SWE619–Fall'21: Final Exam

Your Name: Qingyang Dai (G00983815)

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 (ii) a total contract. For each part, if you need to change the code for the contract, do so and explain what you did
2. Write the rep invs for this class. Explain what they are.
3. Write a reasonable `toString()` implementation. Explain what you did
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
5. Rewrite the `deQueue()` method for an immutable version of this class. Explain what you did
6. Write a reasonable implementation of `clone()`. Explain what you did.

1.

(i) a partial contract:

// Requires: this element cannot be null

// Effects: add the element to the Queue

// Modifies: this (or modifies the Queue representation)

(ii) a total contract:

// Requires: None

// Effects: if attempt to add a null element, throw NullPointerException,

// else add the element to the Queue

// Modifies: this (or modifies the Queue representation)

The code is as following:

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

2.

Rep-invariants:

(1) elements cannot be null: elements != null

(2) size cannot be negative: size>=0

(3) If size >0, then elements from 0 to size should not be null

3.

```
@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;

		if (!(obj instanceof Queue))
			return false;

		Queue s = (Queue) obj;
		return super.equals(obj) && elements.equals(s.elements);
	}
```

If this is the “Object obj” , so this means they are equal. And we just need to return true. Here we can use “instanceof” to check whether they are comparable. If there the input of equals() is not a type of Queue, this means these two objects is not comparable, so we need to return false. If there are comparable, we need to cast “obj” to Queue type and assign it to “Queue s”, and then check all the elements whether they are the same.

4.

Contract of deQueueAll():

// Requires: None

// Effects: if the Queue is empty, throw IllegalStateException,

// else delete all the elements

// Modifies: this (or modifies the Queue representation)

The code is as following:

```
	@SuppressWarnings("unchecked")
	public Queue<E> deQueueAll() {
		if (isEmpty()) {
			throw new IllegalStateException();
		}
		List<E> result = new ArrayList<>();
		int j = elements.size();
		for (int i = 0; i < j; i++) {
			result.add(elements.get(0));
			elements.remove(0);
		}

		return (Queue<E>) result;
	}
```

Explanation:

Firstly, we need to check whether the Queue is empty. If it is empty, we need to throw illegalStateException. Then, I set an empty list "result" to add the elements, and these elements are exactly what the Queue deletes. Finally, we can return this result to show which elements are deleted. And I also add a uncheck suppresswarnings of deQueueAll() method to improve code security.

5.

To make deQueue() method immutable, we can add "final" field in deQueue (). The code is as following:

```
	public final E deQueue() {
		if (size == 0)
			throw new IllegalStateException("Queue.deQueue");
		E result = elements.get(0);
		elements.remove(0);
		size--;
```

```
                return result;
        }
```

6.

The code is as following:

```
@SuppressWarnings("unchecked")
        @Override
        public Queue<E> clone() {
                Queue<E> result;
                try {
                        result = (Queue<E>) super.clone();
                        result.elements = new ArrayList<E>(elements); // deep
copy
                        return result;
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                        return null;
                }
        }
```

In order to avoid that the clone() method may has an issue of returning a new Queue type object which gets us into trouble as we have a super type and attempt to cast it down to a subtype. We need to deep copy the elements.

For this implementation, I created an Queue object called “result” outside of a try catch block. Next, called Queue super.clone() and casted that into an Queue. Next, I performed a deep copy of the elements into the Queue result object. Lastly, I return the result Queue object from the clone() method. If the exception is caught, then the exception information will be handed over to e, which will be processed by e. And I also add a uncheck suppresswarnings of clone() method to improve code security.

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.
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.
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.

1.

Rep-invariants:

(1) choiceList cannot be null: choiceList != null

Explanation: if choiceList is null, then we cannot create a chooser and cannot add choices to choiceList.

(2) the size of choices should be a positive integer: choices.size()>0

Explanation: if the size is zero or negative, we cannot call the choose() method to return a random choice. In the other word, if the size is zero, there is no choice on the chooser, so we cannot call the choose() method to return a random choice; if the size is negative, this is an illegal argument, and we need to throw IllegalArgumentException.

(3) Collection<T> choices cannot be null: choices != null

Explanation: if Collection<T> choices is null, then we cannot add this kind of choices to choiceList.

2.

The contract of choose():

// Requires: None

// Effects: returns a random choice in List<T> choiceList

According to the answers of previous questions, we need to obey these rep-invariants. So I recode "GenericChooser" as following:

```java
public GenericChooser (Collection<T> choices) {
        if (choices.size() > 0)
            throw new IllegalArgumentException();
        if (choices == null)
```

```
                throw new NullPointerException();

        choiceList = new ArrayList<>(choices);
    }
```

I add two if-expressions to check the null choices and non-positive size, which makes rep-invariants valid before call choose() method. And then we can invoke choose() following the contract.

3.

The choose() method is correct, because

(1) it satisfies the contract in my answers of the previous question. It doesn't have precondition, and it also satisfies the postcondition that it returns a random choice in List<T> choiceList;

(2) it preserves the rep-invariants in my answers of the first question. The constructor GenericChooser (Collection<T> choices) has already checked whether preserves the rep-invariants. So we don't need to worry about the rep-invariants when we invoke choose().

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.
2. As written, `pushAll()` requires documentation that violates encapsulation. Explain why and then write a contract for `pushAll()`.
3. Rewrite the `pop()` method for an immutable version of the `Stack` class. Keep the same instance variables. Rewrite what you did.
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.

1.

This toString() prints the entire contents of elements[], instead of just the elements of the stack, including to print junk data. We just want contents up to size and not elements.length, which outputs everything in the array.

Fix: Since size accounts for the # of elements in the stack and not the full length of elements[]. So we can replace elements.length with size. The code is as following:

```
@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.

pushAll() requires documentation that violates encapsulation, because this pushAll() didn't check for null element. If "Object[] collection" is null or contains null elements, the runtime error might happen.

The contract for pushAll():

// Requires: None

// Effects: if attempt to add a null element in collection, throw NullPointerException, else add all elements in collection to the stack

// Modifies: this (or modifies the stack representation)

The code is as following:

```
public void pushAll(Object[] collection) throws Exception {
            // check for null element
            for (Object obj : collection) {
                  if (obj == null)
                        throw new NullPointerException();
            }
            for (Object obj : collection) {
                  push(obj);
            }
      }
```

3.

To make pop() method immutable, we can add "final" field in pop(). The code is as following:

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

4.

Because lists allow duplicate elements and are ordered. The List interface equals() contract looks at the elements, the size of the List and the order of them to determine the equivalence of a List. Thus, if we use list, we don't need to override equals(), we can just use it.

If we use array, we need to override equals(), the code is as following:

```java
@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;

		if (!(obj instanceof Stack))
			return false;

		Stack s = (Stack) obj;
		return super.equals(obj) && elements.equals(s.elements);
	}
```

4 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).
2. Give 3 loop invariants for the while loop in this program. For each loop invariant, informally argue why it is a loop invariant.
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.
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.

1.

The precondition P is {y>=1}.

The postcondition Q is {x>=y}.

The initial condition is x:=0.

The while loop is when x is less than y, then executing x:=x+2. When x is equal to or larger than y, the loop will terminate.

The precondition P{y>=1} makes sure that y will be not less than x initially. Because x is 0 initially, if precondition makes y be less than the initial x, the while loop will never happen. Thus, the precondition makes sure that it will execute the while loop and satisfy the postcondition.

2.

I provide 3 loop invariants

$x \leq y$: If initially x is larger than y, the while loop will never happen. So this loop invariant make sure it can enter the loop.

$x \geq 0$: Because x is 0 initially, so x should be always larger than or equal to 0 when we do the while loop.

$y > 0$: If initially y is not larger than 0, the while loop will never happen. Because x is 0 initially, if y is smaller than x initially, the while loop will not happen. If we assume x and y should be an integer according to the requirement, we also can set this loop invariant as y>=1.

3.

A sufficiently strong loop invariant is $x \leq y$.

WP(x:=0; while[$x \leq y$] x<y do x:=x+2, {$x \geq y$}) =

WP(x:=0; WP(while[$x \leq y$] x<y do x:=x+2, {$x \geq y$}))

Here, WP(while[$x \leq y$] x<y do x:=x+2, {$x \geq y$})=

(1) $x \leq y$

(2) ($x \leq y$ && x<y) => WP(x:=x+2,{x<y})   // I && B => WP(S,I)

    x<y   =>x+2<y   // reduced I && B, WP substitution for assignment

        // and assume x and y should be an integer according to the requirement

⇒ True

(3) $x \leq y$ && !(x<y) => $x \geq y$

  $x \leq y$ && $x \geq y$ => $x \geq y$

  x =y   => $x \geq y$

⇒ True

Thus,

$x \leq y$ && True && True

⇒ True

WP (x := 0; {$x \leq y$}) =

$0 \leq y$

Creating and checking verification condition vc): P ⇒ WP (S, Q)

- P ⇒ WP (x := 0; while[$x \leq y$] x < y do x:= x+2, {$x \geq y$ })

- P ⇒ $0 \leq y$

- $y \geq 1$ ⇒ $0 \leq y$

⇒ True

- Thus, using this loop invariant we can prove the validity of the Hoare triple

4.

An insufficiently strong loop invariant is y > 0.

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

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

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

(1) y > 0

(2) (y > 0 && x<y) => WP(x:=x+2,{ y > 0 })   // I && B => WP(S,I)

y > 0 && x<y => x:=x+2 && y > 0

Cannot simplify further

(3) y > 0 && !(x<y) => x ≥ y

y > 0 && x ≥ y => x ≥ y

x > 0 => x ≥ y

False

Thus,

y > 0 && (y > 0 && x<y => x:=x+2 && y > 0) && False

⇒ False

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

WP (x := 0; {False}) ⇒ False.

The vc is then

- P ⇒ False
- y ≥ 1 ⇒ False
- ⇒ False
- Thus, using this loop invariant we cannot prove the validity of the Hoare triple.

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.
2. Explain the difference between rep invariants, loop invariants, and contract/specifications (i.e., pre/post conds). Use concrete examples to demonstrate the difference.
3. What are the benefits of using JUnit *Theories* comparing to standard JUnit tests. Use examples to demonstrate your understanding.
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)?
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.

1.

A method is correct means

(1) it should satisfy the contract of this method;

(2) it should preserve the rep-invariants of the class.

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

In Question 2, The choose() method is correct, because:

(1) choose() method satisfies the contract in my answers of the previous question. It doesn't have precondition, and it also satisfies the postcondition that it returns a random choice in List<T> choiceList;

(2) in Question 2, the modified code of GenericChooser (Collection<T> choices) checked whether preserves the rep-invariants. So we don't need to worry about the rep-invariants when we invoke choose().

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

In Question 2, the original code of GenericChooser (Collection<T> choices) doesn't checked whether preserves the rep-invariants. So when we call the choose() method, we cannot guarantee preserving the rep-invariants. So in this case, the choose() method is not correct.

2.

Rep-invariants:

Rep invariants can always be assumed to be true for a given object, and operations are required not to violate them. This means the given object and all the methods in class cannot violate them.

Take Question 2 as an example, the Rep-invariants are:

(1) choiceList cannot be null: choiceList != null

(2) the size of choices should be a positive integer: choices.size()>0

(3) Collection<T> choices cannot be null: choices != null

If the input violates one of these Rep-invariants, the methods choose() cannot be called correctly. And after every method being called, these Rep-invariants also cannot be violated.

Loop invariants:

Loop invariants capture the meaning of the loop. Their properties hold when the loop is entered and they are preserved after the loop body is executed. A loop invariant is a condition that is necessarily true before and after each iteration of a loop.

Take Question 4 as an example, a loop invariant is $x \le y$. It should hold when the loop is entered and they are preserved after the loop body is executed.

Contract/Specifications (i.e., pre/post-conds)

Contract is a kind of mechanism that requires a developer to comply with the specifications of an Application Programming Interface. Contract enables specifying conditions that must hold true when the flow of runtime execution reaches the contract. If a contract is not true, then the program is assumed to have entered an undefined state.

Take Question 4 as an example, a total contract for enQueue() method is:

// Requires: None

// Effects: if attempt to add a null element, throw NullPointerException,

// else add the element to the Queue

// Modifies: this (or modifies the Queue representation)

The enQueue() method doesn't have precondition, and it has postconditions. This means after call this method, the contract in the postcondition should be realized. If the caller attempts to add a null element, the compiler will throw NullPointerException, otherwise will add the element to the Queue.

3.

when you run JUnit Theories, it will enumerate all over all these values here and it will check each pair for us, one by one. It will tell us whether they all pass or they all fail and that allows us to check for mistakes.

I would like to take class MapPoly as an example, here is the Junit @Theory code:

```
@DataPoints
	public static Object[] test1 = { new MapPoly(2, 5), new
MapPoly(2, 2), new MapPoly(3, 5) };

	@Theory
	public void test1(MapPoly x, MapPoly y) {
		assumeTrue(x != null);
		assumeTrue(y != null);
		MapPoly z = x.mul(y);
		assertTrue(z.degree() == x.degree() + y.degree());
	}
```

There are 3 MapPoly instances, we want to check if we multiple two MapPoly instances, the degree of the result is equal to the sum of these two MapPoly instances' degrees. So here, using Junit @Theory will enumerate all over all these values and it will check each pair for us. And It will tell us whether they all pass or fail. This is very useful and convenient.

4.

Testing:

Testing is a practical method to try to find out how good or bad the programming is. And program testing are used to show the presence of bugs, but never to show their absence.

- Testing deal with the dynamic analysis by running the program;
- Testing will run the program on just some inputs;
- Testing is fast, and it does not need to analyze complex code;

- But testing could miss corner cases due to limited number of inputs.

Proving:
Proving is to prove that something is true or correct, you provide evidence showing that it is definitely true or correct.
- Proving deal with the static analysis by analyzing the source code;
- Proving will not run the program;
- Proving attempts to reason about the program on all possible inputs;
- But Proving is slow and infeasible to analyze some program source code;
- We can use Hoare logic to prove.

If you cannot prove (e.g., using Hoare logic), this doesn’t mean the program is wrong. Take the Question 4 as an example, if we cannot prove using some loop invariant, this means we cannot prove the validity of the Hoare logic using this loop invariant. It does not mean that you disprove it or show that the Hoare logic is invalid.

5.
The meaning of Liskov Substitution Principle (LSP) is:
If B is a subtype of A, B can always be substituted for A. In the other word, if B extends A (B is a subtype of A, A is a supertype of B), B should be more precise than A, strengthen properties of A.
(1) If A has some N methods, B will have those methods. B can have extra ones, and B overrides those N methods;
(2) If B overrides a method from A, an overriding method must have a stronger (or equal to) specification than the original method of A;
(3) B should have weaker precondition and stronger postcondition than A.
For example, there are 2 classes A and B as following:

```
class A:
    public void reduce (Reducer x)
        // Effects: if x is null throw NPE
        // else if x is not appropriate for this throw IAE
        // else reduce this by x
```

```
class B:
    public void reduce (Reducer x)
        // Requires: x is not null
        // Effects: if x is not appropriate for this throw IAE
        // else reduce this by x
```

Here B cannot extend A, because B has stronger precondition and weaker postcondition than A. But A can extend B, because A doesn't have precondition and B has precondition, so A has weaker precondition than B. And the postcondition of A contains "if x is null throw NPE" besides of the content of B's postcondition, so A has stronger postcondition than B. Thus, A can extend B, which follows Liskov Substitution Principle.

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?
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

1. Vu Doan, Qingyang Dai, Dana Jamous, Huan Le

2. (c), (c), (c), (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? Favorite topics?
2. Favorite things the professor did or didn't do?
3. What would you change for next time?

1. Favorite: Program Verification; Least: mutable and immutable.

2. Presenting many in-class exercises is really benefit for me to do practice.

3. Do more practice.