SWE619–Fall'21: Final Exam
Vu Doan
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

(i) partial contract:

```
// Require:          e != null
// Effects:    add element to the Queue
// Modifies:  this (or modifies the queue representation)
public void enQueue (E e) {
   elements.add(e);
   size++;
}
```

(ii) total contract:

```
// Require:   None
// Effects:    If e is null, throw NPE else add element to the Queue
// Modifies: this (or modifies the queue representation)
public void enQueue (E e) {
      if (e == null) throw new NullPointerException("Queue.push"); // added code
   elements.add(e);
   size++;
}
```

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

a. elements != null // The List<E> elements can not be null.
b. 0 ≤ size ≤ elements.length // Size of the queue is contain inside the List length
c. if size > 0 then elements from 0 to size should not be null

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

```
@Override public String toString() {
   String result = "size = " + size;
   result += "; elements = [";
   for (int i = size - 1; i >= 0 ; i--) {
      if (i > 0)
         result = result + elements[i] + ", ";
      else
```

```
            result = result + elements[i];
        }
        return result + "]";
    }
```

We only want to print out the content of the queue which is up to size, and not elements.length which outputs everything in the array. Here, I traverse the queue backward and print out every element.

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

```
// Remove all items of a queue
// Requires:    None
// Effects:     The queue is empty
public void deQueueAll(Collection <? super E> dest) () {
        while(!isEmpty()) { dest.add(deQueue()); }
}
```

For maximum flexibility, use wildcard types on input parameters that represent producers or consumers. If an input parameter is both a producer and a consumer, then wildcard types will do no good: need an exact type match.

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

```
public final E immutableDeQueue () {
    if (size == 0) throw new IllegalStateException("Queue.deQueue");

        // make a copy of the queue and operate on this array
    List<E> copyElements = new ArrayList<E>[size];
    for(int i = 0; i < copyElements.length; i++) { copyElements[i] = this.elements[i]; }

    // operate on the new array and return the element
    E result = copyElements.get(0);
    copyElements.remove(0);
    copyElements.length--;
    return result;
}
```

SWE619–Fall'21: Final Exam
Vu Doan
12/13/2021

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

```
// Clone method for class with references to mutable state

@Override public E clone() {
   try {
      E result = (E) super.clone();
      result.elements = elements.clone();
      return result;

   } catch (CloneNotSupportedException e) {
         throw new AssertionError(); // public clone method should omit the throws
clause
   }

}
```

In effect, the clone method functions as a constructor; need to ensure that it does no harm to the original object and that it properly establishes invariants on the clone. In order for the clone method on Queue to work properly, it must copy the internals of the queue. The easiest way to do this is to call clone recursively on the elements array.

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.

Rep invariants: choiceList != null && size(choices) > 0

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 youare doing and why.

```
public class GenericChooser<T> {
private final List<T> choiceList;

        // Requires: None
        // Effects: if choices is null, throws IAE, else creates the choiceList
        //                  if choices is empty, throw exception
        //                  !choices.contains(null), throw exception
```

```
        //                          create a chooser with choices
        public GenericChooser (Collection<T> choices) {
                if (choice.size() == 0) throw IllegalException;                  // ADD
                if (choice == null) throw NPE;                                   // ADD
                if (!choices.contains(null)) throw IllegalException;    // ADD
                choiceList = new ArrayList<>(choices);
        }

        // Requires: None
        // Effects: returns random choice in List<T> choiceList
        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 is correct because it is satisfy both the rep invariants of the class and the contract.

Explaning:

The Effects of method choose() is returing random choice in List<T> choiceList which is the list of choices. Looking at the GenericChooser() constructor we see that the postconditions ensure the rep-invariants of the whole class. Hence the choose() 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.

The StackInClass.java implementation would print all the elements in the lenght of elements, it it may be more elements than the Stack has. elements.lenght might be different comparing to size. We would only want to print out the content of the Stack, so in the for loop we only want to have the upper limit as size.

FIX:

```
@Override public String toString() {
   String result = "size = " + size;
   result += "; elements = [";
   for (int i = size - 1; i >= 0 ; i--) {
      if (i > 0)
         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().

The current pushAll() violates encapsulation because someone would override push() and violate the rep-invvariants. We ned to re-design pushAll() to either for inheritance or forbid it.

FIX:

```
// Effects:     If any element is null, then raise exception
// Modifies: Add everything to this ( or the stack

public void pushAll (Object[] collection) {
                for (Object obj: collection) { if (obj == null) throw NPE; } // added
                for (Object obj: collection) { push(obj); }
}
```

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

```
public Object pop () {
    if (size == 0) throw new IllegalStateException("Stack.pop");

    // make a copy of the stack and operate on this array
    Object[] newStack = new Object[size];
    for(int i = 0; i < newStack.legnth; i++) { newStack[i] = this.elements[i]; }

    // operate on the new array and return the element
    Object result = newStack[--size];
    newStack[size] = null;
```

```
    return result;
}
```

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.

If elements is Array, loop upto size of the stack and check if each elements are equal, reason being re-use equals() for Array will check the Array length, we only care about size of the stack. It would be much easier if the array was replaced by a list because we could just re-use the equals() of ArrayList due to the fact the ArrayList would dynamically scale.

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).

Given input y = 1, line 3 set x to 0, and 0 is less that 1, x < y so the loop starts. line 5 added 2 to x and x now become 2. The post condition saying that x ≥ y and 2 ≥ 1. HOLD.

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

LOOP INVARIANTS:

a. TRUE
b. x ≥ 0
c. y ≤ 2

Given input y = 1 , line 3 set x to 0, and 0 is less that 1, x < y so the loop starts. line 5 added 2 to x and x now become 2. The post condition saying that x ≥ y and 2 ≥ 1. HOLD.

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.

- using x ≥ 0 as loop invariant to prove program
- WP(while[x ≥ 0] x := 0 x < y do x += 2, {x ≥ y} =
    1. x ≥ 0 is the weakest precondition
    2. x ≥ 0 & x < y -> WP(x += 2, x ≥ 0)
       x ≥ 0 -> x += 2
       2 ≥ N -> TRUE
       TRUE
    3. x ≥ 0 & !(x < y) -> WP(x += 2, x ≥ 0)
       x == 0 -> x += 2
       x ≥ 0 -> 2 >= 0
       TRUE

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

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.

- using y ≤ 2 as loop invariant to prove program
- WP(while[y ≤ 2] x := 0 x < y do x += 2, {x ≥ y} =
    1. y ≤ 2 is the weakest precondition
    2. y ≤ 2 & x < y -> WP(x += 2, y ≤ 2)
       y ≤ 2 -> x += 2
       y ≤ 2 -> x ≥ y
       TRUE
    3. y ≤ 2 & !(x < y) -> WP(x += 2, y ≤ 2)
       y == 0 -> x += 2
       y ≤ 2 -> x ≥ y
       TRUE

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

5. Question 5

Note: you can reuse your answers/examples in previous questions to help you answer the following questions.

SWE619–Fall'21: Final Exam
Vu Doan
12/13/2021

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.

A program (or a method) is correct when it is satisfy both the rep invariants of the class and the contract.

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

```
public class GenericChooser<T> {
private final List<T> choiceList;

Rep invariants: choiceList != null && size(choices) > 0

    // Requires: None
    // Effects: if choices is null, throws IAE, else creates the choiceList
    //                  if choices is empty, throw exception
    //                  !choices.contains(null), throw exception
    //                  create a chooser with choices
    public GenericChooser (Collection<T> choices) {
        if (choice.size() == 0) throw IllegalException; // ADD
        if (choice == null) throw NPE;                  // ADD
        if (!choices.contains(null)) throw IllegalException; // ADD
        choiceList = new ArrayList<>(choices);
    }

    // Requires: None
    // Effects: returns random choice in List<T> choiceList
    public T choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }

}
```

The Effects of method choose() is returing random choice in List<T> choiceList which is the list of choices. Looking at the GenericChooser() constructor we see that the postconditions ensure the rep-invariants of the whole class. Hence the choose() is correct.

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

```
public class GenericChooser<T> {
private final List<T> choiceList;

Rep invariants: choiceList != null && size(choices) > 0

public GenericChooser (Collection<T> choices) {
        choiceList = new ArrayList<>(choices);
}

// Requires: None
// Effects: returns random choice in List<T> choiceList
public T choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
}

}
```

The Effects of method choose() is returing random choice in List<T> choiceList which is the list of choices. Looking at the GenericChooser() constructor we see that if the postconditions did not have the check similar to what we have in the correct example, then one could pass null as a choice into the choiceList and this would violate one of the rep-invariant leading to an incorrect implementation.

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

Rep invariants are type checking guarantees that whenever a method or constructor is called, its object this belongs to its class.

Example:

This repOk() ensure the rep-variants for a Stack:

a. elements != null

b. 0 <= size <= elements.length

c. If size > 0 then elements from 0 to size should not be null

```
public boolean repOk() {
  if (elements == null) { return false; }
  if (size != elements.length) { return false; }
```

```
        for (int i = 0; i < size; i++) {
            if (elements[i] == null) {
                return false;
            }
        }
    return true;
    }
```

Loop invariant are the standard approach to verify programs with loops. The technique is practically successful for both specifying and verifying loops in automated tools. The corresponding proof obligations propagate invariants forwards over a single arbitrary iteration, and soundness is justified by the induction principle of the least fixpoint of the loop. In sum, loop Invariant is the property that holds when the loop entered and it is preserved after the loop body is executed (inductive loop invariant)

```
while(1) {
// loop invariant (equal to all the loop invariances above)
if (!(condition)) break;
//code
}
```

Example:

```
{N >= 0}
i := 0;

// LOOP INV: i <= N
// LOOP INV: i <= 0 # NOT LOOP INV
// LOOP INV: i >= 0
// LOOP INV: N >= 0
// LOOP INV: TRUE

while (i < N) i := N;

{i == N}
```

Contract/Specifications: are the set of all program modules that satisfy it. This definition captures the intuitive purpose of a specification—namely, to state what all legal implementations of an abstraction have in common. Such a specification tells users what they can rely on and tells implementors what they must provide. A specification defines a contract between users and implementors: implementors agree to provide an implementation that

satisfies the specification, and users agree to rely on not knowing which member of the specific and set is provided. Writing a specification sheds light on the abstraction being defined, encouraging prompt attention to inconsistencies, incompleteness, and ambiguities. It forces us to pay careful attention to the abstraction and its intended use. An abstraction is intangible; without a description, it has no meaning. The specification provides this description.

Example:

(i) partial contract:

```
    // Require:          e != null
    // Effects:     add element to the Queue
    // Modifies:    this (or modifies the queue representation)
public void enQueue (E e) {
   elements.add(e);
   size++;
}
```

(ii) total contract:

```
// Require:   None
// Effects:    If e is null, throw NPE else add element to the Queue
// Modifies: this (or modifies the queue representation)
public void enQueue (E e) {
   if (e == null) throw new NullPointerException("Queue.push"); // added
   elements.add(e);
   size++;
}
```

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

The Theories runner allows to test a certain functionality against a subset of an infinite set of data points. A Theory is a piece of functionality (a method) that is executed against several data inputs called data points. To make a test method a theory you mark it with @Theory. To create a data point you create a public field in your test class and mark it with @DataPoint. The Theories runner then executes your test method as many times as the number of data points declared, providing a different data point as the input argument on each invocation. A Theory differs from standard test method in that it captures some aspect of the intended behavior in possibly infinite numbers of scenarios which corresponds to the number of data points declared.Using assumptions and assertions properly together with covering multiple scenarios with different data points can make your tests more flexible and bring them closer to scientific theories (hence the name).

For example:

```
@RunWith(Theories.class)
public class UserTest {
    @DataPoint
    public static String GOOD_USERNAME = "optimus";
    @DataPoint
    public static String USERNAME_WITH_SLASH = "optimus/prime";

    @Theory
    public void filenameIncludesUsername(String username) {
       assumeThat(username, not(containsString("/")));
       assertThat(new User(username).configFileName(),
containsString(username));
    }
 }
```

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)?

- Testing
    - Dynamic Analysis: analyze the code by running the code (Testing).
    - Run the program on some inputs
    - Strength: Fast, does not need to analyze complex code
    - Weakness: could miss corner cases
    - Program testing are used to show the presence of bugs, but never to show their absence
- Proving
    - Static Analysis: analyze the code without running it (Abstract Syntax Tree (AST), Bytecode ...)
    - Do not run the program
    - Strength: attempt to reason about the program on ALL possible inputs
    - Weakness: slow, infeasible, analyze the program source code.

If you cannot prove (e.g., using Hoare logic), then it means that the chosen invariant can not prove the correctness of the program, does not meant the program is 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.

SWE619–Fall'21: Final Exam
Vu Doan
12/13/2021

Liskov Substitution Principle (LSP):
-- If B is a subtype of A, B can always be substituted for A

B extends A (B is a subtype of A, A is a subtype a B)

In mathematics, a Square is a Rectangle. Indeed, it is a specialization of a rectangle. The "is a" makes you want to model this with inheritance. However, if in code you made Square derive from Rectangle, then a Square should be usable anywhere you expect a Rectangle. This makes for some strange behavior.Imagine you had SetWidth and SetHeight methods on your Rectangle base class; this seems perfectly logical. However, if your Rectangle reference pointed to a Square, then SetWidth and SetHeight doesn't make sense because setting one would change the other to match it. In this case Square fails the Liskov Substitution Test with Rectangle and the abstraction of having Square inherit from Rectangle is a bad one.

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? Qingyang Dai, Dana Jamous, Huan Le.
2. For each group member, rate their participation in the group on the following scale:
   (c) Regular participant; contributed reliably: Qingyang Dai, Dana Jamous, Huan Le.

7. Question 7

There is no right or wrong answer for the below questions, but they can help me improve the class. I mightpresent your text verbatim (but anonymously) to next year's students when they are considering taking thecourse (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?
   I like the way the class is now.
2. Favorite things the professor did or didn't do?
   Good at explaning the materials.
3. What would you change for next time?
   I would not change anything.