SWE 619 Final- Suraj Varma Guntumadugu

1)Question 1:

Here I am considering the given Queue.java program

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

A. (i)Partial Contract: By definition, contract that has precondition is called as Partial Contract.

```
//Requires: Element e should not be null
//Modifies: size gets incremented.
//Effects:  Elements get added into the enQueue

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

(ii)Total Contract: By definition, contract that has no precondition is called as Total Contract

```
//Requires: No pre condition
//Modifies: Increment the size.
// Effects: If the element e is null, throw NullPointerException or else add the elements into the enqueue

public void enQueue (E e) {
  if(e==null)
{
throw new NullPointerException("elements should not be null")
}
elements.add(e);
    size++;
  }
```

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

A. The following are the two rep Invariants which I found for the given program.

(i) Elements e!=null – The internal implementation of elements cannot not be null
(ii) 0 <= size <= elements.length() – To keep away the possibility of adding the scrap or junk and the null elements into the list and the size should be non-negative.
(iii) If size>0then elements from 0 to size should not be null.

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

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[i] + ", ";
      else
        result = result + elements[i];
    }
    return result + "]";
```

A. In the toString() method, here I am trying to print the size of the queue and also the elements or the values which are stored in the queue. It prints all the elements which are in the queue and as I have used If-else condition, the elements will be printed by having a , and also be printed in string format.

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

A.

The reasonable Contract for the below considered deQueueAll() method is

//Requires: size>0

Because size cannot be negative and zero as atleast we need one element or more elements to perform this dequeue.

Effects: size=0, size should become zero.

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

}
```

The prominent function of deQueueAll is to iteratively repeat the loop until it becomes zero.

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

```
public List<E> deQueue(List<E> e) {
if(isEmpty()){
        throw new IllegalArgumentException("Queue is Empty");
}
if(Null){
        throw new NullPointerException("Queue is Null");
}

List<E> e1= new ArrayList<E>();
e1.addAll(elements);

e1.remove(0);
return new ArrayList<E>(e1);

}
```

In the above immutable version of Queue implementation, the actual list of elements e is given into new array list e1 and it get added by e1.addAll and then the first element is removed by using e1.remove(0) and finally it returns new updated array list i.e., e1. So the original list of elements does not change.

6. Write a reasonable implementation of clone(). Explain what you did.
   A.
```
public List<E> clone(List<E> e){
List<E> dummyQueue = new ArrayList<>();
dummyQueue = (ArrayList) e.clone();
return dummyQueue;

}
```

Object cloning refers to the creation of an exact copy of an object. It creates a new instance of the class of the current object and initializes all its fields with exactly the contents of the corresponding fields of this object. Here In this clone implementation, I have initialized an empty array list dummyQueue and cloned all the elements from the original array list. And at final it returns the dummyQueue value which is an copy of them.

2)Question 2

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

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

A. The Rep invs for the given Class are
choiceList !=null and
size(choices) > 0
choices.contains(null) == false or choices.contains(null) != true
In the given program the choiceList should not be null and the size(choices) should be greater than zero as the size cannot be negative or zero for a chooseList kind of method.

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.

A. Here for the suitable contracts for the constructor, I am choosing a total contract scenario, where the
//Requires : No pre condition
// Effects:
a) If size = 0 throw IllegalArgumentException
b) If choices are null throw NullPointerExxception
c) If choices are empty throw IllegalArgumentException
d) Choices should not contain null and if it contains so, throw IllegalArgumentException.

Even for the Choose() method I am considering the total contract scenario:

//Requires : No pre condition
// Effects:
a) Returns the random choiceList from the arrayList of Collections and returns the size.

The above-mentioned contracts are consistent with the previous question by effecting the size, empty list and checking of null constraints. The below is the code which I have changed according to the contracts I have mentioned above:

The code changes are in the public GenericChooser class

```
if(choices==null){
        throw new NullPointerException("Cannot be Null")
}
if(choices.size()==0){
        throw new IllegalArgumentException("Size cannot be zero")
}
if(choices.contains(null)){
        throw new IllegalArgumentException("Cannot be Null")
}
if(choices.isEmpty==true){
        throw new IllegalArgumentException("Cannot be Empty")
}
```

There won't be any changes in the choose method. Every change will be happened at the constructor class itself.

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.

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

The above mentioned is the choose() method which returns the random choice of elements from the list. The functionality of this would be like, it chooses the elements from the chooseList.size() and selects one of the random size and then uses the choiceList.get function to return an random element from the choice of list. We do not require to make any changes in the choose method and instead I have implemented changes in the constructor so that the changes made such as the size cannot be zero, choiseList cannot be null and choice.contains should be null. Choose() method uses these changes and returns the random element without any problem.

3)Question 3

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

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

A. The issue with the toString() method here is that it is printing complete array elements but we require to print elements upto the size itself. In toString() method few are iterating complete or the whole elements array and so we need to print the elements that are only stored in the stack.

So the fix we need to implement here is to replace elements.length with size.
The modified code after implementing the replacement is 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().

A. Firstly, by definition Encapsulation means mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Here in this scenario pushAll() depends on the push() method and whereas push() method overhere is public and so there are chances of overriding the method through inheritance. Then pushAll() will get effected at the runtime. Hence, it violates the encapsulation principle as it is accessed through other classes even. SO to overcome this we need to make push() method as final or the private.

The contract for the pushAll() method is:
//Requires: All elements are non-null
//Effects: If any element is null, then raise exception, otherwise , add everything to this or to the stack.

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

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

A.

Immutable pop() method implementation can be as follows,
We have to create a new object and store all the elements from the old object into the new one. Now we will perform delete operation and then return the new object created.
Below is the implementation,

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

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.

A. Yes, Implementing the equals( ) method for the given class is bit a messy thing and by replacing the array with the List it would be bit better. As the list is dynamic in nature. As far as arrays are concerned, equals() method checks the array length but in this scenario we need to check the stack size so it would be difficult to use .equals() method to implement and instead it's preferrable to use List.

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).

A. Yes, it satisfies the given specification because firstly the pre condition is y>=1. So, for instance let's consider y as 1 and given that x=0. So, it satisfies the while loop (0<1) and so the x becomes 2. Finally which satisfies the post condition which is x>=y i.e., 0>=1. So, it's proved that the specification is satisfied 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.

A. The 3 loop invariants I have considered for the while loop overhere is :

x>=0

y>=1

x<=y

TRUE

a) Firstly, the definition of loop invariant is, it is a property of a program loop that is true before and after each iteration. x>=0 holds as a loop invariant because, before entering into the loop x=o and the precondition says that y>=1. And after executing the while loop x<y and the post condition says x>=y which matches purely.

b) y>=1 holds as a loop invariant because, the simplest way to reason here is that we never change the value of y in the whole scenario. Firstly, the precondition holds this invariant and carries down to the while loop where we don't alter the y value and so it holds.

c) x<=y holds as a loop invariant because, before entering into the loop x=o and the precondition says that y>=1. And after executing the while loop we get x=y and the loop invariant says x<=y which matches purely.

d) True holds for the pre condition and carries all way to while loop and doesn't alter anything and finally executes to the post condition.

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.

A. WP(x:=0; while[x<=y] x < y do x+=2, {x> =y}) =
WP(x := 0; WP(while[x<=y] x < y do x+=2, {x> =y}) =
// WP(while[x<=y] x<y do x+=2, {x>=y}) =
1. x<=y

2. (x<=y & x<y) => WP(x+=2, {x<=y})
By simplifying it further, we get
x<y => x+2<y
x<y => x<y (True)

3. x<=y & !(x<y) => x>=y
We can rewrite this as
x<=y & x>=y => x>=y
x=y => x=y (True)

a) x<=y && True && True
we get x<=y

b) WP(x:=0; {x<=y})=
By substituting x we get
y>=1=> 1<=y

c) Compute the verification condition
vc (P => wp(..)) P => WP(x := 0; while[x<=y] x<y do x+=2, {x>=y}) =
P => 1<=y =
y>=1 => 1 <= y (True)

d) Thus using this loop invariant we can prove the validity of the Hoare triple and this is sufficiently strong.

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.

A. WP(x:=0; while[y>=1] x < y do x+=2, {x> =y}) =
WP(x := 0; WP(while[y>=1] x < y do x+=2, {x> =y}) =

// 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) => y>=1 (False)
   We Cannot simplify this further.
   Which is false

3. y>=1 & !(x<y) => x>=y
   We can rewrite this as
   y>=1 & x>=y => x>=y (False)

(i)So the WP of the while loop is False, and therefore the WP of the entire program is also False as shown below:

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

(ii)Compute the verification condition:
vc (P => wp(..))
P => False
y>=1 => False
False

(iii)Analyzing the vc to determine whether the program is proved or not
Thus, using this loop invariant we cannot determine or prove the validity of Hoare Tripple. So this insufficiently strong.

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.

A. The basic definition of the correctness of the program depends on the specifications of the contracts i.,e for a given pre condition the statement should hold true and the post condition should be executed
{P} ->S -> {Q}
And if this doesn't happen the program would be considered as incorrect.

For Example: Given the below program:

{N >= 0} // precondition

i := 0 ;

while(i < N)
  i := N;

{i == N} // post condition

possible loop invariants

- i <= N
- i>=0
- N >= 0
- TRUE

but need to use a loop invariant that is sufficiently strong, otherwise we cannot prove the program (so unlikely we want to use TRUE, even if it is a loop invariant).
Consider i<=N
WP(i := 0; while[i<=N] i < N do i:= N, {i == N}) =
WP(i := 0; WP(while[i<=N] i < N do i:=N, {i == N}) =
 // 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

= i <= N && True && True
= i <=N

WP(i := 0; {i<=N}) =
0<=N
Creating and checking verification condition vc

P => WP(i := 0; while[i<=N] i < N do i:= N, {i == N}) =
P => 0<=N =
N>=0 => 0 <= N # True
Thus using this loop invariant we can prove the validity of the contract or the program and can conclude that the program is correct.

(ii) Example for program to be incorrect:
WP(while[N >= 0] i < N do i:=N, {i == N}) =
1. N >= 0
2. (N >=0 & i < N) => WP(i := N, N >= 0) =
   (N >=0 & i < N) => i >= 0

3. N >=0 & !(i<N) => i ==N
  (N >= 0 & i >= N) => i == N
  i>= 0 => i == N
  False

= N >=0 & (N >=0 & i < N) => i >= 0 & False
= False
So the WP of the while loop is False, and thus the WP of the entire program is also False as shown below

WP(i:=0; {False}) =
False
The vc is then

P => False
N >= 0 => False
False (Hence, this 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.

A. **Rep Invariants**: The rep invariant specifies legal values of the representation, and should be checked at runtime with checkRep() . The abstraction function maps a concrete representation to the abstract value it represents. And in other words, Exposing the rep means to violate the rule that an object's methods control its state. For example, if an object has instance variables and mutators to change their values, then the object has a means of controlling its state.

Loop Invariant: A loop invariant is a condition [among program variables] that is necessarily true immediately before and immediately after each iteration of a loop.

Contract/Specifications: There will be pre conditions and post conditions for a specified scenario or the program. And for the given pre condition if the state holds perfectly and executes or effects the post condition is what we call as contract.

Examples for each are as follows:
For repInvariant I am considering the program of question 2 of this exam:

The Rep invs for the given Class are
choiceList !=null and
size(choices) > 0
choices.contains(null) == false or choices.contains(null) != true

Loop Invariants example:
{N >= 0} // precondition

i := 0 ;

while(i < N)
i := N;

{i == N} // post condition

possible loop invariants

- i <= N
- i>=0
- N >= 0
- TRUE

Contract/Specification:
I am considering the 1st question program of enQueue method

```
//Requires: Element e should not be null
//Modifies: size gets incremented.
//Effects:  Elements get added into the enQueue

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

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

   From what I understand: With Junit tests you can supply a series of static inputs to a test case.

   Theories are similar but different in concept. The idea behind them is to create test cases that test on assumptions rather than static values. So if my supplied test data is true according to some assumptions, the resulting assertion is always deterministic. One of the driving ideas behind this is that you would be able to supply an infinite number of test data and your test case would still be true; also, often you need to test an universe of possibilities within a test input data, like negative numbers. If you test that statically, that is, supply a few negative numbers, it is not guaranteed that your component will work against all negative numbers, even if it is highly probable to do so. So these are couple of benefits using Junit theories compared to standard Junit test cases.

   ```
   @RunWith(Theories.class)

   public class AdditionWithTheoriesTest {
   ```

```
  @DataPoints
  public static int[] positiveIntegers() {
      return new int[]{
                    1, 10, 1234567};
  }

  @Theory
  public void a_plus_b_is_greater_than_a_and_greater_than_b(Integer a, Integer
b) {
     assertTrue(a + b > a);
     assertTrue(a + b > b);
  }
}
```

The same thing would not be more effective the junit tests when compared with Junit theory.

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)?

   When you test, you are making sure that the product or system works but only after you have created the product components, chips, boards, packages and subsystems.

   When you verify or prove, you are making sure that the product or system works before it has been created or manufactured. In systems engineering, you verify to make sure that what you designed actually meets the requirements.

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.

A. 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 .

For instance, consider the below example for LSP.

```
class A {
  public Iterator compose (Iterator itr)
  // Requires: itr is not null
  // Modifies: itr
  // Effects: if this is not appropriate for itr throw IAE
  // else return generator of itr composed with this
class B {
  public Iterator compose (Iterator itr)
  // Modifies: itr
  // Effects: if itr is null throw NPE
  // else if this is not appropriate for itr throw IAE
  // else return generator of itr composed with this
class C {
  public Iterator compose (Iterator itr)
  // Modifies: itr
  // Effects: if itr is null return iterator equal to this
  // else if this is not appropriate for itr throw IAE
  // else return generator of itr composed with this
```

Analyze the compose() method in each of these cases. For each case, state if the precondition and the postcondition parts are OK or fail, and justify.

1) B Extends A- OK

Precondition Rule: presuper => presub

Postcondition Rule: (presuper && postsub) => post super

Sub : B

Super : A

Pre condition: Ok (Because B has the no pre condition which can be considered as weakest Pre condition)

Post Condition: Ok(Because, the post condition of B is stronger as it is even justifying to throw NPE if the itr is null)

Properties Rule:

2) C Extends A: OK

Precondition Rule: presuper => presub

Postcondition Rule: (presuper && postsub) => post super

Sub: C

Super: A

Pre Condition: Ok (Because C has the no pre condition which can be considered as weakest Pre condition)

Post Condition: OK(Because, The post condition of C is stronger as it returning some value even if it is null)

3) A Extends B: Fail

Precondition Rule: presuper => presub

Postcondition Rule: (presuper && postsub) => post super

Sub:A

Super: B

Pre Condition: Fail(Because, B has the no pre condition which means it is considered as weaker pre condition)

Post Condition: Fail(Because, B has the stronger post condition as it is justifying to throw NPE if the itr has null value.)

4) C Extends B: OK

Precondition Rule: presuper => presub

Postcondition Rule: (presuper && postsub) => post super

Sub: C

Super: B

Pre Condition: OK (Because, both have no pre conditions)

Post Condition: OK(Because, C has the strongest post condition as it returns some value even if it is null but B throws an exception if the itr is null)

5) B Extends C: Fail

Precondition Rule: presuper => presub

Postcondition Rule: (presuper && postsub) => post super

Sub: B

Super: C

Pre Condition: Ok (Because, both have no pre conditions)

Post Condition: Fail (Because, B has the weaker post condition as it throws NPE if itr is null where as C returns some value if the itr is null, hence C has stronger post condition and so it fails.)

6)Question 6

1. Who are your group members?

A. Team 3

- Hussain Rohawala - c
- Jagadish Ramidi- c
- Suraj Varma Guntumadugu- c
- Prabhath Surya- c

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

Everyone in the group are so supportive and contributed well.

7)Question 7

1. What were your favorite and least aspects of this class? Favorite topics?

A. Almost all topics were pretty interesting and the one I liked the most is verification or Hoare Tripple.

2. Favorite things the professor did or didn't do?

A. Almost every aspect is pretty good. No issues with the course.

3. What would you change for next time?

A. The present curriculum is pretty good and no changes are required.