Maurice Joy

SWE 619 DL-01 Fall 21

Final Exam

Question 1 Queue.java

1. (i) partial contract

/** Precondition – e is not null
* Parameters – e is the object to be added to the Queue.
* Effects – Adds element to the end of the Queue.
* Modifies – This
**/
No change to code required.
(ii) total contract
/**
* Parameters – e is the object to be added to the Queue.
* Effects – If e is null, throw NulllPointerException,
* else Adds element to the end of the Queue.
* Modifies – This
**/
Code change

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

2. Rep invariants
elements !=null, the elements container will not be null since it is created with the constructor
size >= 0. The size will not be a negative number. It is not decremented when Queue is empty
size == number of elements in the Queue.
contents of elements will not be null. e is prohibited from being null based on above total contract
3. toString

```
   public String toString() {
        String result= "Type=Queue\n";
        result += "number of elements in Queue: " + size + "\n";
        result += "elements in the Queue: " + elements;
        return result;
   }
```

I printed out the type of object, the size of the Queue and all the elements in the Queue with the inherited toString of an ArrayList.
4. deQueueAll

```
    /**
     * Effects: if Queue is empty throws IllegalStateException
     *          else removes all elements from the Queue and returns them as
a list
```

```
     **/
    public List<E> deQueueAll(){
       if (size==0) throw new IllegalStateException("Queue.deQueueAll");
       List<E> result= new ArrayList<>(elements);
       elements.removeAll(elements);
       size=0;
       return result;
    }
```

5.deQueue immutable

```
   public Queue<E> deQueue (E e) {
        if (size == 0) throw new IllegalStateException("Queue.deQueue");
        Queue<E> result = new Queue<>();
        result.elements = new ArrayList<>(elements);
        result.size = size;
        e = result.elements.get(0);
        result.elements.remove(0);
        result.size--;
        return result;
    }
```

Created a defensive copy of Queue. Copied the elements and size from **this** to the copy, removed the element from the copy and put it into a parameter that could be accessed, decremented the copy's size and returned the modified copy. **this** remains unchanged.

6. clone()

```
public Queue<E> clone(){
        try {
        Queue<E> result = (Queue) super.clone();
        result.elements = elements.clone();
        result.size = size;
        return size;
        }
        catch(ClonenotSupportedError e){
         throw new AssertionError()
}
}
```

Cloned using super method and casted to Queue type, cloned elements and size as mutable state references.

Question 2 Chooser

1. rep invs would be

ChoiceList != null =>choicelist is created with the constructor
Choicelist not empty => choicelist has to be created with at least one object and objects are not removed
Choicelist.size>0 size is based on contents of choicelist and will always be greater than 0
Choices in choiceList != null nulls are not allowed as items in the choicelist

2. contracts

constructor

```
    /**
     * parameters choices - Collection used to create chooser
     * Effects: if choices is empty throw IllegalArgumentException
                Else if any element in choices is null, throw NPE
                 Else if choices is null, throw NPE
     *          else create a new Chooser containing choices
```

```
   public GenericChooser (Collection<T> choices) {
        if (choices == null)  throw new NPE;
        if (choices.isEmpty()) throw new IAE;
        if (choices.contains(null)) throw NPE;
      choiceList = new ArrayList<>(choices);
   }
Choose()
/** effects: returns an object at random from the this**/
No change to code
The choose method is correct because it satisfies the contract and holds true
to rep invs.

Question 3
1. toString()
The toString method will give the entire contents of elements, but only
elements of index < size are valid.  It should use size instead of
elements.length
   @Override public String toString() {
      String result = "size = " + size;
      result += "; elements = [";
      for (int i = 0; i < size; i++) {
         if (i < size-2)
            result = result + elements[i] + ", ";
         else
            result = result + elements[i];
      }
      return result + "]";
2. pushAll gives the user direct insight into the internal structure that is
being used for elements because the user will have to pass an array of
Objects
/** parameter: collection is a an array of objects to be added to the
StackinClass
* effects: adds collection to the StackInClass.
* Modifies: this
3. immutable Class Pop
   public StackInClass pop (Object e) {
     if (size == 0) throw new IllegalStateException("Stack.pop");
     StackInClass result = new StackInClass();
     result.size = this.size;
     result.elements = new Object[2*this.size + 1];
     for (int i=0;i<elements.lenght();i++){
               result.elements[i]= this.elements[i];
       }

     e = elements[result.—size];
     return result;
   }

Made defensive copy of this for result, copied contents of elements and size
to copy, popped the top element to parameter e, and returned the new stack
4. equals with List over Array
Equals would be easier with a list because you do not have to track the size
separately and you could use the List equals method unlike with an array
where you would have to compare each value of the two arrays up to an index
of size

Question 4 Hoare logic
```

1. The program satisfies the specification because y will start of greater than x, which is initially set to 0, the program will then enter the while loop and increment x by 2. It will continue to increment x until either x is incremented to be equal with y or x will be greater than y by 1, at which point the loop will terminate, the program will finish and x will be >= to y meeting the post condition.
2. loop invariants
y>=1 this is a precondition and y is not modified during the loop
x>=0 x starts of at 0 and is incremented until it is greater or = y, which is great than 0
true a loop invariant that always holds
3. WP(while[y>=1] x<y do x=x+2, {x>=y}) =
I && (I && B →WP(S,I)) && (I && !B → Q)
1. y>=1
2. y>=1 & x<y => wp(x=x+2, y>=1)
y>=1
true
3. y>=1 & x>=y => x>=y
True
y>=1&true&true
WP = y>=1
vc
P=>WP(S,Q)
y>=1 => y>=1 == true program is correct
4. WP(while[true] x<y do x=x+2, {x>=y}) =
I && (I && B →WP(S,I)) && (I && !B → Q)
1. true
2. true & x<y => wp(x=x+2, true)
true
3. true & x>=y => x>=y
true
true
WP = true
vc
P=>WP(S,Q)
y>=1=>true
cannot prove program is correct

Question 5
1. correct means it holds rep invariants and satisfies the contract.
(i) correct method

```
/**
  * parameters choices - Collection used to create chooser
  * Effects: if choices is empty throw IllegalArgumentException
             Else if any element in choices is null, throw NPE
             Else if choices is null, throw NPE
  *          else create a new Chooser containing choices
public GenericChooser (Collection<T> choices) {
     if (choices == null)  throw new NPE;
     if (choices.isEmpty()) throw new IAE;
     if (choices.contains(null)) throw NPE;
   choiceList = new ArrayList<>(choices);
```

```
   }
(ii) incorrect method - does not meet contract
   /**
     * parameters choices - Collection used to create chooser
     * Effects: if choices is empty throw IllegalArgumentException
                Else if any element in choices is null, throw NPE
                Else if choices is null, throw NPE
     *          else create a new Chooser containing choices
   public GenericChooser (Collection<T> choices) {
      choiceList = new ArrayList<>(choices);
   }
```

2. rep invariants are things that will not change in an object after creation and when executing methods or operations on the object (an object may never have a collection that is null), where as loop invariants hold true prior to the execution and following completion of the loop (the counter for a loop is <= the loop condition). Pre/post conditions are what need to be satisfied for a method and are defined by the contract to indicate limits of implementation. a precondition puts the onus on the user to calling to meet that requirement (precondition: collection parameter must not be null)

3. Junit theories allows you to provide a data set and it will run all possible combinations of that data set to test your codes whereas Junit test will only test with single set of parameters. For a theory you have to provide the assumeTrue, assertTrue model to set up the right conditions for the test. If the AssumeTrue is not, then that particular combination will not run the assertTrue portion

```
   @DataPoints
    public static Point[] points = {null, new Point(2,2), new
ColorPoint(2,2,COLOR.BLACK),
            new ColorPoint(2,2 ,COLOR.RED)};


    @Theory
    public void testEquals(Object a, Object b) {
        assumeTrue(a!=null && b!=null);
        System.out.println(a + " " + b);
        assertEquals(a.equals(b), b.equals(a));
```

4. proving is done statically by analyzing the code, whereas testing is running the code dynamically to find errors. If you cannot prove it does not mean the program is wrong, merely that you are not certain it is correct

5. LSP substation principle is that an inherited object should act the same way as the parent. Do that if you use a child object in place of the parent you will not get unexpected behavior.

```
Class point {
Int x, y;

Public point(x,y){
This.x=x;
This.y=y;
}

Public void swap(){
X=-x;
Y=-y;
}
}

Class Colorpoint extends point{
```

```
Color color;

Public Color(x,y,c){
Super(x,y);
This.color=c;
}
}
```

Question 6
1. Team members:
Aastha Neupane A
Anum Qureshi A
Saivarun Kandagatla B – got better towards end of class after we spoke with him

Question 7
Least favorite topic was Generics because it was most difficult, but I found it very useful. Hoare logic was second least favorite.
Favorite was contract writing

I felt we could have reviewed the quizzes the following week to go over correct answers. The comments in the grades of quizzes were not very useful.