Lucas Leitz
G01034246
SWE-619
Dec. 13, 2021

Final Exam

Question 1

Consider Queue.java, shown below.

```java
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;
    }
}
```

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.
    (i) From my understanding, the difference between the partial and total contract is that the partial is less safe than the total one. This is because the total one is specified for all legal inputs while the partial must include a requires clause. So, in the example of enQueue, a partial example contract could be as follows.

    // Requires: e to not be null
    // Effects: Adds element to the queue
    // Modifies: this

(ii) Now, if we want to make this a total contract it will need to be legal for all instances and therefore, we should remove the requires. To do this, we can simply add a line that checks if e is null and if so, throw an NPE. The new code would look like the image below. The new contract would be as follows.

// Effects: throw NPE if attempt to add a null element, otherwise add element to the queue

//Modifies: this

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

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

From class we know that a rep invariant is a statement of property that all legitimate objects satisfy, and all methods must preserve. With this in mind, we could attempt to write the rep invariants as follows.

// Rep-invariant1: size >= 0

// Rep-invariant2: If size > 0, then the elements in ArrayList elements from 0 to size should not be null.

These two rep invariants should hold, even in deQueue because deQueue will first check to see that size > 0 before decrementing.

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

My toString() is show in the image below. To begin, it starts by printing out the total size of the array. Then it initializes i to 0, and while i is less than size, it will print out each element in the ArrayList elements. It checks if i < size – 1 in order to add a comma after each element and make it look cleaner. It finally ends by returning the result and "]" to make a clean implementation of the total Queue object which will include all relevant information.

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

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.

    Below is an image of my representation for deQueueAll. Like deQueue, it first checks that size != 0, and then it starts a while loop with the condition size > 0. For each element in elements, it will first get element(0), add it to result, then remove it from elements and decrement size by 1. Finally, it returns result. The contract could be as follows.

    // Effects: If size == 0, throw ISE. Else, dequeue each element in elements and return result.

    // Modifies: this

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

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

    This was done as an in-class exercise before so I will be using my work from that. To make this class immutable, we would first need to make each of the variables elements and size final as well as make the class final. We could then add public

and private Queue constructors as follows: public Queue() and private Queue(List<E> newElements, int newSize). Now with all of this in mind we can begin writing immutable methods that will simply make copies of Queue so it is immutable. For deQueue we could write the following method.

```
public Queue<E> deQueue () {
    if(size == 0) throw new IllegalStateException("Queue.deQueue");
    final List<E> newElements = new ArrayList<E>(size - 1);
    newElements.addAll(elements);
    newElements.remove(0);
    return new Queue(new_elements, size - 1);
}
```

This will first check size > 0, then create a new final List<E> that has size – 1 of our current list of elements. It will then add all the elements we currently have to this new list and remove the head (dequeue). Finally, it will return a newly constructed Queue with our new variables. This method will make sure our implementation is immutable.

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

   First in order to be able to implement clone our class will need to add "implements Cloneable" at the top. Other than that, it is pretty straight forward, we override the method like in toString then return a newly created List<E> that has the same elements as our current List<E> elements. The code is shown below.

```
@Override
public List<E> clone () {
    return new List<E> ( new ArrayList<E>(elements));
}
```

Question 2

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

```
// List-based Chooser - typesafe
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()));
    }
}
```

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

This class is relatively small and only does a handful of things yet it will still require a few rep invariants. The first thing we might want to make sure of is that none of the elements in choices are null. We will also want to make sure that choices is not null and has size > 0 in order for choose to run properly. We need the size to be greater than 0 because when we run rnd.nextInt(choiceList.size())) if the size is 0 we will get an IllegalArgumentException

// Rep-invariant1: choices.size > 0

// Rep-invariant2: The elements in choices from 0 to size should not be null.

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.

As stated previously, we need to make sure that choiceList.size() is > 0 and we must make sure that the elements in choices are not null. Below is a modification to the code to ensure this stays true. The only code that needs to be modified is the Chooser constructor because we need to make sure any new chooser object has size > 0 and does not have any null elements.

```
public Chooser(Collection<T> choices) {
        if(choices.size() == 0) throw new IllegalStateException;
        for(int i = 0; i < choices.size(); i++){
                if (choices.get(i) == null) throw new NullPointerException
            }
        choiceList = new ArrayList<>(choices);
    }
```

The contract for Chooser would be as follows

// Effects: Throws an ISE if choices.size == 0. Throws an NPE if any of the elements in choices are null. Otherwise, creates a new Chooser object with choices.

// Modifies: this.

The contract for choose would be as follows. Because it only returns an element from choiceList it doesn't modify it in any way.

// Effects: Chooses and returns a random element from choiceList.

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.

   We ask the question here, how do we know if a method is *correct*. We know it is correct when it satisfies the given requirement/specification. In my previous answer, I said the contract for choose was that it chooses and returns a random element from choiceList. We know this is correct because when we create a new Chooser object, it satisfies the contract that the size > 0 and none of the elements are null. Because of this, choiceList.get(rnd.nextInt(choiceList.size())); will run properly because choiceList.size() will be a valid and legal int. We also know this will run properly because rnd.nextInt will then choose a random int from 0 to choiceList.size() and then we will return that element in that position in choiceList, and that will be a valid, legal element that is guaranteed to not be null. Therefore, the contract for choose is satisfied and 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.

   With the current implementation of toString in StackInClass, it will print out the entire length of array because of the line for(int i = 0; i < elements.length; i++). We don't actually want this because it could print out null elements, we instead only want to print out up to size. To fix this, we simply change that line and have i < size.

   for(int i = 0; i < size; i++){

2. As written, pushAll() requires documentation that violates encapsulation. Explain why and then write a contract for pushAll().

   If we look at Bloch Item 15, we see that it says instance fields of public classes should rarely be public and classes with public mutable fields are not generally

thread-safe. We want to make each class or member as inaccessible as possible, and a big issue here is that we have pushAll and push as both public and they can currently accept really any value to add. Because pushAll calls push we need to make push final or private because someone could override and make the rep invariant invalid. A new contract we could make for pushAll could be the following.

// Require: All elements are non-null

//Effects: If any element is null, then raise exception, otherwise, push everything to this

//Modifies: this.

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

   To make this immutable we would essentially need to create a new instance of StackInClass and copy everything from the current object over, except for the element we are popping out. I believe that it would look something like the code below.

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

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 an implementation in your answer, but if you find it helpful to do so, that's fine.

   In an array, we would have to compare the elements up to size. But if the elements are in an arraylist, we can simply reuse .equals(). When doing equals with arrays we would have to create a new hashcode such as one of Bloch's examples of using hash() * 32, but in an arrayList we can simply use hashCode() again.

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.

   We see that at first we have the precondition of y being greater than or equal to 1. Therefore, no input of y less than 1 will allow of code to run. Next we see on line 3 that we initialize x to 0. Because of this, the loop on line 4 will run. This is guaranteed to run every time because x will always be 0 in the first iteration and y will always be greater than or equal to 1, because of the precondition. For each loop on line 4, line 5 shows x being incremented by 2. As soon as x is no longer less than y, the loop terminates and we end up on line 7 with the post condition being x >= y. This post condition will always hold because the loop on line 4 cannot terminate until x is no longer less than y.

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

   With a loop invariant, we know that it is a condition that is true immediately before, during, and after each iteration of the loop. Therefore, the following three statements are loop invariants

   //LoopInvariant1: y > 0. We know this is true because y the precondition has y >= 1 and nothing modifies the value of y within the loop

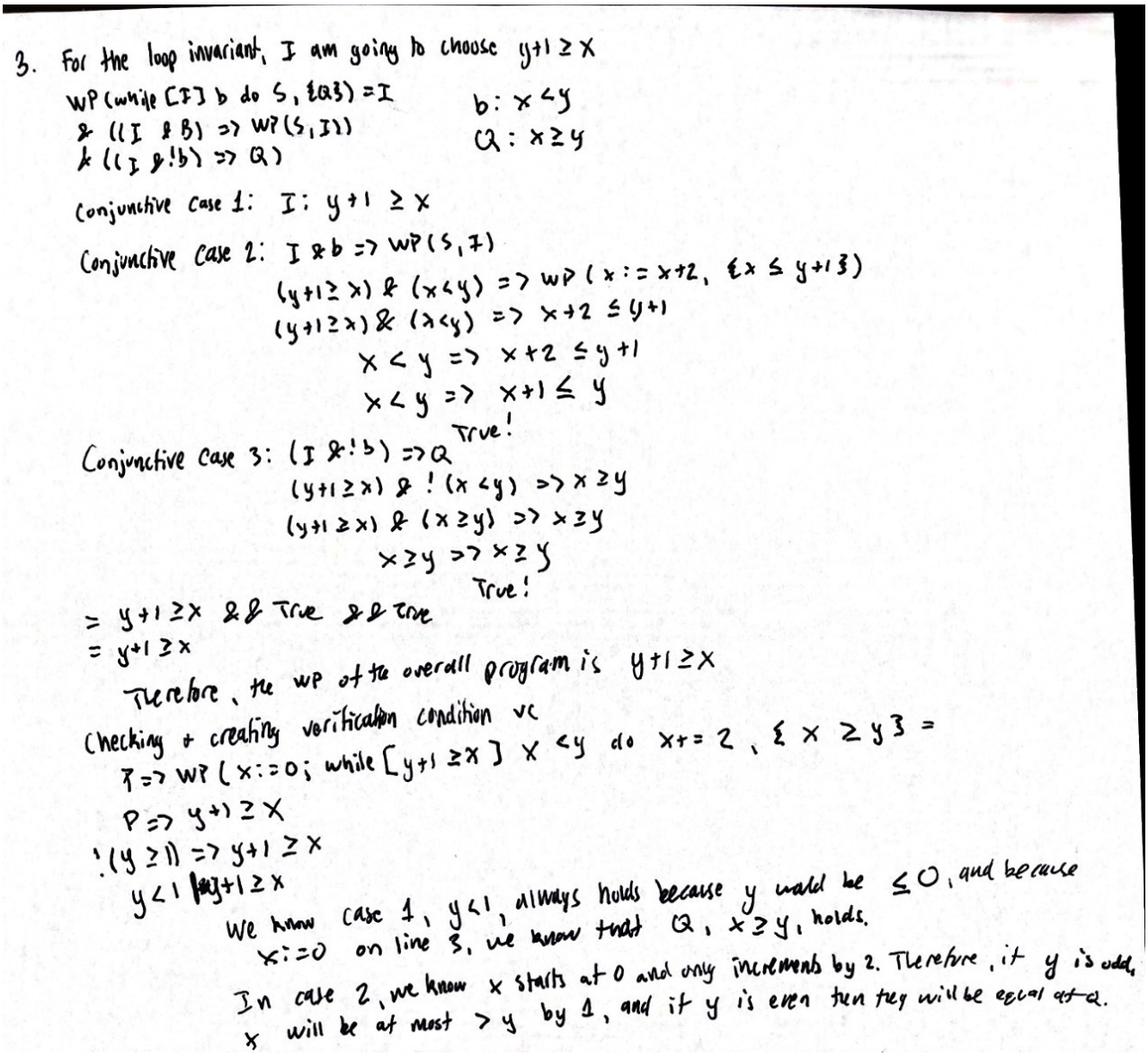
//LoopInvariant2: x >= 0. We know this is true because x is set to 0 before the loop and its value only increases
//LoopInvariant3: x % 2 = 0. This is saying that x is an even number, and we know this is true because only the value 2 is added to x each iteration, so the value of x is always evenly divisible by 2.

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.

   My solution is in the image below. I chose to use y + 1 >= x as my loop invariant.



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.

   My solution will again be in the image below. I chose the invariant to be y >= x.

4. For the loop invariant I am going to choose $y \geq x$

Case 1: I: $y \geq x$

Case 2: $(y \geq x) \,\&\, (x < y) \Rightarrow WP(x := x+2, \; y \geq x)$

$(y \geq x) \,\&\, (y > x) \Rightarrow y \geq x+2$

$y > x \Rightarrow y \geq x+2$

Case 3: $(y \geq x) \,\&\, !(y > x) \Rightarrow x \geq y$

$(y \geq x) \,\&\, (x \geq y) \Rightarrow x \geq y$

$y = x \Rightarrow x \geq y$

False!

Therefore, the WP of the program is False

Making the VC also false.

$P \Rightarrow$ False

$y \geq 1 \Rightarrow$ False

False

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, and (ii) an example showing a program (or method) is incorrect.

   For this problem we can look back at Question 2.3. As I stated, a program (or method) is correct when it satisfies the given requirement/specification. I previously stated that he contract for Chooser would be as follows

   // Effects: Throws an ISE if choices.size == 0. Throws an NPE if any of the elements in choices are null. Otherwise, creates a new Chooser object with choices.

   // Modifies: this.

We know that the implementation is correct because it does everything the contract requires and states. If we wanted to show an example that is incorrect, we could keep this contract the same but remove the line of code that throws an NPE if an element in choice is null. Therefore, we could have null elements and we would violate the contract.

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

   Contracts/specifications define what should be occurring at each method/invocation. The main point of a contract is to remove anything implementation related and make it as general as possible. Rep invariants are certain statements that must be held throughout the entire class, such as making sure no elements are null. Loop invariants are statements that hold before, during, and after a loop. If we look at our Queue class from question one each method has its own contract stating what it does and what it modifies, and then there are certain rep invariants that must hold for any method such as making sure size >= 0 and that elements does not contain any null elements. Good examples of loop invariants are seen in the previous problem question 4.

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

   With theories, unlike standard tests, we can create *general* statements that will work as theories that can allow data reuse across multiple tests whereas junit tests will really only test exactly what you code with a specifically provided data set.

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)?

   The difference between testing and proving is that testing will only use specific sets of data to verify code functions as it should while proving will generalize the code and assure that it will work for any valid input with the correct circumstances. Just because you cannot prove with Hoare logic does not mean a program is wrong, it still can 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.

   This basically states that a superclass can be replaceable with its subclasses without breaking the application. There were many examples from the semester, we could even just look at In class exercise 1 with User and SpecialUser. SpecialUser extends user and has the same methods and variables (along with new ones) but can still take the place of User and do anything it was intended to do because it has all the same methods and fields.

Question 6

1. Who are your group members?

   I was in group 6 and my group members were Luke Young, Mackenzie High, and Mozhgan Momtaz.

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

   I rate all members in my group as a (c) Regular participant; contributed reliably. We all worked together on Discord for each assignment and reading reflection with a set weekly meeting. Everyone was present for our meetings and contributed.

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?

Hoare Logic sucked!!! It was so hard!

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

I like that the professor gave us time during the class to work with partners and made most of our assignments and reading reflection group based. I worked really well with my team.

3. What would you change for next time?

I would possibly make the final a bit shorter! Thank you for a great semester professor!