SWE 619: Final Exam

Srujan Reddy Tekula (G01240653)

1 Question 1

Consider Queue.java.

```
import java.util.*;

public class Queue <E> {

   private List<E> elements;
   private int size;

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

   public void enQueue (E e) {
     elements.add(e);
     size++;
   }

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

   public boolean isEmpty() {
      return size == 0;
   }

  public static void main(String[] args) {
     // Simple exercise to enQueue/deQueue cmd line args
     // Usage:  java Queue item1 item2 item3 ...
     Queue <String> q = new Queue <String>();
     for (String arg : args)
        q.enQueue(arg);
     while (!q.isEmpty() )
        System.out.println(q.deQueue());

  }

}
```

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

➔ The partial contract for enQueue() can be written as follows,
- Pre-Condition: e != null
- Post-Condition: e should be added to the list elements.
- No change in the code

➔ The total contract for enQueue() can be written as follows,
- Pre-Condition: None
- Post-Condition:
  - If e == null, throw IAE
  - Add e to the list elements if e is not null
- Code can be changed as follows,

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

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

➔ The rep invs for this class could be as follows,
- The arraylist, elements should not be null
- The size should be greater than or equals to 0
- The size should not be greater than the size of the arraylist elements

```
elements != null && size >= 0 && size <= elements.size()
```

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

➔ The toString() can be implemented as follows,

```
@Override public String toString() {
      String result = "size = " + size;
      result += "; elements = [";
      for (int i = 0; i < size; i++) {
         if (i < size - 1)
            result = result + elements.get(i) + ", ";
         else
            result = result + elements.get(i);
      }
      return result + "]";
}
```

➔ This code is similar to the toString() used in stack class.
➔ It prints the output as follows,

```
size = 5; elements = [2,3,4,5,6]
```

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

- ➔ Pre-Condition: None
- ➔ Post-Condition:
  - Size > 0
  - List should be dequeued
- ➔ Code can be implemented as follows,

```
public E deQueueAll () {
  if (size == 0) throw new IllegalStateException("Queue.deQueue");
  while(size > 0){
      deQueue();
  }
}
```

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

- ➔ The deQueue() method for an immutable version of this class can be re-written as follows,
- ➔ Created a temporary queue as a new list and to store the values, instead of using the original queue.

```
 public List<E> deQueue(List<E> elements) {
      if (isEmpty()) throw new IllegalArgumentException();
      List<E> tempQ = new ArrayList<>(elements);
      tempQ.remove(0);
      return new ArrayList<E>(tempQ);
 }
```

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

- ➔ Clone() can be implemented as follows,

```
public List<E> clone(List<E> elements){
      ArrayList cloneQ = new ArrayList();
      cloneQ = (ArrayList)elements.clone();
      return cloneQ;
}
```

- ➔ Created a new arraylist as cloneQ and cloned the values of elements to cloneQ

2 Question 2

Consider Bloch's final version of his Chooser example, namely GenericChooser.java.

```
import java.util.*;
import java.util.concurrent.*;

// Bloch's final version
public class GenericChooser<T> {
   private final List<T> choiceList;

   public GenericChooser (Collection<T> choices) {
      choiceList = new ArrayList<>(choices);
   }

   public T choose() {
      Random rnd = ThreadLocalRandom.current();
      return choiceList.get(rnd.nextInt(choiceList.size()));
   }
}
```

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

➔ The pre and postconditions for GenericChooser() can be assumed as follows,
- Pre-Condition: None
- Post-Condition:
  - If choices is null, throw IAE
  - If choices is empty, throw IAE
  - choices.contains(null), throw NPE
  - Create a GenericChooser with choices

➔ The pre and post conditions for choose() can be assumed as follows,
- Pre-Condition: None
- Post-Condition:
  - Returns random choice in list<T>

Considering the above pre and post conditions, a good rep-invariant for this class could be as follows,

```
choiceList != null && size(choiceList) > 0 && !
choiceList.contains(null)
```

or/and

```
choices != null && size(choices) > 0 && ! choices.contains(null)
```

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.

➔ With the post conditions assumed, we can recode the GenericChooser() as follows,

```
public GenericChooser (Collection<T> choices) {
```

```
        if(choices == null) throw new IllegalArgumentException();

        if(choices.size() == 0) throw new IllegalArgumentException();

        if(choices.contains(null)) throw new NullPointerException();

        choiceList = new ArrayList<>(choices);

}
```

Considering our rep-invariant to be correct and pre and post conditions assumed are correct, there is no need to change the choose() method.

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.

- ➔ Considering the rep invariant and the post conditions to be correct for this constructor, they already validate if the input list is not null or size is not zero or doesn't contain any null values in the GenericChooser().
- ➔ So, there is no need to validate these conditions again in choose() method.
- ➔ If, no such validation are made in GenericChooser() considering the post conditions are incorrect of assumed differently, then the choose() method might need to be recoded.

3 Question 3

Consider StackInClass.java. Note of the push() method is a variation on Bloch's code.

```
import java.util.*;

public class StackInClass {
   private Object[] elements; private int size = 0;

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

   public void push (Object e) {
     if (e == null) throw new NullPointerException("Stack.push");
     ensureCapacity(); elements[size++] = e;
   }

   public void pushAll (Object[] collection) { for (Object obj:
collection) { push(obj); } }

   public Object pop () {
     if (size == 0) throw new IllegalStateException("Stack.pop");
     Object result = elements[--size];
     // elements[size] = null;
     return result;
   }

   @Override public String toString() {
      String result = "size = " + size;
      result += "; elements = [";
      for (int i = 0; i < elements.length; i++) {
         if (i < elements.length-1)
            result = result + elements[i] + ", ";
         else
            result = result + elements[i];
      }
      return result + "]";
   }
  private void ensureCapacity() {
      if (elements.length == size) {
         Object oldElements[] = elements;
         elements = new Object[2*size + 1];
         System.arraycopy(oldElements, 0, elements, 0, size);
      }
   }
}
```

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

- ➔ In the For loop condition of toString() and in the IF condition, variable I is compared with elements.length and iterating the entire array which is not required as the arrays is frequently updated with pop() and push() methods.
- ➔ So, instead of iterating entire array with its length, it can be iterated till the top most element present using size.

➔ Considering size instead of elements.length, toString() can be re-coded as follows,

```
@Override public String toString() {
    String result = "size = " + size;
    result += "; elements = [";
    for (int i = 0; i < size; i++) {
       if (i < size-1)
          result = result + elements[i] + ", ";
       else
          result = result + elements[i];
    }
    return result + "]";
 }
```

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

➔ As we know that global variables can be easily violate the encapsulation rules and in this code, pushAll() method uses the push() method which can be easily be overridden
➔ By declaring the push() method with either protected or final will make sure that the push() method will not be overridden.
➔ The contract for pushAll() can be considered as follows,
- Pre-Condition: None
- Post-Condition: If elements.contains(null), throws NullPointerException

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

➔ Keeping the same instance variables, the pop() method can be rewritten for an immutable version of the stack class as follows,
➔ A place holder array stack can be used which stores all the elements of our array stack and returns the new stack instead of the object when pop() is called.

```
public Object[] pop (Object[] elements) {
     if (size == 0) throw new IllegalStateException("Stack.pop");
     Object[] placeHolderArray = elements; // placeholder array
     placeHolderArray [size - 1] = null;
     return placeHolderArray;
}
```

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.

➔ We can use a default equals() method of lists to compare instead of implementing a user defined method for equals() in case of stack array.
➔ The equals() method will not work as expected when the array has a null value as the end element and truncates the size to one less.

4 Question 4

Consider the program below (y is the input).

```
{y ≥ 1} // precondition

x := 0;
while(x < y)
      x += 2;

{x ≥ y} // post condition
```

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

➔ Yes, it satisfies the given specification because of the following,
- Pre-Condition is y>=1.
- Let us consider that y = 5 and given x = 0;
- The while condition satisfies as 0 < 5
- The program enters inside loop and x iterates to 2…4 and 6 in three iterations util while condition fails.
- So after the loop, the value of x is 6. Hence, x >= y
- So, it satisfies the specification provided by the program.

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

➔ The three loop invariants are as follows,
- x>=0 : As x is initialised with 0, it can be equal to zero, and if the condition satisfies, irrespective of y value x will be greater than 0 as it will be incremented by 2.
- y>=1 : As per the pre-condition it has to be satisfied that y >= 1 and it is not changed in the code
- True : The loop will trigger only when the condition is true, in any other condition the loop action will not trigger

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.

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.

5 Question 5

Note: you can reuse your answers/examples in previous questions to help you answer the following questions.

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.

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:

➔ Rep invariant captures the rules that are required for reasoning the data structure declared

For generic chooser class, the rep invariant is,

```
choiceList != null && size(choiceList) > 0 && !
choiceList.contains(null)
```

Loop Invariant:

➔ A condition or statement that should be true in order to enter the loop and will be validated after every iteration. If the condition still holds true, it will enter the loop until the condition holds false.

```
{y >= 0} //pre
Y = 8;
x = 0;
while(x < y)
      x += 2;
{x >= y} //post
```

➔ Here, consider the value of y as 8 and x as 0

➔ The loop invariant is as follows,

- True
- x <= 0
- y >= 0

➔ So, until the value of x is >= y, the while loop iterates.

Contract:

➔ They are the pre and post conditions that are to be satisfied in order for the function to be considered without any violation. The contract refers to the publicly exposed field or functions of the class.

For the generic chooser class, the pre and postconditions for GenericChooser() can be assumed as follows,

- Pre-Condition: None
- Post-Condition:
  - If choices is null, throw IAE
  - If choices is empty, throw exception
  - choices.contains(null), throw exception
  - Create a GenericChooser with choices

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

- ➔ In the standard Junit tests, we generally provide the finite tests with test data based on few assumptions and try to test with them.
- ➔ If the inputted test data is true, according to few assumptions, the result is always deterministic.
- ➔ But in case of JUnit theories, you would be able to supply an infinite test data.
- ➔ As per my understanding, Junit theories creates all possible combinations of the inputted test data.

- ➔ Example:
  - If for code which needs to be tested requires 5 parameters and user generates for 6 test cases,
  - Standard JUnit tests will generate only 6 tests that are provided
  - JUnit Theories will generate a total of 30 (5*6) JUnit tests.

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 verifies program for all possible executions but testing validates only for the specified execution. Though proving is not practical, it tells us to think about program correctness and its importance.
- ➔ If we cannot prove (e.g., using Hoare logic) then the program doesn't do anything.

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.

- ➔ LSP is a principle in object-oriented programming which states that the object and the sub-object must be interchangeable without violating the programs rules.
- ➔ **Example:**

```
class TrasportationDevice
{
   void startEngine() { ... }
}

class Car extends TransportationDevice
{
   @Override
   void startEngine() { ... }
}
```

- ➔ There is no issue here, as car is definitely a transport device and engine could be started
- ➔ Let us add an another transport device.

```
class Bicycle extends TransportationDevice
{
   @Override
   void startEngine() /*problem!*/
}
```

➔ Here, though bicycle is a transportation device, it is not possible to start and engine

Instead, the above example can be re-coded as follows,

```
class TrasportationDevice
{
   ……
}

class DevicesWithoutEngines extends TransportationDevice
{
   void startMoving() { ... }
}

class DevicesWithEngines extends TransportationDevice
{
   void startEngine() { ... }
}

class Car extends DevicesWithEngines
{
   @Override
   void startEngine() { ... }
}

class Bicycle extends DevicesWithoutEngines
{
   @Override
   void startMoving() { ... }
}
```

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?

➔ Our group members are: Rachana Thota, Srikar Reddy Karemma, Tuljasree Bonam and Me (Srujan Reddy Tekula)

2. For each group member, rate their participation in the group on the following scale: (a) Completely absent (b) Occasionally attended, but didn't contribute reliably (c) Regular participant; contributed reliably

➔ (c) All of our group members are regular participants and contributed reliably throughout all the assignment, reading reflections and in-class exercises.