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

Partial Contract

// Effects: add Object e to the end of the Queue

// Modifies: this

Total Contract

// Effects: If the Object is null, throw NPE

// add Object e to the end of the Queue

// Modifies: this

public void enQueue (E e) {

if (e == null) throw new NullPointerException("Queue.enQueue "); //added for total contract

elements.add(e);

size++;

}

In order to fulfill a total contract, enQueue needs to catch if Object e is null and throw a NPE if it is. The line “if (e == null) throw new NullPointerException("Queue.enQueue ");” is added to the function to fulfill the total contract.

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

elements != null: List<E> elements should not be null

0 <= size == elements.size(): int size and the size of List<E> elements should be the same, since both values represent how many elements are in the Queue, the size of the queue also cannot be less than 0/empty.

If size > 0, then elements from 0 to size should not be null: all of the elements in the Queue, represented by List<E> elements, are not null

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

@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 + "]";

}

A complete toString() method for Queue should have a human-readable representation of the size and contents of the Queue, represented by int size and List<E> elements respectively. The toString() written mimics the one done in In-Class 13, modified to fit the differing representation of Queue.java compared to Stack.java. The size of the Queue is printed, then each of the elements are printed in order of the Queue (the first element shown is the front of the queue).

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

// Effects: dequeue all of the elements in Queue

//store the dequeued elements

// Modifies: this

public void deQueueAll(Collection<? super E> dst){

while (!elements.isEmpty())

dst.add(deQueue());

}

The goal of deQueueAll() should be to empty the queue and, based on how deQueue() returns a given element, to have some sort of collection of the deQueued items ready after method execution. These parameters serve as the EFFECTS in the specification of the function.

Bloch’s popAll() method served as the model for this implementation. A while loop dequeues all of the elements in the queue and stores the elements in dst. Of note is his recommendation for generics regarding bounded wildcards. <? super E> is used because deQueueAll() is a consumer. using bounded wildcards allows for API flexibility and for deQueueAll() to handle possible supertypes of E.

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

An immutable Queue class would, instead of modifying “elements” to remove the item at the front of the line, generate a new queue with the element at the front of the line removed. All elements but the first in this.elements are passed into a temp Queue object. this.elements is overwritten with the resulting elements, and the size is passed in through result.elements.size(), which is representative of the new size of the Queue.

To retain the functionality of deQueuing an element but still returning it, a new method “front” would need to be created, since deQueue now returns a new Queue object instead of the element that is removed.

public Queue deQueue () {

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

//Immutable rewrite

Queue result = new Queue();

for(int i = 1 ; i < size ; i++){

result.elements.add(this.elements(i));

}

this.elements = result.elements;

this.size = result.elements.size();

return this;

}

public E top(){

return elements.get(0);

}

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

@Override public Queue clone(){

Queue result;

try {

result = (Queue) super.clone();  
 result.elements = new ArrayList(elements); // deep copy

result.size = this.size; // copy size of Queue

return result;

} catch (CloneNotSupportedException e){ // TODO Auto-generated catch block

e.printStackTrace(); return null;

}

}

The Queue class should implement Cloneable to indicate that a Queue can be cloned. Following the recipe for the clone method, I called super.clone(), did a deep copy of the elements in “this”, copied over the size of the Queue, and returned the resulting cloned Object. A catch block is added just in case to catch if cloning is not supported.

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: List<T> choiceList should never be null

If size > 0, then elements from 0 to choiceList.size should not be null: all of the elements instantiated in choiceList and represented in chooser should be valid objects and not 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.

// EFFECTS: if choices is null, throw NPE

// if choices contains any null values, throw NPE

// Initializes this to be a GenericChooser with list of elements as specified.

public GenericChooser (Collection<T> choices) {

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

for (T obj : choices) {

if (choices == null)

throw new NullPointerException(“GenericChooser.GenericChooser);

}

choiceList = new ArrayList<>(choices);

}

// EFFECTS: return a random element stored in GenericChooser

public T choose() {

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.

The choose() method, as documented and written, is correct. The stipulation that choose() is correct involves maintaining both rep invariants, and carrying out the behavior as specified in its contract. Since the rep invariants are caught in the constructor, and there are no other methods that modify choiceList, there is no need to handle catching rep invariants in choose().

The contract of choose() specifies that it returns a random element stored in GenericChooser. As written, choose() will successfully return a non-null element from GenericChooser. Having choose() be safe from null values results in a weak precondition and preferable contract.

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.

toString() uses elements.length instead of size when printing out the elements of StackInClass. This is a problem because size is the number of elements in the Stack and elements.length is the total size of the Stack. As written, the toString() method will print out all of the elements of Object[] elements rather than just the elements of the Stack. For any values between int size and elements.length, null or junk values will be displayed.

To fix this problem, rewrite the toString() method, replacing instances of elements.length with size.

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

Bloch's advice is that the class must document precisely the effects of overriding any method. In other words, the class must document itself-use of overridable methods. Since push (Object e) can be overridden, and pushAll() uses push (Object e), encapsulation is violated and Bloch would suggest that the class must document the effects of overriding push (Object e). A solution would be to make push (Object e) final so that it cannot be overridden or to make it private.

//REQUIRES: all elements in collection are not null

//EFFECTS: add everything to the the stack

//MODIFIES: this

I would prefer to rewrite pushAll() to throw an exception, but the question does not mention code rewriting.

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

public StackInClass pop () {

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

StackInClass result = new StackInClass();

int newSize = this.size - 1;

for(int i = 0; i >= newSize ; i++ ){

result.push(elements[i])

}

this.elements = result.elements;

this.size = result.size;

return this;

}

Public Object top(){

Object result = elements[size];

}

An immutable StackInClass class would, instead of modifying “elements” and size to remove the item at top of the stack, generate a new stack with the element at the top of the stack removed. To do this, a temp StackInClass is generated and the contents of the current stack copied over, sans the top of the stack which is represented as the last element in the array. The instance variables are copied over to the result and a new StackInClass is returned.

To make up for the loss in function of not having pop() return the top of the stack, a new function top() is created to replace that functionality.

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.

Since the array implementation of StackInClass has an elements.length >= the size of the stack, the empty values between this.size and this.elements.length are either junk or null values.

eg. [“dog”, null, null, null] has the same representation as [“dog”], but differs due to implementation

This means that an equals method written for the current array implementation of StackInClass must account for only the valid values of the stack. An implementation of equals() for the current StackInClass would compare this.size to another stack, and then loop through the array elements up to this.size to compare element by element.

This is avoided in a List implementation, since Lists dynamically scale in size based on how many elements it contains meaning there are only valid values and no null/junk values to account for in an equals() method. An implementation of equals() should be able to use elements.equals() to compare this.elements to another stack.

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).

y is given a value of 1 or greater. x always starts as 0. The program will increment x by 2 while x < y. This means that x will increment until x == y in the case that y is an even number, and x will increment until x >= y if y is an odd number. This satisfies the post-condition of {x ≥ y}.

Simplified, given P: y ≥ 1, and x := 0, the while loop will run until b: (x < y) and either x == y or x == y + 1

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

**y ≥ 1:** this is the precondition, and y is never modified by the code. Thus, it should hold that y ≥ 1 before and after each loop iteration

**0 <= x:** x is assigned 0 and is never decremented, thus holding that x should remain equal to or greater than 0 before and after each loop iteration

**x <= y + 1:**  since the loop terminates when !(x < y), and x is incremented by 2 each iteration, it should hold that x should be x <= y + 1 before and after each loop iteration

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.

I: x <= y + 1

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

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

1. x <= y + 1
2. (x <= y + 1 & x < y) => WP(x := x + 2, {x <= y + 1}) =

**x <= y + 1 & x < y => x + 2 <= y + 1**, x + 2 <= y + 1 simplifies to x + 1 <= y

**x <= y + 1 & x < y => x + 1 <= y**, x <= y + 1 can be represented x - 1 <= y

**x - 1 <= y & x < y => x + 1 <= y**, x - 1 <= y falls in the domain of x < y

**x < y => x + 1 <= y, x <= y - 1** can simplify to x < y for int values

**x < y => x < y**

**TRUE**

3. x <= y + 1 & !(x < y) => x ≥ y

**x <= y + 1 & x ≥ y => x ≥ y**, x ≥ y, x <= y + 1, x == y and x <= y + 1

**y <= x <= y + 1 => x ≥ y,** x can be == y or == y + 1

**TRUE**

Verification Condition:

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

P => x <= y + 1

**y ≥ 1 => x <= y + 1**

**y ≥ 1 => 0 <= y + 1**

**y ≥ 1 => -1 <= y**

**y ≥ 1 => y ≥ -1**

**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

I: x <= y

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

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

1. x <= y
2. (x <= y & x < y) => WP(x := x + 2, {x <= y}) =

**x <= y & x < y => x + 2 <= y**, x <= y & x < y simplifies to x < y

**x < y => x + 2 <= y**, x + 2 <= y rewritten as x <= y – 2

**x < y => x <= y – 2**, x <= y – 2 is the same as x < y – 1 for int values

**x < y => x < y – 1**

**FALSE**

1. x <= y & !(x < y) => x ≥ y

**x <= y & x ≥ y => x ≥ y**

**x = y => x ≥ y**

**FALSE**

WP({False}) = True

P => False

y ≥ 1 => False

False

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.

For a program or method to be correct, the software should satisfy some contracts or specifications. Strong and precise contracts are necessary to hone in the function of a method, class, or program.

An example of a correct method is from In-Class 1. The equals() contract is as follows.

* The function should be
  + It is reflexive
  + It is symmetric
  + It is transitive
  + It is consistent
  + For any non-null reference value x, x.equals(null) should return false.
* The return type should always be boolean

In In-Class 1, it is demonstrated that for

public class User {

private String name;

public User (String name) { this.name = name; }

@Override public boolean equals (Object obj) {

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

return ((User) obj).name.equals(this.name);

}

The overridden equals method fulfills the contract for equals; by testing the method, it is shown that the method is reflexive, symmetric, transitive, consistent, and for any non-null reference value x, x.equals(null) returns false. Therefore, User.equals is correct.

An example of an incorrect method is in the very same example, In-Class 1 and the SpecialUser.

public class SpecialUser extends User {

private int id;

public SpecialUser (String name, int id) { super(name); this.id = id; }

@Override public boolean equals (Object obj) {

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

return super.equals(obj) && ((SpecialUser) obj).id == this.id;

}

With SpecialUser added, the contract for equals is now violated. When given the User and SpecialUser  
 User u = new User("Vu")

User w = new SpecialUser("Vu");

…

u.equals(w); // returns true

w.equals(u); // returns false

This violates the contract by demonstrating the lack of symmetry for SpecialUser

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 are properties of a class that hold true both before and after **method execution whenever being used outside of its class**. A good example of rep invariants can be seen in Question 1.2, shown as the following.

1. elements != null
2. 0 <= size == elements.size()
3. If size > 0, then elements from 0 to size should not be null

Justifications for why these three properties are loop invariants can be found in question 1.2. These rep invariants hold before, during, and after public method execution ie. when a method is invoked outside of its class.

Loop invariants are **properties of a loop that will hold both before and after each iteration**. For example, the loop invariants from question 4 are preserved before and after each iteration of while(x < y) x += 2; runs.

1. y ≥ 1
2. 0 <= x
3. x <= y + 1.

Justifications for why these three properties are loop invariants can be found in question 4.2. All of these loop invariants hold before and after each iteration of the loop.

A contract/specification model is represented as

* P (Precondition) established by the client
* Q (Postcondition) expected results of passing P through S
* S (Program) implemented by the developer

These components result in the model

* P {S} Q, where P is modified by S and results in Q

A specification models a procedure via 3 clauses

* REQUIRES: This clause states any constraints on use
* MODIFIES: This clause identifies all modified inputs
* EFFECT: This clause defines the behavior and outputs

The Contract/Specification Model defines the contract and specification for which a class and its methods function. **A contract/specification informally or formally lays out the acceptable inputs/preconditions of a function, and for every expected input, the outputs/postconditions those inputs produce. Most importantly, it is independent of the implementation of a class and its methods.**

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

JUnit is an automated testing framework to generate tests for Java methods and programs. JUnit tests objects and their interactions with other objects and usually consists of statically defined values to test specific cases.

For example, in my group’s Assignment 1, we generated JUnit tests to verify that the Loan.months() function

@Test

void testNumberOfMonthsAreCorrect() {

assertEquals(166, Loan.months(100000, 0.08, 1000));

assertEquals(141, Loan.months(100000, 0.08, 1100));

assertEquals(35, Loan.months(100, 0.02, 3));

}

The shortcoming of this method is that it only tests 3 very specific sets of inputs. It is possible that there exists a principal, rate, and payment that would result in a failed assertion.

JUnit Theories are data-driven tests that work on assumptions rather than static values. Instead of writing multiple tests to handle the expected function of a single method, a theory is written that will accept all appropriate inputs to test against.

For example, our In-Class 11 had us write a theory that would test any equals function’s symmetry. The example is as follows.

@Theory

public void testEquals(Object a, Object b){

assumeTrue(a!=null && b!=null);

assertTrue(a.equals(b) == b.equals(a));

if(a.equals(b){assertTrue(b.equals(a));

if(b.equals(a){assertTrue(a.equals(b));

}

This theory will successfully take any combination of objects a and b and test to see if the equals method implemented for the two Objects are indeed symmetrical, meaning they return true or false regardless of the placement of a or b as the input or caller.

The benefits of JUnit Theories over standard JUnit tests are that

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)?

Proving a method or program involves logical reasoning that, for all preconditions, the postcondition will hold true. This is done through Hoare logic or a Hoare Tripple. Proving a method or program via Hoare Logic shows that a given function is partially or totally correct in its implementation.

Testing is a form of dynamic analysis and involves verifying specific conditions/inputs to show that a method or program demonstrates the expected behavior given a specific set of inputs. Testing cannot prove the total correctness of a program, since there are possible edge cases that are missed through testing.

Not being able to prove a program through Hoare logic or other means does not mean that the program is wrong, it just means that there is no proof that it is complete/correct. A program can fail to be proven and work entirely as intended from a user-perspective.

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.

Liskov Substitution Principle is a methodical way for deciding whether the subtype relation holds by examining the specifications of the subtype and the supertype. The subtype should satisfy the LSP such that a programmer or user could reason about the function of a subtype based solely on the specification of its supertype.

It postulates that, if B is a subtype of A (B extends A), B can always be substituted for A.

There are 3 rules related to the substitution principle

1. Signature Rule - a subtype must have all the methods of a supertype
2. Methods Rule - call to these subtype methods should behave like calls to its supertype
3. Properties Rule - the subtype must preserve all the properties that can be proven about a supertype object.

An example exploring the methods rule of LSP is Quiz 6, with the following code given as part of 3 different classes and analyzing each condition given. We were to analyze if, based on the compose method, if when one class extends another, the methods rule would be violated. The subtype should have no precondition or a weaker precondition than the superclass to prevent cases where an input that would be acceptable in the supertype is not allowed in the subtype. The subtype should have a stronger post-condition than the supertype to .

class A {

public Iterator compose (Iterator itr)

// Requires: itr is not null

// Modifies: itr

// Effects: if this is not appropriate for itr throw IAE

// else return generator of itr composed with this

class B {

public Iterator compose (Iterator itr)

// Modifies: itr

// Effects: if itr is null throw NPE

// else if this is not appropriate for itr throw IAE

// else return generator of itr composed with this

For example, when B extends A, the precondition of B is weaker than the precondition of A since A catches itr null, which is a stronger precondition than B’s no precondition. Additionally, B has a stronger post-condition than A, since it can throw an NPE, IAE, and correct generator while A will only either throw an IAE or return the correct generator.

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?

Vu Doan

Dai Qingyang

Dana Jamous

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

All of my groupmates receive a rating of (c) 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?

Favorite aspect of the class was the structure. Having lectures and in-class assignments paired with reading and homework assignments gave great coverage for learning and application both in and outside of class.

Class materials were sometimes convoluted; it was difficult to decipher at first the acceptable file submission types. Material was often written in an unclear way or with typos. It was sometimes hard to find files; It wasn’t clear on the final that the file names linked to code at first.

Favorite topics were Lambdas and Generics; very useful tools to streamline programming and taught well.

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

ThanhVu bringing up his own research and relating concepts from class to his work, particularly in regards to testing, was nice to see how content from class can be applied practically.

3. What would you change for next time?

We didn’t have a mid-semester recap; it would have been nice to refresh on the materials learned so far.