SWE619–Fall'21: Final Exam

Your Name: Jessie Gonzalez

12/13/2021 Instructions

1. This is an open-book exam. This means that you can access course materials in the book/lecture notes/videos.
2. It is a violation of the honor code to communicate with any other person (except the instructor or TA) about this exam.
3. It is a violation of the honor code to discuss or share the contents of this exam in any way with any student who is currently registered for this course but who has not yet completed this exam.
4. You must type all solutions. You can use plain text format or markdown. If you use something else such as Word or LaTeX, you need to export to PDF and submit the PDF. Do Not submit any code (.java) file. if you need to change the code, put the modified code directly in your submission.
5. You need to submit on Blackboard by the deadline. If, for any reason, you have a problem submitting to BB, submit your final on Piazza in a private post. Your post should also explain your problem.

Section	Points	Score
Question 1	20	
Question 2	20	
Question 3	20	
Question 4	20	
Question 5	20	
Question 6	0	
Question 7	0	
Total	100	

1 Question 1

Consider Queue.java.

1. For enQueue, write

   (i) a partial contract and

   Partial Contract - The requires clause is needed if the procedure is partial.
   - Added a requires to make this a partial contract.

```
//Requires: e != null
//Effects: Add e to this
//Modifies: this
public void enQueue (E e) {
elements.add(e);
size++;
}
```

   (ii) a total contract. For each part, if you need to change the code for the contract, do so and explain what you did

   Total contract - contract that has no preconditions is a total contract

- Added logic in code to check for e being null and throwing NPE. Additionally, removed the requires clause to make this a total contract.

```
//Effects: if e == null throw NPE
//else add e to this
//Modifies: this
public void enQueue (E e) {
if(e == null){throw new NullPointerException();}
elements.add(e);
size++;
}
```

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

   elements != null, size != null, 0<=size

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

- I overrode toString method to display all elements or display empty queue. Looped through this to find all elements.

```
@Override
public String toString(){
String build = "";
if(this.isEmpty()){
build += "Empty queue";
return build;
}
for(E item: elements){
build += item.toString();
}
return build;
}
```

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

   - Created a contract that throws ISE if size is 0, else remove items until nothing is left. I also made deQueue and isEmpty final so it cannot be overwritten and break the rep invariant.

   Lastly, used wildcard type for parameter that serves as an E consumer. This will work for "collection of sometype of E", this address the issue of if the element type of the destination collection doesn't match implementation.

```
//Effects: if size == 0 throw ISE
// else remove all in this
final public void deQueueAll(Collection<? super  E> src){
if (size == 0) throw new IllegalStateException("Queue.deQueue");
while(!isEmpty()){
deQueue();
}
}
```

5. Rewrite the deQueue() method for an immutable version of this class. Explain what you did
   - I create a new queue instance then I looped through this instance and added all elements into the new queue instance. I then removed index 0 from the newly created instance and returning the new instance. Additionally, I made enQueue private so it cannot be overridden.

```
//Effects: if size is 0 throw ISE
//else return this with one element popped
final public Queue<E> deQueue () {
  if (size == 0) throw new IllegalStateException("Queue.deQueue");
  Queue<E> a = new Queue<E>();
  for(int i=0; i< size; i++){
    a.enQueue(elements.get(i));
  }
  E result = a.elements.get(0);
  a.elements.remove(0);
  a.size--;
```

```
    return a;
  }
```

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

   - I created a new instance of Queue and then copied all the values from this. This is also referred to a deep copy, no shared mutable state. Then I'm returning a new instance with all this elements.

```
  //Effects: if size == 0 throw ISE
  //else return a new instance of this will all elements of this
  @Override
  public Queue clone(){
    if (size == 0) throw new IllegalStateException("Queue.deQueue");
    Queue<E> q = new Queue<>();
    Iterator<E> itr = elements.iterator();
    while(itr.hasNext()) {
      q.enQueue(itr.next());
    }
    return q;
  }
```

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.

   choiceList != null // We need to initialize it or else we would get NPE

   choiceList < 1 // We need something greater or equal to one or else choose will throw IAE.

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.

   - Checking for choices and this to make sure they're not null if they are then throw NPE added code for this check. Additionally, added logic to make sure choiceList.size is greater than 0 if not throw IAE.

```
//Effects: if choices null throw NPE
//else create this with supplied parameters
//Modifies: this
public GenericChooser (Collection<T> choices) {
   if(choices == null){throw new NullPointerException();}
   choiceList = new ArrayList<>(choices);
}
//Effects: if this == null throw NPE
// if choices not greater than 1 then throw IAE
// else returns a random value from this
public T choose() {
   if(this == null){throw new NullPointerException();}
   if (choiceList.size() < 1) {throw new IllegalArgumentException();}
   Random rnd = ThreadLocalRandom.current();
   return choiceList.get(rnd.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 questions.

   - Choose is correct because it satisfies the contract and satisfies the rep invariant.

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.
   - It was using elements.length instead of size causing null/junk to be printed.

```
@Override public String toString() {
  String result = "size = " + size;
  result += "; elements = [";
  for (int i = 0; i < size; i++) {
    if (i < size)
      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().
   - It violates encapsulation since it's dependent on implementation details of push and ensureCapacity. If someone was to change these it may break pushAll.

   //Effects: Adds all items from collection parameter to this

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

- I created a new instance of StackInClass then I looped through the existing elements and added them to my new instance. After that I modified my new instance for removing the item, then returning my instance. This doesn't affect the rep invariant.

```
//Effects: removes item from this and returns this
public Object pop () {
   if (size == 0) throw new IllegalStateException("Stack.pop");
   StackInClass ab = new StackInClass();
   for(int i=0; i < this.size; i++){
      ab.push(elements[i]);
   }
   Object result = ab.elements[--ab.size];
   // elements[size] = null;
   return ab;
}
```

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.

   - Pop method wouldn't need to set elements[size] == null
   - We wouldn't need a separate variable size to keep track of size
   - EnsureCapacity method is no longer needed

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).

   If the program satisfies pre and post conditions, then y needs to be greater or equal to 1 then x will increment by two each time. Hence, x will be greater or equal to y after while loop executes. Using Hoare Tripple logic, if preconditions aren't meet then results are undefined.

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

   x≥0 //x gets created at zero and is only incremented

   y≥0 // since preconditions require y to be greater or equal to 1

   True //means you hit that location

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})
Using x≥0 as loop invariant to prove program
WP(while[I] B do S, {Q})

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

1. I
   x≥0
2. (I & b) ⇒ WP(S, I)
   (x≥0 & x < y) -> WP(x = x+2, x≥0)
   (x≥0 & x < y) ->WP(x+2≥0)
3. (I &!b) ⇒ Q
   (x≥0 & !(x < y)) -> x≥y //negation of x<y is x≥y
   (x≥0 & x≥y) -> x≥y
   True

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

WP(x:=0,{ x≥0})

= x≥0

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

P ⇒ WP(S, Q)

y≥1 -> WP(S,Q)

y≥1 -> x = x+2, { x≥0}

True

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

y≥1 -> x = x+2 >0 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})
Using y≥1 as loop invariant to prove program
WP(while[I] B do S, {Q})

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

1. I
   y≥1
2. (I & b) ⇒ WP(S, I)
   (y≥1 & x < y) -> WP(x = x+2, y≥1)
   (y≥1 & x < y) ->WP(x = x+2, y≥1)
3. (I &!b) ⇒ Q
   (y≥1 & !(x < y)) -> x≥y //negation of x<y is x≥y
   (y≥1 & x≥y) -> x≥y
   True

WP(while[y≥1] x < y do x = x + 2, {x≥y} = y ≥1

WP(x:=0,{ y ≥1})

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

P ⇒ WP(S, Q)

y≥1 -> WP(S,Q)

y≥1 -> x = x+2, {y≥1 }

True

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

y≥1 -> x = x+2 >{ y≥1 } True, although this isn't as strong as x≥0 since y isn't the value being incremented in the loop body.

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

- Choose is correct because it satisfies the contract and satisfies the rep invariant.

```
//Effects: if this == null throw NPE
// if choices not greater than 1 then throw IAE
// else returns a random value from this
public T choose() {
if(this == null){throw new NullPointerException();}
if (choiceList.size() < 1) {throw new IllegalArgumentException();}
Random rnd = ThreadLocalRandom.current();
return choiceList.get(rnd.nextInt(choiceList.size()));
}
```

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

This is incorrect since it doesn't say anything about throwing ISE and if this was an immutable instance then modifying this would break immutability.

```
//effects: remove item from this
  public Object pop () {
    if (size == 0) throw new IllegalStateException("Stack.pop");
    Object result = elements[--size];
    // elements[size] = null;
    return result;
  }
```

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

   Rep invariants – hold true throughout the implementation.

   Loop invariants – Must hold when entering loop and preserve when executing loop body.

   Pre/post conditions – are specific to a method or constructor.

   - If we take the previous example a possible rep invariant would be x!= null. This would apply to every method in this instance. Which would be checked by either an exception or a required clause in the methods.
   - Continuing with the example a possible loop invariant would be $x \geq 0$. This is preserved when entering the loop & preserved when executing loop body
   - The pre and post conditions in the program below are specific to this method. Another method may be acceptable to have y less than 1.

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

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

Junit theories allows to test a certain functionality against a subset of infinite set of data points. A parameterized test you would have to explicitly send each parameter to test against.

Example of Theories would be data points @DataPoints

```
public static String[] stuff = { "cat", "cat", "dog"};
```

Theories will test 2^3 just by the statement above.

Normal junit test you would need to explicitly pass each pair of parameters.

4. Explain the differences between proving and testing.

   Testing – Dynamic analysis – running the programing checking it with unit tests. With some finite number of inputs/tests.

   Proving – trying to reason with the code without running it. Checking the program over ALL possible inputs.

   In addition, if you *cannot* prove (e.g., using Hoare logic), then what does that mean about the program (e.g., is it wrong)?

   - Hoare logic is testing for all possible inputs.
   - It's not necessarily wrong it's just we cannot prove the validity of Hoare logic. It doesn't mean that you disproved it or show it's invalid.

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.

   Signature rule: a type for subtype should work for super type.

   Methods rule: Calls should work for supertype even though it goes to subtype.

   Properties rule: Subtype contains all properties in the super type.

   A basic example would

   Public class Bird{

   Public void fly(){}

   }

   Public class Duck extends Bird{}

   Public class Ostrich extends Bird{}

   In the example above both Duck and Ostrich can fly since they extend bird. We know Ostrich cannot fly so this is incorrect. Proving the signature rule since both duck and ostrich can extend. The calling of the fly() method demonstrates the signature rule. Additionally, both duck and ostrich can call the fly method proving the properties rule.

```
Public class Bird{}
Public class BirdFlys extends Bird{
Public void fly(){} }
public class Duck extends BirdFlys{}
public class Ostrich extends Bird{}
```

Now we have fixed the issue and ducks can fly and ostrich cannot.

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? Jacob Lin, Ariunsaikh Munkhbat
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) (c) Regular participant; contributed reliably

I would rate both C.

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? I think Junit was my least favorite since it's specific to Java.

   Favorite topics? I really enjoyed the generics piece of the course.

2. Favorite things the professor did or didn't do? Professor rocked he was super helpfully and responsive couldn't ask for anything more.

3. What would you change for next time? I wish we would have covered reflection since this came up in my job recently and maybe touched on functional programming with lambda expressions.