**SWE619–Fall21: Final Exam**

**Your Name**: Saivarun kandgatla

**Gnumber:**G01329543

## 1 Question 1:

**Consider Queue.java.**

**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**

**Answer:**

Pre-condition: partial contract.

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

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

There are no pre-condition is total contract.

Total Contract:
Pre-condition: No pre-condition.
Post-condition: The element 'a' is null, throws an IllegalArgumentException elements are added to the list if they are not already there.

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

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

**Answer:**

Rep invariants:
(i)elements!=null
There should be no null elements in the arraylist..

(ii) 0<=size<=elements.length

The size variable should be between 0 and the element list's length (inclusive).

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

**Answer:**

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 + "]";
}
```

I am printing the sizes and the elements stored in the queue.

The size is shown using the result variable, and the elements are printed using a for loop that iterates the list and prints all of the elements in the queue.

To put a comma after each element, an if-else condition is needed. It returns the result in string format this way.

**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**

**Answer:**

Pre-condition: size  greater than 0.
Post-condition: size  equal to 0.

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

The deQueueAll() method iteratively executes the dequeue method until the size is zero.

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

**Asnwer:**

```
public List<E> deQueue(List<E> elements)

{

if(isEmpty())

{

throw new IllegalArgumentException("Queue is Empty!!");

}

List<E> elements2= new ArrayList<E>();

elements2.addAll(elements);

elements2.remove(0);

return new ArrayList<E>(elements2);

}
```

The original list named elements is imported into a new array list named elements2 in the immutable version of Queue implementation.

Then the first element is removed by using elements2.remove(0) and finally it returns new updated array list example elements2.

In this way the given list elements won't change.

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

**Answer:**

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

I created an empty array list called dummyQueue and cloned all of the elements from the original array list.
Then the method returns dummyQueue which is the original array list copy.

**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.**

**answer:**

Rep invariants: choiceList !=null && size(choices) > 0
The created array list example the choiceList should not be null and empty.

**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.**

**Answer:**

Constructor method,

Pre-condition: None
Post-condition:

1)Choices are null and empty it should throw an IllegalArgumentException.
2)Because null will create problems in the choose() method, we must address the situation with an exception.putting together a choiceList.

Choose() method,

Pre-condition: None
Post-condition:

1) Returns random choice from the arrayList example 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 red = ThreadLocalRandom.current();
return choiceList.get(red.nextInt(choiceList.size()));
}
}
```

**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 quest**

**Answer:**

Choose() method returns a random choice from list.
I've taken care of all the conditions that could cause an issue for the choose() method in the changed code. In the constructor, I've wanted to make sure that the choiceList can't be empty, that it can't be null, and that it can't include any null elements.
Because all of these criteria are handled in the constructor, there is no need to change the choose() method, and returning the random element is also no problem.
choose() method which is documented above is correct.

**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.**

**Answer:**

Problem:
The toString() method prints all of the array elements, but we only need those that are up to size.
We update the size of the array but not the elements in the push() and pop() methods, but in toString() we iterate through the entire array and output it, when we only need to print the elements that are stored in the stack.
Fix:
Replacing elements.length with size.

```
@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().**

**Answer:**

Only the required set of elements should be able to access the data, according to encapsulation.

If a method is open to the public, it can be used by anybody and is subject to change.
It does not hide the program's internal workings.
The pushAll() function is dependent on the push() method, however push() is public and can be overridden through inheritance.
PushAll() will be affected as a result. As a result, because it is accessed through other classes, it violates the encapsulation principle.
We can make the push() method final, which prevents us from changing it and ensures encapsulation.
The following is the contract for the pushAll() method
Contract:
Pre-condition**:** If any element is null use NullPointerException.
Post-condition: Add everything to this (or the stack).

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

**Answer:**

To make immutable version of pop() method,
We must construct a new object array that has all of the elements from the present object array, as well as remove and return an element from the new object array.
We won't change the old object array this way, and the pop() method will always return a new object array.

```
public Object[] pop (Object[] elements)
{
if (size == 0)
```

```
throw new IllegalStateException("Stack.pop");
Object[]N_O1= elements; // created a new object as N_O1
int n_s1 = size-1;  // new size as n_s1
N_O1[n_s1] = null;
return N_O1;   // return new object
}
```

The existing method will pop() and the we will return new elements.

**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.**

**Answer:**

Using list is simple because it is dynamic and does not require the implementation of equals() methods; instead, we can simply use. equals(). It comes in array

**4 Question 4**

**Consider the program below (y is the input).**

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

**Answer:**

Yes, it satisfies the given specification because firstly the pre condition is y>=2.

So, let's consider y is 2 and given that x=0.

So, it satisfies the while loop (0<2) and so the x becomes 3.

which satisfies the post condition which is x>=y example 0>=2.

So that it satisfies pre and post condition.

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

**Answer:**

The 3 loop invariants I have considered for the while loop over here is :

x>=1

y>=2

x<=y

TRUE, and it satisfies 3 loop invariants as post and pre 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. • 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.**

**Answer:**

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)

if x is greaterthan y it equals to true

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

x<=y & x>=y => x>=y

x=y => x=y (True)

here, we are taking x equal to y will be true

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

we get x<=y as true condition

b)WP(x:=0; {x<=y})=

By substituting x we get

y>=1=> 1<=y

we consider x equal to zero and satisfy the condition by substituting x we get y less then are equal to one

c)Compute the verification condition

vc1 (P => wp1(..)) P => WP(x := 0; while[x<=y] x<y do x+=2, {x>=y}) =

P => 1<=y =

y>=1 => 1 <= y ( True)

if it satisfies the vefication condition it is true.

**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.**

**Anwers:**

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

Consider y less then are equal to 1

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

(y>=1&x<y) =>y>=1 (False)

We Cannot simplify this further.

We will get false statement.

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

y>=1 & x>=y => x>=y (False)

here, we are getting false condition

(i)So the WP of the while loop is False, and therefore 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

we will get verification as false condition.

(iii)Analyzing the verification condition to determine whether the program is proved or not

Thus, using this loop invariant we cannot determine or prove the validity of Hoare Tripple

Here we are getting false in the three loop invariant

**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.**

**Answer:**

```
i:=1;

while(i<N)

{

i:=N;

}
```

Condition: i == N

Loop Invariants for the above program is:

Loop Inv: i<=N

Loop Inv: i>=2

Loop Inv: N>=2

Loop Inv: True

In the given i=1 where loop invariant it is i greater  then equal to 2 . where it satisfies it 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:=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

Where it holds i<=n is true condition

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

**Answer:**

Loop Invariant: A statement that should be true upon beginning a loop and at the end of each iteration.

Rep Invariant: capture rules that are critical in determining the correctness of a data structure.

Contract/specifications: They are the program's pre- and post-conditions, respectively. Pre-conditions are assumed to be true required conditions, while post-conditions are the program's effects.

**Example:**

```
public Class Chooser<T>{

private final List<T>choiceList;


public Chooser(Collection<T> choices){

choiceList = new ArrayList<>(choices);

}


public T choose(){

Random red = ThreadLocalRandom.current();

return choiceList.get(red.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 invariant :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

```
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

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

**Answer:**

Junit makes use of parameterized tests to support functionality.

The test data pieces in Junit are statically defined, and you, as a programmer, are responsible for determining what data is required for a specific set of tests.

Junit Theory applies a theory to a set of data inputs known as data points.

For Example:

```
@Theory
```

```
public void testCreateED1(String first, String second) throws Exception

{

String actual= ED1Utility.createED1(first,second);

assertThat(actual,is(allOf(containsString(first), containsString(second))));

}
```

The createED1() method takes two String parameters and returns an email ID in the format specified.

"Provided stringA and stringB passed to createED1() are non-null, it will return an email ID containing both stringA and string" -> "Provided stringA and stringB passed to createED1() are non-null, it will return an email ID containing both stringA and string" -> This is the test theory for it. testCreateEmailID() is a function that takes two String parameters and creates an email ID. TheTheories runner will call testCreateEmailID() at runtime, passing every possible combination of the type String data points we defined. For instance, (mary,mary), (mary,first), (mary,second), etc.

**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)?**

**Answer:**

Observable qualities are determined during testing. It is the most practical way for verifying a program for a single execution. Proofs, on the other hand, determine any program property. It verifies all of the executions of the programs. In 10-20 years, it may be feasible for small programs.

The formal reasoning concerning program correctness utilizing pre-conditions and post-conditions, according to Hoare Logic, is as follows:

Syntax: {i} v {a}
Where, i and a are predicates and v is a program.
If we start in a state where i is true and execute v, then v will terminate in state where a is true.

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

**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**

**Anwer:**

Liskov Substitution Principle:

It is an Object Oriented Programming Language idea that specifies that functions that use base class pointers or references must be able to use objects from derived classes without knowing it.

Example:

```
public class BD{
String title;
Integer userID;
}
public class OD extends BD
{
void getDeliveryLocations()
{
..
}
}
public class OD extends BD
{
void getSoftwareOptions()
{
..
}
}
```

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

**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?**

Team 2:

Saivaun kandagatla

Aastha Neupane

Anum Qureshi

Maurice Joy

**2. For each group member, rate their participation in the group on the following scale:**

**(a) Completely absent**

There no absents .

**(b) Occasionally attended, but didn't contribute reliably**

No, we did every week in group.

**(c) Regular participant; contributed reliably**

All, participated regularly and contributed eqully.

**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?**

Liskov Substituion principle was my favourite topic.

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

Group in class exercises

**3. What would you change for next time?**

I'd like the class to be IN-person