridden hashcode()

```
@Override public int hashCode() {
   int result = Short.hashCode(areaCode);
   result = 31 * result + Short.hashCode(prefix);
   result = 31 * result + Short.hashCode(lineNum);
   return result;
}
```

4 Question 4

Consider the program below (y is the input).

```
1 {y ≥ 1} // precondition
2 3
x := 0;
4 while(x < y)
5 x += 2;
6 7
{x ≥ y} // post condition
```

1. **Informally argue that this program satisfies the given specification (pre/post conditions)**

Our Goal here is to informally reason that the program is correct wrt a given precondition P and postcondition Q.

The pre condition is already given we check if it execution terminates and satisfies the post condition: Therefore we check assume P holds, if S successfully executes, then Q hold.
This program is correct and valid because first y>=1 and for x<y we increment the value of x by 2 until it successfully executed and reaches x ≥ y , which confirms the post-condition. Besides the loop invariant that we propose in next questions holds when we enter the loop and while it runs in the while loop.
let say y=1 it satisfies the pre-condition and it enters the loop (x=0<y=1) :

first loop: x=x+2=2

for the next loop round it does not satisfy the B (x<y since x=2 ! <y=1):

so we do not enter the loop and we move to the post condition:

x>=y which is the case : 2>=1

Lets consider another y=2, it satisfies the pre-condition (y>=1) and it enters the loop (x=0<y=2) :

first loop: x=x+2=2

for the next loop round it does not satisfy the B (x<y since x=2 ! <y=2):

so we do not enter the loop and we move to the post condition:

x>=y which is the case : 2>=2

Lets consider another y=3, it satisfies the pre-condition (y>=1) and it enters the loop (x=0<y=3) :

first loop: x=x+2=2

second round we enter the loop since (x=2 < y=3), we increament x by 2 and we get x=4,

for the next loop round it does not satisfy the B (x<y since x=4 ! <y=3):

so we do not enter the loop and we move to the post condition:

x>=y which is the case : 4>=3

…

Here we saw from the logical reasoning and examples provided that describes different scenario of x and y wrt to the loop , and we see that the program S is correct wrt to the given pre and post conditions.

2. Give 3 loop invariants for the while loop in this program. For each loop invariant, informally argue why it is a loop invariant.

Here are the three loop invariants:

x>= 0 ;

y>=1;

True;

x>= 0 ;

The first invaraints x>=0 is a loop invariants for the given loop: bcz this property holds when we enter the loop, x is initially is zero, so it is zero and if we continue running the loop x gets increamented and we still preserve this loop invariant of x>=0 , so we can argue that this invariant is a loop invariant and it preserves till the loop executed. Here are the examples:

First round : X=0 x>=0

Second round (if it satisfies the B) x=2 , x>=0;

…..

y>=1;

The second invaraints y>=1 is a loop invariants for the given loop: bcz this property holds when we enter the loop, y is initially equal or greater than 1, and in the loop body we do not even change that y to check if it still preserves the condition of loop invariant or not. Here are the examples :

First round : pre condition y >=1,

When we enter the loop body it holds and when we execute the loop body it still hold bcz we only incerasment x nothing happens to y/…

…..

True;

The third invaraints True is a loop invariants for the given loop: bcz this is a very weak precondition and we only need to hit that location that we want to check its loop invariant

3. *Sufficiently strong loop invariants:* Use a sufficiently strong loop invariant to formally prove that the program is correct with respect to given specification. This loop invariant can be one of those you computed in the previous question or something new.

• Note: show all works for this step (e.g., obtain weakest preconditions, verification condition, and analyze the verification condition).

• Recall that if the loop invariant is strong enough, then you will be able to do the proof. In contrast, if it is not strong enough, then you cannot do the proof.

• We need a loop invariant to obtain the WP of loop

• Loop invariant I: captures the meaning of the loop (manually provided by you)

o property that holds when the loop entered

o is preserved after the loop body is executed (inductive loop invariant)

Now we need to use Hoar Triple logic to prove it as the

WP(while [I] b do S, {Q}) =
I && (I && b => WP(S,I) && (I && !b) => Q)

b: x <y , Q: x>=y ,I= x>=0

WP(while[x>=0] x<y do x+= 2, {x >=y}) =

1-x>=0;

2-x>=0 & x<y => WP(x+=2, {x>=0})

y>0 or y>=1 (from the precondition we had) => x+2>=0

True => True

3- x>=0 & !(x<y) => Q=x>=y;

x>=0 & x>=y => x>=y;

we know that x>=0 so we shoreten the first part to x>=y => x>=y; True

or we can say Logical Equivalence: ((p & q) => r) <=> ((p => r) or (q => r
(x>=0 => x>=y) or (x>=y => x>=y)

(x>=0 => x>=y) or True is True

= x >= 0 && True && True
= x >= 0

WP(x := 0; {x>=0}) = 0>=0 True

Creating and checking verification condition vc:

P => WP(while[x>=0] x <y do x+= 2, {x >=y}) =
P => True
y>=1 => True =>True

Thus using this loop invariant we can prove the validity of the Hoar Triple logic.

The given loop invariant x>=0, holds true before entering the while loop as well as after exiting it. While we increament the x in loop this invariant still holds and it does not get negative for instance. So the x>=0 is a loop invariant as it will stay true when enters the loop and holds till the program is exceuted.

4.Insufficiently strong loop invariants: **Use another loop invariant (could be one of those you computed previously) and show that you cannot use it to prove the program.**
• Note: show all work as the previous question

WP(while [I] b do S, {Q}) =
I && (I && b => WP(S,I) && (I && !b) => Q)

b: x <y , Q: x>=y ,I= x>=0

WP(while[y>=1] x<y do x+= 2, {x >=y}) =

1-y>=1;

2-y>=1 & x<y => WP(x+=2, {y>=1})

y>=1 & x<y => 2>=1 True

y>=1 & x<y => True

3- y>=1 & !(x<y) => x>=y

y>=1 & x>=y => x>=y

x>=1 => x>=y Counter example: x=1 and y=2 x<y

= y>= 1 && …. && False
= False

WP(x := 0; False) = False

Since the wp of the while loop is false, the wp of the program is also false, this implies that the vc of the program is also false for the given insufficient loop invariant y>=1 .

P => False
Y>=1 => False
False

Since the vc with the invariant y>=1 is proven false, it implies that the invariant is not strong enough to prove the Hoar Triple and thus the validity of the given program can not be proven in this case. In other words, we are not able to prove the validity of the program with this insufficient loop invariant.

5 Question 5

1. **What does it mean that a program (or a method) is *correct*? Give (i) an example showing a program (or method) is correct, an (ii) an example showing a program (or method) is incorrect**.

   For a program to be correct it need to satisfies two conditions:

   1-it satisfies its contract

   2- It preserves rep invariants

   Yes it is correct, bcz of two reasons:

   1-it satisfies its contract (written above)

   Consider the contract written above for method choose there is no precondition, but the method choose will satisfies its postconditions. Given the rep invariant above the choose method will exceuted successfully and return a random choice in choicelist.

   2- It preserves rep invariants , for this method we have two rep invariants choiceList !null

   2-size(choice)>0 , given that when we enter this method, these two rep-inv hold, they will still hold bcz the method does not change the arraylist.

   Counter example :

```
public class Members {
  // Members is a mutable record of organization membership
  // AF: Collect the list as a set
  // rep-inv2: members != null && no duplicates in members
  // for simplicity, assume null can be a member...

  List<Person> members;   // the representation

  //  Post: person becomes a member
  public void join (Person person) { members.add   (person);}
```

   The join method satisfies its contact bcz there is no precondition, but it satisfies its postconditionand add person to member.

   The join method does not preserves the rep invariant 2 bcz we do not check if the element is already exists in the member and add that to the member and may have duplicate person in member. So this method is not correct wrt to its rep inv.

   To modify it we can put this :

```
  //  Post: person becomes a member
  public void join (Person person) {
If !member.contain(person){
        members.add   (person);}
        }
```

2. Explain the difference between rep invariants, loop invariants, and contract/specifications (i.e., pre/post conds). Use concrete examples to demonstrate the difference

Rep invariant:

Every object instaniation need to hold this, we assume the **rep invariant holds when we enter** the method and **we check If the changes of the method affect the rep invariant or not**. A representation invariant is a condition concerning the state of an object. The condition can always be assumed to be true for a given object, and operations are required not to violate it.

This rep invariant need to be satisfied by all the methods.

```
public class Members {

  // Members is a mutable record of organization membership

  // AF: Collect the list as a set

  // rep-inv1: members != null

  // rep-inv2: members != null && no duplicates in members

  // for simplicity, assume null can be a member...


  List<Person> members;   // the representation


  //  Post: person becomes a member

  public void join (Person person) { members.add   (person);}

  //  Post: person is no longer a member

  public void leave(Person person) { members.remove(person);}
```

for this example we assume the rep invariants hold when we enter the method and we need to check if all methods preserve that rep invariant:

in this example both method preserve and hold rep invariant 1

but the join method does not preserve the rep invariant2 bcz when we wantto join a element to member we do not check if the element is already in the member, so can end up having a duplicate.

loop invariants

It holds the meaning of the loop and that is a property that holds **when the loop entered** and preserved **after the loop body is exceuted**. A loop invariant is a condition [among program variables] that is necessarily true immediately before and immediately after each iteration of a loop.

Example:

```
1 {y ≥ 1} // precondition
2 3
x := 0;
4 while(x < y)
5 x += 2;
6 7
{x ≥ y} // post condition
```

We detect some loop invariant consider the loop invariant x>=0.

This while loop can be a part of method but we only need to check these conditions specific to the loop:

x>= 0 ;

The first invaraints x>=0 is a loop invariants for the given loop: bcz this property holds when we enter the loop, x is initially is zero, so it is zero and if we continue running the loop x gets increamented and we still preserve this loop invariant of x>=0 , so we can argue that this invariant is a loop invariant and it preserves till the loop executed. Here are the examples:

First round : X=0 x>=0

Second round (if it satisfies the B) x=2 , x>=0;

…..

What we do is basically is checking Hoar Logic conjuctors:

I : the loop invariant (should hold when entering the loop)

I && b => I : I is preserved after each loop body execution

I && !b => Q if the loop terminates, the post condition holds

contract/specifications (i.e., pre/post conds)

Every method has its own specifications that we need to check if that method satisties its own contact.

The contract of a class or interface, in Java or any other OO language, generally refers to the publicly exposed methods (or functions) and properties (or fields or attributes) of that class interface along with any comments or documentation that apply to those public methods and properties

```
public class Members {

  // Members is a mutable record of organization membership
```

```
// AF: Collect the list as a set

// rep-inv1: members != null

// rep-inv2: members != null && no duplicates in members

// for simplicity, assume null can be a member...

List<Person> members;   // the representation

//  Post: person becomes a member

public void join (Person person) { members.add   (person);}

//  Post: person is no longer a member

public void leave(Person person) { members.remove(person);}
```

In this example we can see we have two methods that each has its own contact that they should hold their preconditions (when enter the method) and after exceuting the method should satisfy the postcondition.

In this example join method satisfy both its pre and post constions

But only if we assume rep invariant2, the leave method would satisfy the contact (including pre and postcondions)

For rep-inv1 the leave method does not gurauntee to remove the element bcz we may have aduplicate, so with this rep-inv this method does not satisfy its postcondition and contract.

Conclusions Summary : From all above examples we saw the differences between contract (including pre and post conditions) in which each method has its own contract that we need to check if the method holds precondition upon entering the method and post condition should hold upon exiting the method, For rep invariant we assume the invariant/s hold when we enter the method and we check if the rep-inv still holds after executing the method. In case of loop inv we check if the loop invariant holds when we enter the loop and is preserved after the loop body is executed (inductive loop invariant)

3.What are the benefits of using JUnit *Theories* comparing to standard JUnit tests. Use examples to demonstrate your understanding

1-A test captures the intended behavior in one particular scenario. A developer, however, knows more about how a program should behave than can be expressed as a set of a few concrete examples. There are behaviors that can be easily and precisely described, but cannot be captured as a simple example, because they pertain to a large, or infinite, set of concrete executions.

JUnit Theories make tests more generalized,instead of testing for specific values, you might require to test for some wider range of acceptable input values. For this scenario, JUnit provides theories.

Theories not only makes your tests more expressive but you will see how your test data becomes more independent of the code you're testing. This will improve the quality of your code since you're more likely to hit edge cases, which you may have previously overlooked. The idea behind them is to create test cases that test on assumptions rather than static values.

This @theories development that enables developers also to make general statements at the level of tests. We call these statements theories. they can empower developers in several important ways. • Theories facilitate the development of complete abstract interfaces by allowing the developer to remain at the level of domain abstraction such as numerical systems or currencies. • Theories enable reuse of the test code, since many traditional specific tests can be recast as instantiations of a specific theory. • Theories afford the use of automated testing tools to amplify the effects of a theory by testing a theory with multiple data points.

Traditional unit tests in test-driven development compare a few concrete example executions against the developer's definition of correct behavior. However, a developer knows more about how a program should behave than can be expressed through concrete examples. These general insights can be captured as theories, which precisely express software properties over potentially infinite sets of values. Combining tests with theories allows developers to say what they mean, and guarantee that their code is intuitively correct, with less effort. The consistent format of theories enables automatic tools to generate or discover values that violate these properties, discovering bugs that developers didn't think to test for.

Example:

Now, let's start with a simple test class for a type that performs arithmetic on numbers of dollars:

```
public class DollarTest {
    @Test
    public void multiplyByAnInteger() {
        assertThat(new Dollar(5).times(2).getAmount(), is(10));
    }
}
```

In this example we test the program on only one particular example using annotation @Test.Now,we extends JUnit to allow the specification and verification of general properties. First, give Dollar test a new superclass:

```
import net.saff.theories.api.TheoryContainer;

public class DollarTest extends TheoryContainer {

  // Next, add a theory to the test class:

  @Theory

  public void multiplyIsInverseOfDivide(int amount, int m) {

    assumeThat(m, not(0));

    assertThat(new Dollar(amount).times(m).divideBy(m).getAmount(), is(amount));
```

```
}
```

This theory expresses a general truth about currency multiplication and division: one is the inverse of the other. By adding two parameters to the method, and adding the assumption that m is not zero, you've specified that this theory should hold true for any value of amount and all non-zero values of m.

Another example from inclass excerices 11 is:

```
package swe619.team6.assignment10;


import static org.junit.Assert.*;
import static org.junit.Assume.*;
import org.junit.experimental.theories.DataPoint;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;
import org.junit.runner.RunWith;
import swe619.team6.InClass11.ColorPoint;
import swe619.team6.InClass11.Point;
@RunWith(Theories.class)
public final class ColorPointTest {
  @DataPoint
  public static final Point DATA_POINT_0 = null;
  @DataPoint
  public static final Point DATA_POINT_1 = new Point();
  @DataPoint
  public static final ColorPoint DATA_POINT_2 = new ColorPoint();
  @DataPoint
  public static final ColorPoint DATA_POINT_3 = new ColorPoint();
  static {
    DATA_POINT_1.setX(1);
    DATA_POINT_1.setY(2);

    DATA_POINT_2.setX(1);
    DATA_POINT_2.setY(2);
```

```
    DATA_POINT_3.setX(1);
    DATA_POINT_3.setY(2);
  }
  private static int counter = 0;
  @Theory
  public void testList(Point X, Point Y) {
    assumeNotNull(X, Y);
    assumeTrue(X.equals(Y));
    System.out.println("counter = " + (counter++));
    assertTrue(Y.equals(Y));
  }
}
  @Test
            public void test()  ) {

              assumeNotNull((2,1), (2,2));
              assumeTrue((2,1) .equals(2,2,));
              assertTrue(2,2.equals(2,1));
```

With @Theory Junit we could test all the possible combinations of points which is 16 in this example but with standard @Junit test we need to test for specific scenario.

4. Explain the differences between proving and testing. In addition, if you *cannot* prove (e.g., using Hoare logic), then what does that mean about the program (e.g., is it wrong)?

Proving or verification means reason the source code without running it, check the program over all possible/infinite tests/inputs

Testing: check the program over finite number of tests/inputs (dynamic analysis) run the program .unit testing pass or fails checks

not being able to prove simply means we cannot prove it using this specific way (in case of using hoar logic to prove a program if a specific loop invariant did not respond well, it does not mean hoar logic is wrong)
It does not mean that you disprove it or show that the Hoare triple is invalid. (in fact, we know the Hoare tripple is valid if we used a different loop invariant)

5. Explain the Liskov Substitution Principle (LSP). Use a concrete example to demonstrate LSP. Note: use a different example than the one given in Liskov.

The Liskov Substitution Principle in practical software development. The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass. The Liskov substitution principle says that any important property of a type should also hold for all its subtypes so that any method written for the type should work equally well on its subtypes [Liskov87].

Here is a very informal example：

```
public class Bird{}
public class FlyingBirds extends Bird{
   public void fly(){}
}
public class Ealgle extends FlyingBirds{}
public class Rooster extends Bird{}
```

We know that , Ealgle is a Bird. Indeed it is a specialization of a bird . Eagle is indeed a flying birds that makes us to model this with inheritance. However if in code we made FlyingBird derive from Bird, then Eagle should be usable anywhere you expect a Bird.

Imagine you put the attributes of birds like specific wing and feather structures. In your Flyingbirds you override those attriute to some values more specific but still reference to bird (which is a case bcz flyingbird need to have a specific wing and feather structures to be able to fly),this will satisfy the LSP. In this case Flyingbird pass the Liskov Substitution Test with bird and the abstraction of having flyingbirds inherit from bird is a good one.

A more formal example is :

In class exercise 9A we saw that Bloch's Point/ColorPoint in which point class was extended with color point class did not satisfy the LSP, bcz in order to satisfy the contract of overridden equal we needed to use get class in equal method but it breaks LSP.

but the other class namely CounterPoint satisfy the LSP since in this class extended method we do not need to override equal bcz unlike color point, the counter point does not add a key feature we do not care if (1,2, counter =2) != (1,2, counter=3) this is not a key attribute this is only an internal attribute ,we use only super.equal that use the same equal to superclass Point; therefore the LSP is no longer violated and this is also another good example of LSP:

```
public class Point {  // routine code

   private int x; private int y;

   ...

      @Override public boolean equals(Object obj) {  // Standard recipe

      if (!(obj instanceof Point)) return false;
```

```
    Point p = (Point) obj;
    return p.x == x && p.y == y;
  }
}

public class CounterPoint extends Point
                private static final AtomicInteger counter =
                new AtomicInteger();

public CounterPoint(int x, int y) {
  super (x, y);
  counter.incrementAndGet();
}
public int numberCreated() { return counter.get(); }
@Override public boolean equals (Object obj) { supe.equal() }
}
```

6 Question 6

This question helps me determine the grade for group functioning. It does not affect the grade of this final.
1. Who are your group members?

- Lucas Leitz (lleitz@gmu.edu)
- Luke Young (lyoung29@gmu.edu)
- M. Mackenzie High (mhigh@gmu.edu)
- Mozhgansadat Momtaz Dargahi (mmomtazd@gmu.edu)

They all have a Regular participant; contributed reliably but Mackenzie High is a relay expert in software engineering that helps us in all cases.

7 Question 7

There is no right or wrong answer for the below questions, but they can help me improve the class. I might present your text verbatim (but anonymously) to next year's students when they are considering taking the course (e.g., in the first week of class) and also add your advice to the project description pages.
1. What were your favorite **and** least aspects of this class? Favorite topics? Favorite : Junit testing , least favorite: equal overriding

2. Favorite things the professor did or didn't do? Favorit Hoar logic
3. What would you change for next time? Provide slides