Final Exam

SWE-619

Question 1:

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. 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
        i. Partial Contract:
            1. Pre Condition/Requires: Element e is not null.
            2. Modifies: elements list size.
            3. Post Condition/effects: Add element to the queue
        ii. Total Contract:
            1. Modifies: elements list Size
            2. Post Condition/effects: Throws null pointer exception if element is null else adds element to the queue

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

For a total contract a method should not have any pre conditions, but the given method has a precondition where an element cannot be null as the method is not doing any validation for the object. To fix this I have added a null check for the element which will remove the pre condition.

    b. Write the rep invs for this class. Explain what they are.
        i. e!=null
        ii. 0<=size<=elements.size()
        iii. Elements from 0 to size doesnot contain any null values
    c. Write a reasonable toString() implementation. Explain what you did

```
if (Size=0) { return " "; }
String result = "";
for(int i=0;i<size;i++)
{
```

```
            result = result+elements.get(i);

              }
        return    result;

        }
```

d. 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.

```
public List<E> deQueueAll () {
    if (size == 0) throw new IllegalStateException("Queue.deQueue");
    List<E> result = new ArrayList<E>();
    while(size>0){
          result.add(deQueue())
    }
    return result;
  }
```

Here I am returning a list of elements which are present in the queue. I am reusing the dequeue method to remove the element from the queue and adding it to the list.

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

```
public Queue<E> deQueue () {
      if (size == 0) throw new IllegalStateException("Queue.deQueue");
      List<E> copy = new ArrayList<>();
     Copy.addAll(elements);
      Copy.remove(0);
      return copy;
   }
```

Here iam creating a new array list adding all the existing elements to the list then removing the first value to dequeue. Then returning the copy. This would maintain the immutability.

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

```
public List<E> clone(List<E> elements){

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

copy = (ArrayList) elements.clone();

return copy;

}
```

Here I created a new arraylist then cloned the existing list into the new one and returned the copy.

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.
    i. Ideal rep invariants would be (null != choices && choices.size > 0), choices.contains(null) != true (i.e collection doesnot contain any null element). Choices cannot be null and should not be empty and should not contain any null elements.
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.
    i. Constructor:
        1. Partial Contract:
            a. Precondition: choices != null and choices.size>0,
            b. Modifies: populates choices to the list
            c. PostCondition: creates a choice list
        2. Total Contract:
            a. Modifies: populates choices to the list
            b. PostCondition: if choice is null throws NPE else if choices.size==0 throws illegalArgumentException else if choices.contains(null) throws IllegalArgumentException else creates a choice list
        3. Code change for total contract: These changes are necessary as choices cannot be null or empty. This is because choice list will not be populated. Choice list cannot have null values as in the

choose method if the index of null element is picked then it returns null.

```
public GenericChooser (Collection<T> choices) {
  if(null == choices) {
        throw new NullPointerException("choices cannot be null");
  }
  if(choices.size() == 0) {
        throw new IllegalArgumentException("choices cannot be empty");
  }
  if(choices.contains(null)) {
        throw new IllegalArgumentException("choices cannot contain null elements");
  }
    choiceList = new ArrayList<>(choices);
}
```

ii. Choose():

1. Contract:
    a. PostCondition: Returns a random choice from the list.
    b. Explanation: Since the failure scenarios like empty list or null list or list containing in the null elements are handled at constructor itself choose() gets executed successfully without any issue.

b. 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.

i. Provided implementation of choose() is correct but prone to premature termination because of the constructor as the failure scenarios are not handled properly. Failure scenarios being possible null or empty collection as argument to constructor. This can be avoided by handling them(Updated code in the above answer). The second issue that is possible is what happens if the collection has a null value which might not lead to premature termination of the program but not an expected postcondition. This can be handled by checking possible null values in the list while populating itself, which is done in the above solution.

Question 3

1. What is wrong with toString()? Fix it.
    i. The problem arises with the pop method. When we do pop we are not removing the elements from the array instead just querying the element by index and returning the value. In toString since we are iterating the array by the size of array this leads to printing all the elements in the array even which were popped. The ideal way to do this is to iterate the array over the size object instead of size of array.
    ii. Code change:

```
@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().
    i. pushAll() depends on push method to insert the values in to the stack. Since both methods are public they can be overridden in case of inheritance which leads to modification in the behavior of the methods, which is a violation of Encapsulation principle. We need to either make them private or final to avoid overriding.
    ii. Contract:
        1. Partial Contract:
            a. PreCondition: collection != null and elements in collection are non null
            b. PostCondition: objects are populated into stack
        2. Total Contract
            a. PostCondition: if collection== null, object == null throw Null pointer exception else populate objects to stack
        3. Code change required for total contract:

```
public void pushAll (Object[] collection) {
```

```
        if(null == collection) {
            throw new NullPointerException("array cannot be null");
        }
        for (Object obj: collection) {
            if(null == obj) {
                throw new NullPointerException("object cannot be null");
            }
            push(obj);
        }
    }
```

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

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

Basically we can maintain the immutability by making a copy of the elements and returning the copied object by setting the index of popped element as null.

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.
    i. By using list we can use the inbuild equals method which will compare all then available elements in the list where as arrays just compare the size of the array which would not define the ideal behavior and we need to do the additional implementation. Using List would help avoid this.

Question 4

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

```
1 {y ≥ 1} // precondition
2
3 x := 0;
4 while(x < y)
5 x += 2;
6
7 {x ≥ y} // post condition
```

ii. The program should satisfy the given specifications as the while loop will ensure that the value of x will always be greater than y. As per precondition when y ≥ 1 will always lead to x ≥ y.

iii. Possible loop invariants for this specification are:

1. Firstly, the definition of loop invariant is, it is a property of a program loop that is true beforeand after each iteration
2. x<=y
3. x>=0
    a. x>0 can be considered as a loop invariant as its equal to 0 before execution and as y>=1 x will be greater than y.
4. y>=1
    a. y>=1 can be considered as loop invariant as it never changes throughout execution
5. TRUE
    a. True is the weakest invariant assumption for any loop

2. 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.

Was not able to identify an invariant which is strong enough to prove the hoare triples.
There are two more possible invariants which can come up in two differenct scenarios:
Scenario one:
If y is even the an invariant x<=y comes up. But this is not true as its not maintained for odd numbers.
Scenario two:
When y is odd x <y || x>y this cannot be considered as well as it does not maintain for even numbers.
With the available invariants I came up with doesn't prove the Hoare Triple.

3. 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.

ii. WP(while[y >= 1] x < y do x=x+2, {x >= y}) =

1. y >= 1

2. (y >=1 & x < y) => WP(x=x+2, y >= 1) =(y >=1 & x < y) => y >= 1 (True)We can't simplify this further.

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

We can rewrite this as:

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

After simplifying, we get:

x>= 1 => x >= y

False

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

= y>=1 & False & False

= False

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}

Compute the verification condition vc(P => wp(..))

P => False

y >= 1 => False

False

Analyze 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.

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.

Formal argument for a program correctness can be done through hoare triples by using pre and post conditions.

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

Depending the loop invariant chosen we can prove if a program/method is correct or wrong. By hoare triples if a program is proven false that doesn't mean program is incorrect. It means that the program is incorrect for that specific invariant where as it might be correct with other invariants.

We can take an example of below program from class excersice:

{N >= 0} // precondition

i := 0 ;while(i < N){i =i+1;}

{i == N} // post condition

Loop invariant:

Possible Loop Invariants:

i <= N

i>=0

N >= 0

TRUE

(i) an example showing a program (or method) is correct

Compute the weakest precondition wp of the program w rt the post conditioning Q

WP(i := 0; while[i<=N] i < N do i=i+1, {i == N}) =

WP(i := 0; WP(while[i<=N] i < N do i=i+1, {i == N}) =

// 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})

After simplifying, we get:

i < N => i+1 <= N

i < N => i < N

True

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

This can be written as:

i<=N && i>=N => i==N

After simplifying we get:

i == N => i == N

True

= i <= N && True && True

= i <=N

WP(i := 0; {i<=N}) =

By substituting i value we get:

0<=N => N>=0

c) Compute the verification condition vc (P => wp(..))

P => WP(i := 0; while[i<=N] i < N do i=i+1, {i == N}) =

P => 0<=N =

N>=0 => 0 <= N

(True)

d) Analyze the vc to determine whether the program is proved or not

Thus using this loop invariant we can prove the validity of the Hoare tripple.

ii) an example showing a program (or method) is incorrect.(consider loop invariant N>=0)

ompute the weakest precondition wp of the program wrt the post conditioning Q

WP(while[N >= 0] i < N do i=i+1, {i == N}) =

1. N >= 0

2. (N >=0 & i < N) => WP(i=i+1, N >= 0) =

(N >=0 & i < N) => N >= 0 (False)

We can't simplify this further.

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

We can rewrite this as:

(N >= 0 & i >= N) => i == N

After simplifying, we get:

i>= 0 => i == N

False

= N >=0 & (N >=0 & i < N) => i >= 0 & False (a & false & false=> false)

= N>=0 & False & False

= False

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

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

c) Compute the verification condition vc (P => wp(..))

P => False

N >= 0 => False

False

d) Analyze 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.

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

Loop Invariant: A statement should be true when entering the loop, maintaining the loop and termination of the loop

Rep invariant: It's the essential rules to capture the correctness of the datastructure or the program.

Contract/Specification : They are the pre and post conditions of the program where pre conditions are something we assume to be true or unhandled failure scenario and

postcondition is the effects or modifications that happen during the execution of the program which leads to expected result

Example: Below examples give a glimple of above concepts

```
//pre condition: x>0

//modifies: updates the value of x

//post condition : x=y

public int invariant(int x){
        Int y=0;
        while(y<x){
                x=x+1;
        }
        return x;
        }
```

Loop Invariants:

1. x>0
2. x<=y
3. y>=0

Rep invariant:

x>0

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

Theories runner allows to test a certain functionality against a subset of an infinite set of data points. Basically it captures some aspect of the intended behavior in possibly infinite numbers of scenarios which corresponds to the number of data points declared.

Example:

```
@DataPoints
  public static int[] positiveIntegers() {
```

```
    return new int[]{
                1, 10, 1234567};
}

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

This is simple test to check a + b = b + a. Since we have multiple theories here defining the data points will help us run both simultaneo usly.

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)?

Testing is done through Observable. Its to verify program for one execution. Where as proof is done on any program property. This property is verified for all executions. The proof of the program depends on the invariant chosen. If the program is wrong for one invariant then it doesn't mean program itself is wrong. There might be other strong invariants which might prove the program correct.

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.

Liskov States that 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. child class should be able to be used in the same manner as the parent class, according to relevant contracts.

Example:

```
class Dog {
  String getName() {
    return "Dog1"
  }
}
```

```
class SecondDog extends Dog {
  String getNoise() {
    return "Dog2!"
  }
}
```

SecondDog can stand in for any Dog in any context and behaves in the same manner, and it's therefore not a violation of the LSP.

Question 6:

1. Who are your group members?

    1. Jagadish Reddy Ramidi
    2. Suraj Varma
    3. Prabhath Surya
    4. Hussain Rohawala

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

All the team mates were available throughout the course and were very insightful and were available in timely manner to complete the assignments, coursework and study meets. Glad to had them as my team. (c) Regular participant; contributed reliably for all the team mates

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?

Favorite aspect would be professor explain the course work by including practical examples on the fly.

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

Professor giving insights on his Research work would be my favorite part of the course.

3. What would you change for next time?

Nothing much. Loved the course.