FINAL EXAM – SWE 619

Rachana Thota
G01237600

Answer 1:

1. Contract that has precondition is partial contract.
   Contract that has no precondition is total contract.

   Partial Contract:
   Pre-condition: Element e should not be null.
   Post-condition: Elements gets added into the list.

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

   Total Contract:
   Pre-condition: No pre-condition
   Post-condition: If the element e is null, throws IllegalArgumentException else elements get added into the list.

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

2. **Rep invs:**
   (i)elements!=null
   The arraylist elements should not be null.
   (ii) 0<=size<=elements.length
   The size variable should be in between 0 and length of the element list(inclusive).

3. ToString() implementation:

```
@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 + "]";
}
```

   In the toString() method, I am printing the size of the queue and the elements stored in the queue. Result variable is taken to print the size and for the elements, for loop is written which iterates the list and prints all the elements present in the queue. If-else condition is used to append comma after each element. In this way, it returns the result in string format.

4. **Contract:**
   Pre-condition: size should be greater than 0.
   Post-condition: size should be equal to 0.

```
public void deQueueAll(){
      while(size>0){
            deQueue();
      }

}
```

In deQueueAll() method, it calls dequeue method iteratively till size becomes zero.

5. **Immutable version of dequeue:**

```
public List<E> deQueue(List<E> elements) {
    if(isEmpty()){
          throw new IllegalArgumentException("Queue is Empty!!");
    }
    List<E> elements1= new ArrayList<E>();
    elements1.addAll(elements);

    elements1.remove(0);
    return new ArrayList<E>(elements1);

}
```

In immutable version of Queue implementation, the original list named elements is loaded into new array list named elements1. Then the first element is removed by using elements1.remove(0) and finally it returns new updated array list i.e., elements1. In this way the original list elements won't change.

6. **Clone Implementation:**

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

}
```

In clone implementation, I have initialized an empty array list dummyQueue and cloned all the elements from the original array list elements. Finally, the method returns the dummyQueue which is the copy of original array list.

Answer 2:

1. The good rep invariants for this class are:

   Rep invs: choiceList !=null && size(choices) > 0
   In the above code, the created array list i.e., choiceList should not be null and it should not be empty.

2. For constructor,

   Pre-condition: None
   Post-condition:
   (i) If choices are null, it should throw IllegalArgumentException.
   (ii) If choices are empty, it should throw IllegalArgumentException.
   (iii) choices should not contain null else it will be problem in choose() method, therefore we need to handle with exception.
   (iv) creating a choiceList with choices.

   For choose() method,

   Pre-condition: None
   Post-condition:
   (i) Returns random choice from the arrayList i.e., choiceList.

   The above-mentioned pre-condition and post-condition are based on the rep-invariants mentioned in the first part. To satisfy the above postcondition, the code needs to be modified as follows:

```
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) {

        if(choices.size() == 0){
                throw new IlegalArgumentException();
        }
        if(choices == null){
                throw new IlegalArgumentException();
        }

        if(choices.contains(null)){
                throw new IlegalArgumentException();
        }

        choiceList = new ArrayList<>(choices);
```

```
}

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

3. choose() method returns a random choice from list. From the modified code, I have taken care of all the conditions that might be problem for choose() method. In the constructor, I have added all the conditions that choiceList cannot be empty, choiceList should not be null and it should not contain any null element. As all these conditions are handled in the constructor, there is no need of any modification in choose() method and also it has no problem in returning the random element. Therefore, choose() method which is documented above is correct.

Answer 3:

1. **Problem:** In toString() method, it is printing all the array elements but we need elements up to size. In the push() and pop() methods we are updating the size but not removing the elements from the array but in toString() we are iterating whole array and outputting it instead we need to print only the elements that are stored in the stack.
   Fix: Replacing elements.length with size.

   //modified code

```
@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. Encapsulation mentions that the data should be accessed only by required set of elements. If the method is public, it can be accessible everywhere, also it is prone to changes. It doesn't hide the internal working of the program. pushAll() method depends on push() method but push() method is public and there is chance of overriding it through inheritance. Then, pushAll() will be affected. Therefore, it violates encapsulation principle as it is getting accessed through other classes. We can make push() method as final, then we cannot modify push method and also it satisfies encapsulation.
   The contract for pushAll() method is:
   Contract:
   Pre-condition: If any element is null, raise NullPointerException.
   Post-condition: Add everything to this (or the stack).

3. To make immutable version of pop() method, we need to create a new object array which stores all the elements from the existing object array and remove an element from new object array and return it back. In this way, we will not be changing original object array and will return every time a new object array from the pop() method.

```
public Object[] pop (Object[] elements) {
        if (size == 0) throw new IllegalStateException("Stack.pop");
        //creating a new object array and storing elements
        Object[] new_object = elements;
        //decrementing the size by 1
        int new_size = size-1;
        //setting new_object[new_size] to null
        new_object[new_size] = null;
        return new_object;
}
```

   The existing pop() method was returning the top element. We need a new method which returns element at the top.

4. Using list would be easy as it is dynamic and there is no need of implementing equals() method, we can directly use inbuilt .equals() method. In case of arrays, equals() method checks the array length but here we need to check the stack size so it would be difficult to use .equals() method instead we need to implement it. For Example, if arr1 has [‘dog’,null] and arr2 has [‘dog’], according to the logic size for both of them should be 1 but in arrays it returns different sizes. So we cannot use inbuilt .equals() method.

Answer 4:

1. It satisfies the given pre-condition. We can explain this through substitution, in the precondition y>=1 if we assume y as 1 with the given x=0. It satisfies the while loop condition i.e., 0<1 and x increments by 2 which means x becomes greater than y which satisfies the post condition. Let it be any value, the post condition satisfies because in loop x always increments by 2 which eventually becomes x>=y. Therefore, the program satisfies the given specification.

2. **Loop Invariants for the Program are:**

   Loop Invariant: **x>=0** (x is greater than or equal to 0 holds because initially it is initialized to zero and then it increments further but does not decrease)
   Loop Invariant: **y>=1**(y is always greater than 1 as it is the pre-condition and we are not changing y in the loop).
   Loop Invariant: **TRUE** (loop iterates until the condition is true)

3. According to the Hoare Condition,
   WP(while[I] B do S , {Q}) =
   1. I
   2. (I & B) => WP(S,I)
   3. (I & !B) => Q

   For loop invariant 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)
   X<y => x+2>=0
   X<y => True
   True

3. (x>=0 & !(x<y)) => x>=y
   x>=y => x>=y
   True

Therefore x>=0 && True && True
We get, x>=0
WP(x:=0,{x>=0}) = 0>=0.

Verification Condition:
vc (P => wp(..)) P => WP(x := 0; while[x>=0] x<y do x+=2, {x>=y}) =
P => x>=0 (True)
Hence Proved.
Therefore, it is sufficiently strong loop invariant.

4.

As per the Hoare logic, for loop invariant y>=1

. WP(x:=0; 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) => y>=1
   False(cannot simplify further)

3. y>=1 & !(x<y) => x>=y
   y>=1 & x>=y => x>=y
   False

The weak pre-condition for while loop is false for y>=1 loop invariant therefore WP of entire program is false

WP(x:=0; WP{False}) = WP{False}

Verification condition:
vc (P => wp(..))
P => False

y>=1 => False
False

As we cannot determine or prove the validity of Hoare Triple, it is insufficiently strong.

Answer 5:

1. **(i) Program:**
   {N>=0}
   ```
   i:=0;

   while(i<N){
         i:=N;
   }
   ```
   { i == N}

   Loop Invariants for the above program is:

   Loop Inv: i<=N
   Loop Inv: i>=0
   Loop Inv: N>=0
   Loop Inv: True

   According to the Hoare Condition,
   WP(while[I] B do S , {Q}) =
   4. I
   5. (I & B) => WP(S,I)
   6. (I & !B) => Q

   Using i<=N as loop invariant to prove correctness of the program,

   WP(while[i<=N] i<N do i:=N, {i==N})
   1. i<=N
   2. (i<=N & i<N) => WP(i:=N, {i<=N})
      i<N => N<=N
      i<N => True
      True
   3. i<=N & !(i<N) => i == N
      i == N => i == N
      True

      It holds for i<=N, therefore the program is correct.

(ii) An example showing the program is incorrect:

```
{N>=0}
i:=0;

while(i<N){
      i:=i+1;
}
```

{ i == N}

Loop Invariants for the above program is:

Loop Inv: i<=N
Loop Inv: N>=0
Loop Inv: True

According to the Hoare Condition,
WP(while[I] B do S , {Q}) =
1. I
2. (I & B) => WP(S,I)
3. (I & !B) => Q

Using i<=N as loop invariant to prove correctness of the program,

WP(while[i<=N] i<N do i:=i+1, {i==N})
1. i<=N
2. (i<=N & i<N) => WP(i:=i+1, {i<=N})
   i<N => i+1<=N
   False(cannot simplify further)
3. i<=N & !(i<N) => i==N
   False(as we cannot simplify)

It does not hold as we cannot simplify further so above program is incorrect.

2.

Loop Invariant: A statement that should be true when entering a loop and after every iteration of the loop.
Rep Invariant: capture rules that are essential in reasoning about data structure and its correctness.
Contract/specifications: They are the pre-conditions and postconditions of the given program. Pre-condition are the required conditions that are assumed to be true and post-conditions are the effects of the program.

Example:

```
public Class Chooser<T>{
```

```
private final List<T> choiceList;

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

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

As above code is linked with data structure, we can find rep invariant and write the contract (pre-condition and post-condition) in such a way that it satisfies the rep invariant which is given.
Rep invs: choiceList!=null && size(choices) >0
Contract:
Pre-condition: None
Post-condition: if choices is null, throws IAE
If choices is empty, throws exception
If choices contain any element null, throws exception
Create a chooser with choices
For loop invariant, need to write a loop which should be true when entering a loop and after every iteration of loop:

```
i:=0;

while(i<N){
      i:=N;
}
```

Loop Invariants for the above program is:

Loop Inv: i<=N
Loop Inv: i>=0
Loop Inv: N>=0
Loop Inv: True

Therfore, all these are involved with checking the correctness of the program. Loop Invariants gives all the conditions that proves before entering and exiting the loop,rep invariant checks the correctness of the data structure and the contract gives the pre-conditions and post-conditions of the program.

3. Junit supports functionality through parameterized tests. In Junit, the test data elements are statically defined and as you as a programmer are responsible for figuring out what data is needed for a particular range of tests. In Junit Theory, it executes a theory against several data inputs called data points

For Example:

```
@Theory
public void testCreateEmailID(String firstPart, String secondPart) throws
Exception {
      String actual= EmailIdUtility.createEmailID(firstPart,secondPart);
```

```
        assertThat(actual,is(allOf(containsString(firstPart),
        containsString(secondPart))));
}
```

Here, the createEmailID() method accepts two String parameters and generates email ID in the specific format. A test theory is established for "Provided stringA and stringB passed to createEmailID() are non-null, it will return an email ID containing both stringA and string" -> the test theory for this is testCreateEmailID() is implemented which accepts two String parameters. At run time, the **Theories** runner will call testCreateEmailID() passing every possible combination of the data points we defined of type **String**. For example (*mary,mary*), (*mary,first*), (*mary,second*), and so on.
So, the benefit of Junit theory is we can test the method through various combination of data points. For Junit, we can give inputs of our own and assert it with the output retrieved from the method and it depends on programmer assumption.

4. In testing, observable properties are determined. It verifies program for one execution, and it is the most practical approach. On other side, in Proofs any program property is determined. It verifies programs for all the executions. It may be practical for small programs in 10-20 years.

   According to Hoare Logic, the formal reasoning about program correctness using pre-conditions and post-conditons are:

   Syntax: {P} S {Q}
   Where, P and Q are predicates and S is a program.

   If we start in a state where P is true and execute S, then S will terminate in state where Q is true.

   If the program cannot prove the correctness through hoare logic, we can do nothing.

5. **Liskov Substitution Principle:**
   It is a concept in Object Oriented Programming Language which states that the function that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
   Example:

```
public class BookDelivery{
      String title;
      Integer userID;
}
public class OfflineDelivery extends BookDelivery{
      void getDeliveryLocations(){
      ..
      }
}
public class OnlineDelivery extends BookDelivery{
      void getSoftwareOptions(){
      ..
      }
}
```

The OfflineDelivery and OnlineDelivery classes split up the BookDelivery superclass. We will also move the getDeliveryLocations() method to OfflineDelivery. Next, we will create a new getSoftwareOptions() method for the OnlineDelivery class (as this is more suitable for online deliveries). In the refactored code, PosterMapDelivery will be the child class of OfflineDelivery and it will override the getDeliveryLocations() method with its own functionality.This demonstrates the Liskov Substitution concept.

Answer-6:

My group members and their participation:
Tuljasree Bonam - Regular participant; contributed reliably
Srikar Reddy Karemma - Regular participant; contributed reliably
Srujan Reddy Tekula - Regular participant; contributed reliably

Answer-7:

1. What were your favorite and least aspects of this class? Favorite topics?
   My fav class and topics are Hoare Logic, Junit Theory.
2. Favorite things the professor did or didn't do?
   The best thing I liked is collaboration with teams.
3. What would you change for next time?
   Everything is good, no need to change anything.