**Aastha Neupane**

**SWE 619**

**Final Exam**

**12/13/2021**

**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.**

**Ans:**

Partial contract:

Precondition: elements is not null

Modifies: this

Postcondition: adds element e to the Queue

public void enQueue (E e) {

elements.add(e);

size++;

}

\*\* ArrayList does not mention if it will allow null elements to be inserted to the list or not and this Queue class, thus it is not a precondition in this case as is.

Total contract:

//Precondition: none

//Modifies: this

//Postcondition: if elements is null throws NullPointerException, else if e is null, throws IllegalArgumentException otherwise adds element e to the Queue.

public void enQueue (E e) {

if(elements == null){ throw new NullPointerException(“Queue.enQueue”);}

if (e == null) { throw IllegalArgumentException (“Queue.enQueue”);}};

else{

elements.add(e);

size++;

}

}

Here to make it a total contract, I removed the precondition and instead handled elements being null by throwing NPE. Also added IAE when the element to be added is null- as we usually don’t want null elements in a Queue. I added these to the Effects/ Postcondition clause.

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

**Ans:**

The rep-invariants for the class are:

* Size >=0: the size of the Queue is 0 when the queue us first initialized and can be a positive integer as elements are added to it. Even deQueue will at most empty out the Queue and even in that case the size is 0. Thus, size >=0 is one of the rep-invariants of this class
* Elements is not null (elements !=null): In this class, elements can be empty, but it cannot be null- which is what we want since we don’t want any possible pointers to a null Queue

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

**Ans:**

public String toString() {

String result = "Queue: size = " + size + “; elements = “+elements;

}

The above will be enough as elements (or even elements.toString()) will print a nicely formatted Queue and this toString method is informative enough (with class name, size and all elements) that it becomes suitable in this case. Alternatively, we could use an iterator to iterate through a copy of the Queue and print elements individually. But I think in this case, the above toString method would suffice considering the implementations given.

1. **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**

**Ans:**

//Requires:

//Modifies: this

//Effects: if the list is null, throws NPE, if the list is empty, throws IllegalStateException otherwise removes all the elements from the queue and returns the removed queue elements.

public List<E> deQueueAll () {

if (elements==null) throw new NullPointerException("Queue.deQueueAll");

if (size == 0) throw new IllegalStateException("Queue.deQueueAll");

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

while(!isEmpty()){

newList.add(deQueue());

}

Return newList;

}

Here, aiming to have a contract with weakest possible precondition and strongest possible postcondition, I handled the conditions where the Queue is null and empty in this deQueueAll method. As the name suggests, this deQueueAll method removes all the elements from the queue and returns those elements (as a new queue/list) similar to what deQueue does. Here the code iterates through the given queue and using the deQueue method already present, dequeues each element and stores the returned (removed) element in the newList. The deQueue method takes care of removing each element and returning it. It also takes care of decrementing the size of the queue. At the end this deQueueAll method returns the newList.

I have followed all of Bloch’s advice while creating this deQueueAll method (don’t use raw types; favor generic types; favor list over array).

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

**Ans:**

public Queue<E> deQueue(){

if(isEmpty()){ throw new IllegalStateException(“Queue,deQueue”)};

List<E> newList = new ArrayList<>(elements);

newList.remove(0);

return new Queue<>(newList);

}

Here, I have modified the Queue to return a new Queue containing all elements of the original, except the first one (deQueue logic). In order to achieve this, I copied the contents of the original Queue into a temporary list and removed the first element from that list. Lastly I returned a new Queue with the contents of the newList. In order to make this deQueue method immutable, I also have to create a new constructor that is parameterized and a size() method as:

Private Queue (List<E> list){

this.elements – elements;

this.size = elements.size();

}

private int size(){

return this.size;

}

This immutable deQueue loses its ability to return the first element, thus if we want to get the top element, we will need to create and make use of a separate peek/top method to get the top element.

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

**Ans:**

If this class were immutable, we need not implement clone and just make use of Object’s clone method. However, since this is a mutable class, we need to implement it. In order to have clone() we need to implement Cloneable interface. Then implement clone as:

@Override public Queue<E> clone(){

try{

Queue<E> qu = (Queue<E>) super.clone();

qu.elements = (List<E>) elements.clone();

Return qu;

} catch (CloneNotSupportedException e) {

System.out.println(“Clone not supported”);

e.printStackTrace(e);

}

This clone() method copies the internals of the Queue; it does so by calling clone recursively on elements. This method returns a shallow copy of the Queue. Although, designing a clone method is very challenging, in this case, this way of calling clone() recursively works well.

**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.**

**Ans:**

* list != null: List cannot be null as it is always initialized through the constructor which creates a list containing elements of Collection choices. Choices may have no elements but the constructor ensures that the list is not null.
* size of the choiceList is greater than or equal to zero: the size of choiceList becomes 0 or greater as soon as its initialized.
* size of choiceList is equal to size of Collection choices: the size of list is equal to choices in this particular class, as it is initialized with the contents of choices and since there is no mutator method, it remains the same as choices throughout the class and its methods.

1. **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.**

**Ans:**

// Requires:

// Effects: throws NPE if collection choices is null, else throws IAE if choices contains null elements, else creates a new list containing all the elements of Collection choices

public GenericChooser (Collection<T> choices) {

if(choices ==null) throw new NullPointerException(“GenericChooser(choices)”);

for(T choices : cho){

if(cho ==null) throw new IllegalArgumentException(“GenericChooser(choices)”);

}

choiceList = new ArrayList<>(choices);

}

// Requires:

// Effects: throws IAE if choiceList is empty else returns a random element from the choiceList

public T choose() {

if(choiceList.size() == null) throws new IllegalArgumentException(“GenericChooser.choose”);

Random rnd = ThreadLocalRandom.current();

return choiceList.get(rnd.nextInt(choiceList.size()));

}

Above are the contracts for the constructor and choose methods (as-is in this GenericChooser

Class). Since, the constructor does not account for when choices is null or when choices contains null elements, in my above implementation (modified), I have included the cases in effects clause. The choose method as specified above (modified) checks if the list is empty or not. If it is then it throws IAE otherwise it returns a random element.

The contract specified above is consistent with the rep invariants I have mentioned above.

1. **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.**

**Ans:**

In order for any method (or program in high level) to be correct, it needs to satisfy its contract or specification. Thus, we need to make sure that the choose() method above adheres to its specification, holds the rep-invariant before and after its body and obey any related contract.

The chooser method above satisfies its postcondition that it returns a random element from the choiceList. This also does not break any contract of the ThreadLocalRandom or its current() method. This modified choose method also maintains/holds the rep invariant that I have mentioned above and even checks to make sure it is held by the method as is required by the contract. For chooser to be correct, it has to meet the contract/specification as well as the rep-invariants. This modified chooser does all of that. Thus, it’s implementation is correct.

**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.**

**Ans:**

There are a couple of things wrong with this toString() method. One of the things is that this toString method does not specify the class information i.e. it should include “StackInClass:….” in the result string.

Most importantly, this toString method is looping through the whole array/Stack length instead of the its size (they are different) and because of pop() method, if we iterate through the length of the array instead of size, we may get null elements if we have popped some things from the stack.

Thus, to fix this toString method, we need to iterate to size instead of elements.length and include the class name:

@Override public String toString() {

String result = "StackInClass: 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 + "]";

}

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

**Ans:**

This pushAll() implementation violates encapsulation because pushAll() method is implemented on top of push() method, which is called “self-use”. Self-use makes a class fragile. If this method were to be overridden by a subclass X, and then this overridden pushAll() were overridden by X’s subclass Y, then this StackInClass will not be aware of Y’s override and thus it is easy for Y to violate the rep-invariants/contracts of original superclass StackInClass.

This means that any subclass reimplementing this method might result in self-use, which is error-prone and reduces performance and possibly expose private fields. Thus, inheritance violates encapsulation.

Contract for pushAll() (StackInClass- as-is in the question)

//Requires: collection is not null, elements in collection are not null

//Modifies: this

//Effects: adds all the elements of the collection to the Stack

If modified to strengthen the contract, one possible implementation is given below (also an updated contract):

//Requires:

//Modifies: this

//Effects: throws IllegalArgumnetException if collection is null, else throws NullPointerException if any element in collection is null else adds all the elements of the collection to the Stack

public void pushAll (Object[] collection) {

if(collection==null) throw new IllegalArgumentException(“StackInClass.pushAll”);

for (Object obj: collection) {

if(obj==null) throw new NullPointerException (“StackInClass.pushAll”);

}

for (Object obj: collection) {

push(obj);

}

}

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

**Ans:**

public Stack pop () {

if (size == 0) throw new IllegalStateException("Stack.pop");

Object[] result = new Object[size-1]

For(int i=0; i<size-1; i++){

result[i]= elements[i]

}

return StackInClass(result, );

}

The above immutable version of pop() method required a new parameterized constructor that takes in an array of Objects, and creates a new Stack object. The pop() method returns a new Stack with the last element removed.

public StackInClass(elements) {

…//ommited

}

1. **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.**

**Ans:**

Implementing equals() with arrays is messy because overriding equals() with arrays is not very efficient- since arrays are primitive constructs in java. Arrays usually use Object’s/Array’s equals without overriding. In contrast, if we were to use List in StackInClass, implementing the equals method would become much easier. Its implementation is easier and done correctly, it will satisfy the contract of equals with more ease. If in this class the array were replaced by a list, we could override equals as:

@Override public Boolean equals(Object obj){

if(this==obj) return true;

if(obj==null) return false;

if(!(object instanceOf StackInClass)) return false;

StackInClass sic = (StackInClass) obj;

return elementsList.equals(sic.elementsList);

//elementsList replaced elements in this implementation

}

As we can see clearly that overriding equals with List is much easier that with array, which is not recommended.

**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).**

**Ans:**

As long as the precondition holds, the post condition of any correct program must be satisfied. In the code above, as long as y is greater than or equal to 1 the value of x at the end of the program body will be greater than or equal to y. To elaborate let’s walk through some scenarios with different values of y, while taking care to meet the precondition given;

* When y=1; x = 2 when it exits the loop and reached line 6/7. Since, 2>1, the postcondition x>= y holds
* When y=2; x = 2 when it exits the loop and reached line 6/7. Since, 2=2, the postcondition x>= y holds
* When y=3; x = 4 when it exits the loop and reached line 6/7. Since, 4>3, the postcondition x>= y holds

We can continue to test this out with other substitution s of y, and we see that as long as the precondition is met, the postcondition always holds. Thus, the given program satisfies the given specification of pre and postconditions.

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

**Ans:**

The three loop invariants for the while loop above are:

1. y>=1: this is a loop invariant because it holds before entering the loop (as it is the precondition here) as well as after completion of the loop body (the value of x changes not y).
2. x>=0: This is a loop invariant because it holds before entering the loop (as x is initialized to be 0 before entering the while loop) and it still holds when the loop body ends (x=2, x=4, …. which all still meet x>=0).
3. True: When we consider the precondition is met, True is the most trivial invariant as it is just a statement. We might not want to consider using true invariant to prove anything, but it still is a loop invariant.
4. **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.**

Ans:

We have the following formula for finding weakest precondition for a loop invariant. Using this WP, we will try to prove whether the program is correct with respect to the given specification or not.

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

Using **y>=1** as invariant:

For the whole program above:

WP (x=0, while [y>=1] x<y do (x=x+2),   {x>=y})

⇒ WP (x=0, WP( while [y>=1] x<y do (x=x+2),   {x>=y}))

Computing the WP of the while loop;

The algorithm for while loop WP is:

I && (I && B →WP(S,I)) && (I && !B → Q)

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

\*\*Here, I = (y>=1); B = (x<y); S = (x = x+2); Q = (x>=y), so

* y>=1 // I
* (y>=1 && x<y) → WP(x = x+2, y>=1) // (I && B) → WP(S,I)

(y>=1 && x<y) → y>=1

**True**  //For all values of y>=1 and x<y

* (y>=1 && !(x<y)) → x>=y // (i && !B) → Q

(y>=1 && x>=y) → x>=y

**True** // For all values tested of y>=1 and x>=y

WP of total program:

WP(x=0, y>=1)

=🡺 (y>=1)

Now verification condition:

P 🡺 (y>=1)

(y>=1) 🡺 (y>=1)

True

Since, we have proven the verification condition, using the WP (y>=1) we can now say that the program is valid.

**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.**

**Ans:**

Using **x>=0** as loop invariant:

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

For the whole program above:

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

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

Computing the WP of the while loop;

The algorithm for while loop WP is:

I && (I && B →WP(S,I)) && (I && !B → Q)

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

\*\*Here, I = (x>=0); B = (x<y); S = (x = x+2); Q = (x>=y), so

* x>=0 // I
* (x>=0 && x<y) → WP(x = x+2, x>=0) // (I && B) → WP(S,I)

(x>=0 && x<y) 🡪 (x+2>=0)

(x>=0 && x<y) 🡪 (x>= -2)

True

// Eg: if x=0, y=1, left hand side is true and right hand side is also (since 0>=-2 is true)

// all higher values of x will result true

* (y>=1 && !(x<y)) → x>=y // (i && !B) → Q

(y>=1 && x>=y) → x>=y

//True for all plausible values of x and y

Thus, WP is x>=0 & true & true

Now combining WP of while with the remain part:

WP(x :=0; x>=0) =🡺 x=0

Now verification condition:

P => x=0;

y >=1 => x=0

**False**  // since we cannot emply that y>=1 => x=0 since they are not comparable in this case.

Since, we cannot prove the verification condition, we cannot use the loop invariant (x>=0) to prove the program.

**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.**

**Ans:**

A correct program is the one that meets its specifications and satisfies some pre-set contract. For any program to be considered correct, it must satisfy the contract that was set before its implementation.

Example of a correct program:

//effects: prints all the elements in an array, prints empty array if it is empty

public void printArray(arr){

for (int i; i< arr.length; i++{

System.out.println(arr[i]);

}

}

This above method does exactly what its specified to do.

Example of an incorrect program:

//effects: prints all non-null elements

Public void printArray(arr){

for (int i; i< arr.length; i++{

System.out.println(arr[i]);

}

}

This same program becomes incorrect in this second case, because it does not meet its specification i.e. it does not handle condition for when arr[i] is null.

Thus, a program/ method is correct if it satisfies its contract otherwise it cannot be considered correct.

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

**Ans:**

Rep invariants are properties that are held true at a class/object level and all the methods, constructors, etc. of the class must hold this rep invariant true as well. The rep invariant must hold at the beginning of the program/class. It must hold before a method and must be held after the method execution.

On the other hand, loop invariants are more limited in scope than rep invariant. Loop invariants are properties that hold before entering a loop and also hold after the loop body executes.

Contract/ specifications are different than invariants. Contracts/specifications are descriptions of all technicalities, scenarios and conditions that a given program must meet or do in order to be accepted as being correct. It is the basis of any program as the program is structured and implemented with the specifications in mind in order to meet the contract.

One example of rep invariant can be a Set: ExampleSet class cannot contain duplicate values. This rep- invariant should hold before any Set methods and after that method finishes executing. Care is taken while implementing the methods of Set to preserve this rep-invariant because it is a property that must be held throughout the ExampleSet class.

A loop invariant can be a simple statement that must be true before the loop executes and after its body. Example;

i=0;

while (i <10){

i++}

Here one loop invariant could be i<10. It may or maynot be the strongest in this case but it still is a loop invariant as it is held before the start of the loop and after its body.

A contract/specification on the other hand is a clear descriptive specification of a procedure/method/class. For example:

Java doc gives contract for each of the java methods in its class/interface. One such method’s contract is given below. It is the contract for getClass() method of Object:

Returns the runtime class of this Object. The returned Class object is the object that is locked by static synchronized methods of the represented class.

**The actual result type is Class<? extends |X|> where |X| is the erasure of the static type of the expression on which getClass is called.** For example, no cast is required in this code fragment:

Number n = 0;  
Class<? extends Number> c = n.getClass();

In addition the specification /contract meeds to contain a precondition and postcondition as:

Precondition: none

Postcondition: returns the runtime class of this.

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

**Ans:**

Junit Theories provide a way to execute parameterized Junit tests. What this means is that we can define/declare a subset of possibly infinite set of data points (which are a set of data inputs) and run Junit test against all possible combination of those data points. In comaparision to standars Junit tests, Junit Theories covers multiple scenarios using Assume and Assert on possibly infinite number of inputs provided through data points. This makes our tests more flexible and vastly increases test coverage.

For example:

Consider the following standard Junit test:

@Test

Void test1(){

a = new ArrayList<>();

b = new ArrayList<>(a);

assertTrue(a.equals(b))

}

The above only executes the test once on the given two values a and b.

In contrast consider the following Junit Theories test:

…

@DataPoints

Public int testPoints = {1, 44, 50, -2}

@Theory

Public void testTheoryHere(int a, int b)(

assumeTrue(…);

AssertTrue(a.equals(b))

The above executes the test 16 times with various combination of the data points as parameters a and b.

Thus, Junit Theories provide more extensive testing than standard Junit tests.

1. **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)?**

**Ans:**

Proving means using some mathematical technique (for example Hoare logic which uses <, >, =, 🡺 etc. for proving) to prove that a program is valid and correct. For proving we usually make use of static analysis tools. Whereas testing is a process of running/executing a program to see if any errors/bugs occur. In contrast to proving/verification, testing is a dynamic process (program is actually run). Testing is done on a finite set of data/inputs whereas proving/testing usually involves checking the program over all possible inputs.

If we cannot prove a program using Hoare logic, that means that we simply cannot say anything about the validity of the program. We definitely cannot say that the program is invalid. This means that we cannot prove anything about the program.

1. **Explain the Liskov Substitution Principle (LSP). Use a concrete example to demonstrate LSP. Note: use a different example than the one given in Liskov.**

**Ans:**

Liskov Substitution Principle (LSP) states that all supertype’s behaviors must be supported by its subtype. A subtype object can be substitutes for supertype object without affecting the behavior of the program that uses it. The idea of LSP is similar to a subtype being a superset of its supertype, where the subtype has all the methods/behavior of its super type and it can have addition behavior as well. LSP requires that the specification of a subtype be either stronger than or equivalent to its supertype. In case of precondition and postcondition, LSP states that a subtype’s precondition must either be weaker than or equal to it’s supertype’s precondition. Conversely, a subtype’s post condition must either be stronger than or equal to its supertype’s post condition.

Let us consider a simple example of food and Gravy (pseudocode):

Class Food {

//Requires: object must not be null and must not be a beverage

//Effects: returns the consitency of the passed object

Public String consistency(Object item){

If(item.category != beverage){

String[] itsConsistency= {“fluid”, “semi-solid”, “solid”};

Switch item:

//Cases to handle different food types: omitted

return itsConsistency;

}

}

}

Class Gravy extends Food{

//Requires: none

//Effects: if item is null returns NPE, if item is a beverage returns IAE otherwise returns the consitency of the passed object

@Override

Public String consistency(Object item){

If(item ==null) {throw NullPointerException(“Food.consistency”);}

If(item.isBeverage) {throw IllegalArgumentException(“Food.consistency”);}

//reste same as suerclass

If(item.category != beverage){

String[] itsConsistency= {“fluid”, “semi-solid”, “solid”};

Switch item:

//Cases to handle different food types: omitted

return itsConsistency;

}

}

}

The above example follows LSP, as the precondition of subclass is weaker than its superclass’s and the postcondition is stronger than its superclass’s. Also, you can substitute the subtype object with the supertype object without affecting the behavior of the program.

**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?**

**Ans:**

Aastha Neupane, Anum Qureshi, Maurice Joy, Saivarun Kandagatla.

**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**

**Ans:**

Anum: Regular participant; contributed reliably

Maurice: Regular participant; contributed reliably

Saivarun: Regular participant; contributed reliably

**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?**

**Ans:**

The concept of pre and post-condition, mutability, the different takes on topics by Bloch and Liskov, LSP, and many more.

1. **Favorite things the professor did or didn’t do?**

**Ans:**

I liked how the professor went about explain things- sometimes from technical to abstract implementation and at times from abstract to technical.

I think it would have helped if you had shared some of your general notes on the topics we went through in the class.

1. **What would you change for next time?**

**Ans:**

Not much. But I think towards the beginning of the semester, having quiz at the end of the class distracted my focus from the class discussion. So maybe have quiz available before or after class.