Steven Story

SWE 619 Final Exam

Question 1:

1.

I. Partial Contract:

REQUIRES: Element e cannot be null
EFFECTS: Add element e to the queue
MODIFIES: this
Since this is a partial contract, no changes need to be done on the code, since the requires constrains values that can be passed to enQueue() to be not null.

II. Total Contract:

EFFECTS: if element e is null, then throw IllegalArgumentException
else, add element e to the queue
MODIFIES: this

The code would be rewritten as follows:

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

A check is done for if parameter e is null. IllegalArgumentException is thrown if it is instead of NullPointerException because one would not occur if a null had been added and it is a more general exception for when the client that tells the client that the element they attempt to passed in not allowed to be added to the queue.

2. Rep invariants for the Queue class are:
   I. elements != null. The queue cannot be null. If the queue is null, this will lead to exception to occur when attempting to use or modify it.
   II. Elements cannot contain null. Elements within the queue should not be null because it is not an expected value for the user (elements in queue should be meaningful).
   III. size == elements.size() && size => 0. Both the size and elements.size() should be the same value since they both represent the queue's current length. Size also cannot be a value below 0, since 0 represents an empty queue.

3. The implementation of toString() is:

```
@Override public String toString(){
        String result = "The Queue size is: " + size; // provides info on queue size
        result += "; Current Queue: [";
        for(int i = 0; i<elements.size(); i++){ // provide info on current elements of queue
                if( i != elements.size() – 1 ){
                        result += elements.get(i) + ", ";
                }else {
                        result += elements.get(i);
                }
        }
        result += "]";
        return result;
}
```

Liskov states that toString() should be written to display meaningful information for users. The toString() above captures the current size of the queue and state of queue formatted to be readable to a user.

4. The contract and implementation of deQueueAll() is:

```
//Effects: If size is 0, then throw a IllegalStateException
        else, while size is not 0, dequeue all elements of queue and add the dequeued
elements into Collection col
public void deQueueAll(Collection<? Super E> col){
        if (size == 0) throw new IllegalStateException("Queue.deQueueAll");
        while(size != 0)
                Col.add(deQueue());
}
```

The Collection is included as parameter to take the elements dequeued off the queue by deQueue(). Collection<? super E> follows Bloch's advice for using wildcard types on input parameters that represent consumers (which deQueueAll() is). Collection<? super E> consumes the E instances from the Queue. A check on if Queue is empty is done, which throws an IllegalStateException is thrown since an empty queue cannot be dequeued. Else, dequeue() is called and removed elements are added to the collection col.

5. An immutable version of the deQueue() would be:

```
public Queue<E> deQueue(){

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

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

        copy.addAll(elements);

        dequeueCopy.remove(0);
```

```
        return new Queue<E>(dequeueCopy);
}
```

In the above, a new List<E> is created. This is a defensive copy that receives all the elements of this using copy.addAll(elements). An element is then removed from dequeueCopy at index 0, and a new Queue object is created and returned using a private Queue constructor that sets this.elements to the elements of the defensive copy and the size to the defensive copy's elements.size(). This is new Queue<E> is returned in deQueue instead of the element removed because the new Queue needs to be received by the user or they would not receive it. To retrieve the value held at the location that was removed (like the mutable version of deQueue), a new method would have to be made like this:

```
        public E getFirst(){
                if (size == 0 ) throw IllegalStateException();
                return elements.get(0);
        }
```

6. Implementation of clone creates a new Queue, which is then cloned using a casted clone.super(). The elements of the Queue are then set using new ArrayList<E>(elements), which rewrites the elements of the clone into a new ArrayList . This is because using just super.clone on the Queue will cause the resulting clone to be set to the same elements list as the original. If the state of the original Queue's elements is changed, the clone's will as well. The cloning is surrounded in a try-catch block to allow CloneNotSupportedException to be an unchecked exception. The implementation is as follows:

```
@Override
Public Queue<E> clone(){
        Queue<E> result = null;
        try{
                result = (Queue<E>) super.clone();
                result.elements = new ArrayList<E>(elements);
        }catch(CloneNotSupportedException e){
                throw new AssertionError();
        }
        return result;
}
```

Question 2:

1. Good rep invariants for this class would be:
    I. The choiceList cannot be null. If the choiceList was null, attempting to choose from it could lead to an error or exception.
    II. choiceList cannot contain null values. The user is not expecting a null value to be selected from the choices, but a meaningful choice/value.
    III. The size of choices in GenericChooser must be greater than 0. This is because the purpose of the method is create a GenericChooser with choices. Allowing an empty Collection<T> choices could lead to potential unexpected/unwanted behavior within the choose method. If anything is removed, it could also lead to an error.
2.
    I. GenericChooser Contract:
        i. **EFFECTS**: if choices is null, then throw IllegalArgumentException()
            if choices contains null, then throw IllegalArgumentException()
            if choices size is empty, then throw
            else, create a new GenericChooser with choices

        GenericChooser would need to be recoded with checks for if choices is empty or null and if it contains any null values. This would be:

```
public GenericChooser(Collection<E> choices){
        if (choices == null)
            throw new IllegalArgumentException();
        if(choices.contains(null))
            throw new IllegalArgumentException();
        if(choices.size() == 0 )
            throw IllegalStateException();
        choiceList = new ArrayList<>(choices);
}
```

    II. choose Contract:
        i. **EFFECTS**: returns a random choice from choiceList

        No changes would need to be done on the code, as the state of the choiceList isn't being changed (choice is a getter method), meaning that the rep invariant is preserved and contract is satisfied with no need for exception handling.

3. The choose() method is correct when it satisfies the rep invariant of the class and its contract. Since choose() does not modify the choiceList (it gets a random value from it instead), it does not cause choiceList or elements within choiceList to be null. The state of all rep invariants is held before, during, and after choose() is called. The contract states that choose() returns a

random choice from choiceList. The implementation of the choose() method, creating a Random object and return a value from choiceList that is selected from a random index from 0 to choiceList.size, satisfies the contract.

Question 3:

1. The toString is using elements.length() when looping to retrieve elements within the StackInClass. When elements popped when pop() is performed, it removes the StackInClass 's elements by setting it to null. toString() as it is written would return all contents of the elements list, including elements that are no longer within the StackInClass. The toString needs to show the current state of the StackInClass object instead, and can be fixed by using the size value instead of elements.length() in the loop that adds element's value into the result String.

```
@Override public String toString(){
        String result = "size = " + size;
        result += "; elements = [";
        for (int I = 0; i < size; i++){
                if( I < elements.length – 1 )
                        result = result + elements[i] + ", ";
                else
                        result = result + elements[i];
        }
        return result + "]";
}
```

2. One issue in pushAll() that would require it to have special documentation is that it calls the push() method, which is overridable. If push gets overridden by a subclass that inherits from Stack, then pushAll()'s behavior will be affected. This could be resolved by following Bloch's advice to "design for inheritance or forbid it" and making either push() or the class final or private. Another issue is that pushAll() method would require documentation is that it would need to check that every element in the object[] collection parameter is not null. If an element is null, this would lead to an NullPointerException to occur when attempting to push the element with the push() method. This could also mean that elements that were pushed from the collection into the stack before the null element have been added to stack, but remaining elements from the collection would not have been pushed. This can be prevented by having pushAll() check for null values before it attempts to push them into the StackInClass. The contract for this version of pushAll() would be:

        Effects: If any element in collection is null, then throw a NullPointerException
                Else, add all the elements to StackInClass
        Modifies: this

3. An immutable version of Pop would need to remove elements with a defensive copy of the Object[] elements and would return a new StackInClass object using a private constructor that create a Stack Object with the copy. Removing the elements can be performed by adding elements of 0 <= I <size-1 of the StackInClass to the copy. The last element, the top of the StackInClass object, would not be added, satisfying the contract of pop() to remove the top of the stack. This implementation would be this:

```
Public StackInClass pop(){
```

```
        if (size == 0) throw new IllegalStateException("Stack.pop");
        Object[] popCopy = new Object[size – 1];
        for (int i = 0; i < size -1; i++)
                copy[i] = elements[i];
        return new StackInClass(copy);
}
```

Another method would also need to be added to get the element from the top of the stack, since the immutable version of pop() requires returning a new StackInClass() in order to preserve immutability:

```
public Object getTop(){
        Object result = elements[size-1];
        return result;
}
```

4. It would be easier to implement equals if the class used a list instead of an array because it would be able to reuse List/ArrayList's equals() method in StackInClass's equals(), which is simpler and more efficient than implementing equals() for an array. List in Java overrides Object's equals method to check if two lists contain the same elements in the same order, which would mean they are equals. Arrays in Java do not have an overridden equals() method, meaning implementation of equals for StackInClass must be implemented by us manually. To implement equals() for an array, a loop would need to be used to compare each elements list of the StackInClass objects against each other.

Question 4:

1. The precondition states that y must be greater than or equal to 1. The value of x before the loop is always 0, meaning that the condition to enter the loop will be meant regardless of y's value. This is because the smallest value y can be is 1, so x<y will be 0 < 1, which is true. If y is 1, x+= 2; will result in 2 and the loop will end with x = 2. The postcondition holds, since 2 is greater than 1 meaning that x >= y is true. If y is an odd value, x will remain in the loop until it becomes a value greater than y. If y is even, x will remain in the loop until it becomes equal to y. If both of these cases the post condition of x >= y will hold.

2. Some loop invariants include for this program are:
   i. y >= 1. y's value never changes before, during, or after the loop, which means that it is preserved for all of it.
   ii. x > = 0. x's value before the loop is 0 and during the loop it will always be a positive number. After the loop is performed, x will remain a positive number, meaning that x>= 0 remains true for the entire loop.
   iii. x> -1. x's value is initially 0 before the loop is performed, so 0> -1 is true. When x is in the loop, x increments and can only be a positive number. After the loop is performed, x remains a positive number. Therefore, x > -1 will remain true.

3. P: y>=1, I: y >= 1, b: x <y, S: x += 2, Q: x>= y
   WP(x:= 0; WP(while[y>= 1] x < y do x+=2, {x >= y}) =

   i. y >= 1 // I (loop invariant)

   ii. y >= 1 && x < y => WP(x: x+2, y >= 1) // (I && b => WP(S, I))
   y >= 1 && x < y => y >= 1 //substituted WP's x:x+2 into loop invariant
   True. If y is at least 1 and x is less than x (x can be equal to 0), the equation is 1 >= 1 && 0 < 1 => 1>= 1, which is true.

   iii. (y >= 1 && !(x<y)) => x >= y // (I && !b) => Q
   (y >= 1 && x >= y) => x>= y // simplified !(x<y)
   True. if y is any value greater than or equal to 1 and x is greater than or equal to y, then x>=y will be true. An example is if y is 1, x will be 2 and the equation will be 1 >= 1 && 2>=1 => 2 >= 1, which is true.

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

   P => y>= 1

   y >= 1 => y>= 1.

   True, since they are equal. If left-hand side is true, right-hand side will always be true. Therefore, the program is correct.

4. P: y> 0 and x >= 0, I: y > 0 && x >= 0, b: x <y, S: x += 2, Q: x>= y

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

i. y> 0 && x >= 0 // I (loop invariant)

ii. (y> 0 && x >=0) && x < y => WP(x: x+2, (y > 0 && x> = 0)) // (I && b => WP(S, I))
y > 0 && y> 0 => x +2 >= 0 && y>= 0//simplified 0<= x <y and substituted x:x+2 into x >= 0
True. The lowest value y can be is 1, and x lowest value is 0. The resulting equation would be 1> 0 && 1>0 => 0+2 (2) >=2 && 1 > 0, which results in true.

iii. (y> 0&& x >= 0) && !(x<y) => x >= y
(y > 0 && x >= 0) && x>= y => x>= y //simplify !(x<y)
x > 0 && x >= 0 => x>= y // simplified 0 < y <= x
x> 0 => x>= y // simplified left hand side (x > 0 and x>= 0 == x>0)
False because on the left-hand side, x can be 2 (after one loop iteration, 0+2 = 2), but y on the right hand side can be 3. The result equation would be (2>0 => 2 >= 3), which is false.

WP(x:= 0; WP(while[x>= 0] x < y do x+=2, {x >= y}) = (y>= 1 && True && False) = False

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

P => False

y =>1 => False

Question 5:

1. A program is correct when it satisfies its rep invariants. A method is correct when it satisfies both the class's rep invariants and its contract. An example of proving correctness can be taken from using Queue's enQueue from Question 1.

   The rep invariants of the class are that elements cannot be null && elements cannot contain null && size >= 0.

   And the contract of this method is:
   EFFECTS: if element e is null, then throw IllegalArgumentException
   else, add element to the queue
   MODIFIES: this

   i. The method below is correct because it satisfies all the rep invariants and its contract above.

```
public void enQueue (E e){
        if (e == null)
                throw new IllegalArgumentException("Element is not valid
addition");
        elements.add(e);
        size++;
}
```

   enQueue() contains a check on e for null values, which prevents null elements from being added to elements. This prevents elements from containing null values and enqueue() never change elements to null, which satisfies rep invariant that elements cannot be null. Size is also only incremented, meaning the method doesn't violate the rep invariant of size >= 0.

   ii. The method below (original implementation of enQueue) is not correct because it violates both its contract and the rep invariants of the class.

```
public void enQueue (E e){
        elements.add(e);
        size++;
}
```

   Since this version of enQueue does not include a check for if e is null, it allows null values to be added into the Queue. This violates both the contract of the method that states the expected behavior is to throw an exception and the rep invariant that elements cannot contain null values. As such, it is incorrect.

2. Rep invariants are constant constraints for the class. All methods in the class must adhere to the rep invariants before, during, and after execution. Any method added to a program must also be implemented to adhere to the rep invariants. An example of rep invariants is the GenericChooser class with the following rep invariants:

   choiceList cannot not be null
   choiceList cannot contain null values
   choices > 0

   Implementation of methods cannot violate any of the rep invariant, or the program is considered incorrect.

   While rep invariants are invariants that are preserved for the whole class, loop invariants must only be preserved for the loops they represent. As such, a method could contain multiple loops being done, each with their own loop invariants. While just preserving a rep invariant is a part of a whole program being correct, loop invariants are used to help prove that a loop is correct using verification like Hoare Logic. Loop invariants must be upheld before the loop is entered, while the loop is being iterated through, and after exiting the loop. Each loop can also have more than one loop invariant, which are found by a user. For the loop below, there are several loop invariants. Examples are x>=0 or i>=0:

   ```
   {x >= 0}
   i = 0;
   while (i < x)
           i: i*2;
   {i >= x}
   ```

   A contract/specification is method specific. The contract gives basic details about how a method act. Contracts can include details on behavior that is meant to maintain the rep invariants. An example of this is GenericChooser's constructor contract:

   EFFECTS: if choices is null, then throw IllegalArgumentException()
   if choices contains null, then throw IllegalArgumentException()
   if choices size is empty, then throw
   else, create a new GenericChooser with choices

The contract gives the user information about the expected behavior of the constructor. Some of this behavior includes throwing exceptions for behavior that could violate rep invariants of the class. Not every method needs to be implemented to satisfy the contract of another method, just written to satisfy both their own contract.

3. A benefit of JUnit theories is that they allow for testing to be done on finite permutation of data provided by the user via DataPoints. This allows for more testing to be done dynamically by performing the Junit Theory against an array of values multiple times automatically, as opposed

to standard JUnit tests, which require the user to have the input of the value entered statically in the test.

An example of this benefit of JUnit Theories is:

```
@DataPoints
        public static int[] dataPoints(){
                return new int[] {
                        20, 80,  40, 25, 30
};
        }
@Theory public void Pythagorean(Integer a, Integer b, Integer c){
        Assume.assumeTrue(a>0 && b >0 && c>0);
        Double equat = (Double.valueOf(a*a + b*b));
        Double csquare = Double.valueOf(c*c);
assertEquals(equat, csquare);
}
```

The above theory would use the dataPoints() above it to test the theory. The test would be done using a permutation of the datapoint's int[], which would lead to 125 runs of the theory against the datapoints. In JUnit, do a similar test would be less efficient, only allowing one run of specific data provided in the test case:

```
@Test
void testPythag(){
        Integer a = 20;
        Integer b = 30;
        Integer c = 40;
        Double equat = (Double.valueOf(a*a + b*b));
        Double csquare = Double.valueOf(c*c);
        assertTrue(equat == csquare);
}
```

Another benefit related to JUnit Theories use of datapoints is that Theories allow us to generify the Theories implementation. The Theory can be written to take an Object as a parameter, which allows it to be reusable for various types of data. Since standard JUnit tests require the user to include the objects being tested in them, they are less reusable.

4. Proving is verifying that the program is correct for all possible values as opposed to testing, which verifies if the program is correct using a finite set of values. Proving uses reasoning and logic for determining if the program works for all the values that can possible be used with it. Testing uses values the user inputs to test the program against. If you cannot prove, then this does not mean that the program is wrong. An example of this is that if a loop invariant that is used result in not being able to prove the program is invalid using Hoare logic, but rather that

loop invariant that was used was insufficiently strong to prove the program. Another loop invariant could be used that proves the program the valid.

5. Liskov Substitution Principle is the principle that subtypes can be substituted with supertype objects without affecting code behavior because subtypes must behave like their supertypes. The principle is divided by three rules:

    a. The Signature Rule, which states that subtypes must have the methods of the supertype and their signatures must be compatible/the same as the supertype (while being allowed to throw less exceptions).
    b. The Methods Rule, which states that calls of the methods for they subtype should also be valid for the supertype. This means that the precondition of subtypes should be weaker the supertype and the postcondition should be stronger.
    c. The Properties Rule, which states that properties of the supertype must be preserved by the subtypes.

    An example of would be for a Person class:

```
class Person{
//rep invariant: hobby does not contain null

        private List<String> hobbies();
        String name;

        //Effects: Creates a new Person with a list of hobbies.
        public  Person(){
                this.hobbies = newArrayList<String>();
        }

//Effects: return person's name
 public String getName(){
        …
}

//Requires n is not null
//Effects: add a hobby of the person
//Modifies: this
public void addHobby(String n){
        hobby.add(n);
}
```

    And a subclass of Student that extends Person;

```
class Student extends Person{
        //rep invariant: hobby does not contain null
```

```
        private int id;
        //Effects: Creates a new Student with a list of hobbies.
        public Student(){
                super();
        }

        //Effects: return person's name
        public String getName(){
                …
        }

        //Effects:  if n is null, then throw a NullPointerException
                else, add a hobby for the person

        //Modifies: this

        public void addHobby(String n){
                if(n == null) throw new NullPointerException();
                hobby.add(n);
        }

        //Effects: gets a hobby of the student from index i
        public String getHobby(int i){
                return hobby.get(i);
        }
}
```

In the above, the signature rule applies, as the methods of Person are included in Student subclass and the signatures of the methods (signature types and parameters) are the same. The methods rule also applies, as the method of addHobby() in Student can be considered the same as the addHobby() method in Person. This is because Person's addHobby() requires n not be null, meaning the postcondition of Person's addHobby() never has to handle a null condition. The properties rule also applies because Student adheres to the same rep invariant as Person (that the hobby list cannot contain null) and contracts of the methods do violate any properties of Person. Student also includes the getHobby() method, which it did not inherit from Person. Added methods like it must also preserve the rep invariant and properties of the supertype. Since all three rules apply, this means that Student is a valid subtype of Person.

Question 6:

1. Eden Knudson, Gustavo Paz, Utkrist P. Thapa
2. Eden Knudson: c, Gustavo Paz: c, Utkrist P. Thapa: c

Question 7:

1. Least favorite aspect. Books not always best in explaining. Lectures go over book material, but do not explain better than book does. In class exercises do not fully prepare for final exam. Hoare logic was tacked on with only a few notes and 1 lecture about it, but appeared on the final exam with an example more difficult than both homework, quiz, and inclass exercise.
2. Professor recorded lectures, but did not cover every topic fully enough to complete understand.
3. More examples, solutions to quizzes given after submission. Having written solutions to homework. More examples.