Final Exam

Tags

Question 1

Consider Queue.java.

```
/**
  * Generic Queue example
  * Mutable Version, without specifications
  * SWE 619
  * @author Paul Ammann
  */

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

```
/* partial contract
  precondition:
    - e cannot be null
  postcondition:
    - e is added to the end of the queue
    - the size of the queue is incremented
*/
public void enQueue (E e) {
  elements.add(e);
  size++;
}
```

- a partial contract has a precondition

```
/* total contract
  precondition:
  postcondition:
    - if e is null, throw a NPE
    - otherwise e is added to the end of the queue
      and the size of the queue is incremented
*/
public void enQueue (E e) {
  if (e == null){
    throw new NullPointerException("cannot enQueue null object");
  }
  elements.add(e);
  size++;
}
```

- a total contract is a contract with no preconditions, as such the code need to be changed so that no preconditions is required

- in order to remove the precondition that e must not be null, a postcondition that handles a null input is needed
- there are several ways to handle a null input, but a reasonable interpertation to make is that the programmer don't actually want to insert a null into the queue and that it was a programming error so an exception should be thrown to let them know what has occured

2. Write the rep invs for this class. Explain what they are.
   - this.size == this.elements.size() - the size of the queue should be equal to that of the number of elements the queue is currently holding. The size variable is redundent since list maintains its own size if there is a mismatch between the two then there is an error somewhere
   - this.size ≥ 0 - the size of the queue should always be greater than or equal to 0 (cannot have a negative sized queue)

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

```
public String toString() {
  return this.elements.toString();
}
```

   - general contract of the toString method only states to return a textual representation of the object, since the rep of the queue only really consist one significant field the elements list, a string of all the elements in the list should be a good enough representation of the queue object
   - although the list interface does not provide a toString method, the constructor initalizes the list to an arraylist that does provide a reliable implementation of the toString method that can be used to return a string representation of the queue
   - I thought about adding the size to the string to aid debugging but decided against it since java collections typically do not print it in their to string methods

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

```
/*
  preconditions:
  postconditions:
  - if dst is null, throw a NPE
  - otherwise remove everything from the queue and add it to the dst collection
*/
public void deQueueAll(Collection<? super T> dst){
  if (dst == null){
    throw new NullPointerException("deQueueAll: collection is null");
  }
  while(!this.isEmpty()){
    dst.add(this.deQueue());
  }
}
```

- this implementation checks that the destination collection is null, and then deQueues elements to the collection until the queue is empty
- follows Bloch's advice and uses wildcard types to allow the method to also export to any super type collections that can also hold objects of type E

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

```
/*
  precondition:
  postcondition:
    if size == 0, throw ISE
    otherwise return a new queue with the first element of the queue removed
*/
public Queue deQueue () {
  if (size == 0) throw new IllegalStateException("Queue.deQueue");
  Queue <E> q = new Queue <E>();
  for (int i=1; i<this.elements.size(); i++){
    q.enQueue(this.elements.get(i));
  }
  return q;
}
```

- since the queue is immutable, the deQueue method will need to return a new queue object
- this code will create a new queue, then loop through every element excluding the first element of the queue and add them all to the new queue, then return the new queue
- an illegal state exception is still thrown since a queue below the size of 0 violates the rep

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

```
public Queue Clone(){
  throw new CloneNotSupportedException("clone is not supported");
}
```

- according to both Bloch and the JavaDocs, if an object does not implement Cloneable then it should throw clone not supported
- since this implementation of queue does not implement Cloneable, the reasonable implementation of clone is to throw a CloneNotSupportedException

---

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.
    - choiceslist should not be null, since it will result in an exception when trying to choose
    - choiceslit should not be empty since the size of choice list is used to call nextInt() since that would result in an exception

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.

```
/*
  precondition:
    - choices is not null
    - choices is not empty
  postcondition:
    - add all choices to the choice list
*/
public GenericChooser (Collection<T> choices) {
  choiceList = new ArrayList<>(choices);
}
```

    - choices cannot be null since an exception will be thrown when trying to pass it to the new ArrayList
    - choices should not be empty to uphold the rep invariant and so that the choose method won't throw an exception when calling nextInt with an choicelist of size 0

```
/*
  precondition:
  postcondition:
    - return a random choice
*/
public T choose() {
  Random rnd = ThreadLocalRandom.current();
  return choiceList.get(rnd.nextInt(choiceList.size()));
}
```

- as long as the preconditions of the constructor is followed, the rep will hold and choose() will not fail

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.
    - the preconditions of the constructor method will maintain the rep invariant since it prevents choicelist from being null or empty
    - choice has no way of violating the rep invarariant as it does not remove the choice from the choicelist
    - choose will always be able to sucessfully call nextInt and get a choice from the choicelist and return a random choice from the choicelist
    - since choose had no preconditions and fullfills its postconditions, while maintaining the rep invariant the method is correct

---

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.
    - the for loop uses elements.length which returns the size of the entire array and not the size of the stack

```
 @Override public String toString() {
   String result = "size = " + size;
   result += "; elements = [";
   for (int i = 0; i < this.size; i++) {
     if (i < this.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().
    - the pushAll() method calls the push() method and this requires documentation

- this violates encapsulation because if a class were to extend the StackInClass and have an overridden pushAll() method that calls super.pushAll(), this may break the code
- the user would have no knowledge of how the pushAll() method is implemented and may make false assumptions
- according to Bloch inheritance violates encapsulationa and should only be used when the programmer controls both the subclass and superclass, otherwise composition should be favored over inheritance

```
/*
  precondition:
    - collection is not null
  postcondition:
    - if any item within the collection is null, throw a NPE
    - otherwise add all items in the collection to the stack
      and increment the size of the stack for each item added
*/
public void pushAll (Object[] collection) {
  for (Object obj: collection) { push(obj); }
}
```

- the precondition that the collection is not null ensures that no null reference is passed in causing an unexpected null pointer exception
- the postcondition documents the NPE that will be thrown by the push method if the collection contains a null and the fact that the size of the stack will be incremented after the call

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

```
public StackInClass pop () {
  if (size == 0) throw new IllegalStateException("Stack.pop");
  StackInClass s = new StackInClass();
  for (int i=0; i<this.size-1; i++){
    s.push(this.elements[i]);
  }
  return s;
}
```

- the pop method for an immutable version of the stack will return a new StackInClass object
- this implementation loops though all elements of the current stack excluding the last element and push it all onto the new stack
- an ISE is still thrown when trying to remove from a stack of size 0, since the rep would still be violated

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.
    - the rep of the stack is simply the elements of the object array
    - to implement equals, it simply needs to iterate through the arrays of both stack and check if they are equal
    - a list that comes with an implementation of the equals method will make this proccess a lot easier since the equals method of the stack could simply call the equals method of the list to check if two stacks are equal or not

---

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).

- the precondition y≥1 will be true since x = 0 and the while loop will only execute if x < y therefore y has to be greater than or equal to 1 to enter the loop

- x will be greater than or equal to y after the execution of the program since in order to exit the loop x must be greater than or equal to y, since 2 is added with each iteration x will be greater than y if one of the two is odd and equal to y if both are even

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 loop invariant holds when the loop is entered and is preserved after execution of the loop
- $x \geq 0$ - program adds to x and x set to 0 at the start
- $y \geq 1$ - the value of y does not change
- $y > 0$ - the value of y does not change and is preconditioned to be 1 or greater

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.

  WP(while[I] B do S, {Q}) =

  1. I and
  2. (I & b) $\Rightarrow$ WP(S, I) and
  3. (I & !b) $\Rightarrow$ Q

  1. $x \geq 0$
  2. $(x \geq 0 \ \& \ x < y) \Rightarrow WP(x = x + 2, x \geq 0)$

     S = x = x + 2

Q = x ≥ 0

E = x + 2

x + 2 ≥ 0

x ≥ -2 solved for WP(x = x + 2, x ≥ 0)

(x ≥ 0 & x < y) ⇒ x ≥ -2

!(x ≥ 0 & x < y) or x ≥ -2 DeMorgan's Law a ⇒ b == !a or b

!(x ≥ 0) or !(x < y) or x ≥ -2 applied negations over (x ≥ 0 & x < y)

x < 0 or x ≥ y or x ≥ -2 applied negations over x ≥ 0 and x < y

True one of them is true

3. (x ≥ 0 & !(x < y) ⇒ x ≥ y)

   x ≥ 0 & x ≥ y ⇒ x ≥ y applied negation to x<y

   !(x ≥ 0 & x ≥ y) or x ≥ y DeMorgan's Law a ⇒ b == !a or b

   !(x≥0) or !(x ≥ y) or x ≥ y applied negations over (x ≥ 0 & x ≥ y)

   x < 0 or x < y or x ≥ y applied negations over (x≥0) and (x ≥ y)

   True one of them is true

so WP(while [x ≥ 0] x < y do x += 2, {x ≥ y}) = x ≥ 0

now to find the weakes precondition of the program

WP(x=0, {x ≥ 0})

S = 0

Q = x ≥ 0

E = 0

0 ≥ 0

True

y ≥ 1 ⇒ True

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.

WP(while[I] B do S, {Q}) =

1. I and
2. (I & b) ⇒ WP(S, I) and
3. (I & !b) ⇒ Q

1. y ≥ 0
2. (y ≥ 0 & x < y) ⇒ WP(x = x + 2, y ≥ 0)

   S = x = x + 2

   Q = y ≥ 0

   x+2 ≥ 0

   x ≥ -2

   (y ≥ 0 & x < y) ⇒ x ≥ -2

   false if y = 0 x = -4
3. (y ≥ 0 & !(x < y)) ⇒ x ≥ y

WP(x=0, {false}) false

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.

    - a program is correct if it statisfy its contracts/specifications

```
/* total contract
  precondition:
  postcondition:
    - if e is null, throw a NPE
    - otherwise e is added to the end of the queue
      and the size of the queue is incremented
*/
public void enQueue (E e) {
  if (e == null){
    throw new NullPointerException("cannot enQueue null object");
  }
  elements.add(e);
  size++;
}
```

    - this method for the Queue in question 1 would be correct since it behaves as the contract specifies
    - when the method executes an element will be added to the queue and the size variable is increment, or in the event e is null an NPE is thrown

```
/* total contract
  precondition:
  postcondition:
    - if e is null, throw a NPE
    - otherwise e is added to the end of the queue
      and the size of the queue is incremented
*/
public void enQueue (E e) {
  if (e == null){
    throw new NullPointerException("cannot enQueue null object");
  }
  elements.add(e);
}
```

- removing the line incrementing the size variable, this method is now inccorect according to its contract
- after an item is added into the queue, the size variable is not incremented
- this breaks the rep of the entire queue since the other methods rely on the size variable and now it no longer matches the actual contents of the internal list

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 invariant is a statement of a property that an object must satisfiy to be considered a legal representation of that particular object
        - example: a rep invariant of a queue is that it should always have a size ≥ 0, if the size of a queue were to be negative it would no be considered a legal representation of a queue since logically a queue cannot hold a negative amount of items
    - a loop invariant is a statement that holds when a loop is entered and is preserved after the execution of the loop
        - example:

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

            - looking at the program from question 4, x ≥ 0 would be a loop invariant since it will hold when the loop is entered and is preserved after the execution
    - a contract is like a promise between the user and the developer of the program. The user promises to satisfiy the preconditions and the developer promises to deliver the postconditions
        - example:

```
// precondition: y >= 1
// postcondition: return x >= y
public int p(int y){
  int x = 0;
  while(x < y){
    x += 2;
  }
  return x;
}
```

- rewriting the program from question 4 into a method
- the user of the method must ensure that the input to the program is correct and satisfies the precondition
- the developer will ensure that the program is correct and will deliver and return a result that satisfy the postcondition as long as the preconditions are satisfied

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

- a JUnit Theory is a property based testing method that tests if a theory holds given some assumptions and a set of data points
- the benefts of using JUnit Theories is the ability to easily limit the values that are taken from the set of DataPoints and test that some property is satisfied (pick data that satisfied the precondition and check if post condition is statisfed)

```
public void symmetry Theory(Object a, Object b){
  assumeTrue(a!=null && b!=null);
  assertEquals(a.equals(b) == b.equals(a));
}
```

- this symmerty theory example here limits the input to non-null values from the data points (get data that meets precondition)
- then it checks that if two objects are symmetrical (checks if postcondition is met)

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 a program means to perform dynamic analysis on it and run it against some input and anaylze how it will behave agaisnt a set of inputs. Testing a program is fast but may miss unexpected inputs.
- Proving the program means to perform static analysis on it and analyze how it behaves on all posible input without running the program. Proving a program is difficult and not practical on large programs.
- If a program is cannot be proven, it does not mean it is wrong only that it could not be proven to be 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.
   - LSP states that if A is a subtype of B then an A is a B
   - as an example if there is a cat class that is a subtype of an animal class
   - an cat object is an acceptable object anywhere an animal object is demanded
     - such as assigning a cat object to an animal variable
     - passing a cat to a method that expects and animal etc.

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?

   Ariuka Munkhbat: C

   Jessie G: 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

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

    I liked that the professor typed out the notes instead of using slides, it made it easier to follow along

3. What would you change for next time?