SWE 419–Fall'24: Review

December 6, 2024

1 Question 1

Consider the following `Queue` class.

```
"""
Generic Queue example
Mutable Version, without specifications
"""
class Queue:
    def __init__(self):
        self.elements = []
        self.size = 0

    def enQueue(self, e):
        self.elements.append(e)
        self.size += 1

    def deQueue(self):
        if self.size == 0:
            raise IndexError("Queue.deQueue")
        result = self.elements.pop(0)
        self.size -= 1
        return result

    def getFirst(self):
        if self.size == 0:
            raise IndexError("Queue.getFirst")
        return self.elements[0]

    def isEmpty(self):
        return self.size == 0
```

1. For `enQueue`, write (i) a partial specification and (ii) a total specification. For each part, rewrite the code if necessary to match the specification.
2. Write the *rep invs* for this class. Explain what they are.
3. Write a reasonable `toString()` implementation. Explain what it is.
4. Consider a new method, `deQueueAll()`, which does exactly what the name suggests. Write a reasonable specification for this method and then implement it (pseudocode is fine). Explain what you did.
5. Rewrite the `deQueue()` method for an *immutable* version of this class. Explain what you did

2 Question 2

Use Liskov Principle of Substitution rules to determine whether the below `LowBidMarket` and `LowOfferMarket` classes are proper subtypes of `Market`. Specifically, for each method, list whether the precondition is weaker, the postcondition is stronger, and conclude whether LSP is satisfied. If it is not satisfied, explain why not.

Note that this is purely a "paper and pencil" exercise. No code is required.

```
class Market:
    def __init__(self):
        self.wanted = set()  # items for which prices are of interest
        self.offers = {}     # offers to sell items at specific prices

    def offer(self, item, price):
        """
        Requires: item is an element of wanted.
        Effects: Adds (item, price) to offers.
        """
        if item in self.wanted:
            if item not in self.offers:
                self.offers[item] = []
            self.offers[item].append(price)

    def buy(self, item):
        """
        Requires: item is an element of the domain of offers.
        Effects: Chooses and removes some (arbitrary) pair (item, price) from
                    offers and returns the chosen price.
        """
        if item in self.offers and self.offers[item]:
            return self.offers[item].pop(0)  # Removes and returns the first price
        return None

class LowBidMarket(Market):
    def offer(self, item, price):
        """
        Requires: item is an element of wanted.
        Effects: If (item, price) is not cheaper than any existing pair
                    (item, existing_price) in offers, do nothing.
                    Else add (item, price) to offers.
        """
        if item in self.wanted:
            if item not in self.offers:
                self.offers[item] = []
            # Only add if price is lower than existing prices
            if not self.offers[item] or price < min(self.offers[item]):
                self.offers[item].append(price)

class LowOfferMarket(Market):
    def buy(self, item):
        """
        Requires: item is an element of the domain of offers.
        Effects: Chooses and removes the pair (item, price) with the
                    lowest price from offers and returns the chosen price.
        """
        if item in self.offers and self.offers[item]:
            # Find and remove the lowest price from the list
            lowest_price = min(self.offers[item])
            self.offers[item].remove(lowest_price)
            return lowest_price
        return None
```

Table 1: Symbolic Execution Example

Loc (l)	**Path Condition** (PC)	**Program State** (PS)
$l0$	T	$\{???\}$
$l1$	$???$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$
$l2$	$???$	$\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$
$l3$	a	$\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$
$l3$	$\neg a$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$
$l4$	$a \wedge b < 5$	$???$
$l4$	$\neg a \wedge b < 5$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$
$l5$	$a \wedge b < 5 \wedge \neg a \wedge c$	$\{???, y \mapsto 1, z \mapsto 0\}$
$l5$	$\neg a \wedge b < 5 \wedge \neg a \wedge c$	$\{x \mapsto 0, y \mapsto 1, z \mapsto 0\}$
$l6$	$???$	$\{x \mapsto -2, y \mapsto 1, z \mapsto 2\}$
$l6$	$a \wedge b < 5 \wedge (a \vee \neg c)$	$\{x \mapsto -2, y \mapsto 0, z \mapsto 2\}$
$l6$	$???$	$\{x \mapsto 0, y \mapsto 1, z \mapsto 2\}$
$l6$	$\neg a \wedge b < 5 \wedge (a \vee \neg c)$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 2\}$
$l7$	$a \wedge b < 5 \wedge \neg a \wedge c$	$???$
$l7$	$a \wedge b < 5 \wedge (a \vee \neg c)$	$\{x \mapsto -2, y \mapsto 0, z \mapsto 2\}$
$l7$	$\neg a \wedge b < 5 \wedge \neg a \wedge c$	$\{x \mapsto 0, y \mapsto 1, z \mapsto 2\}$
$l7$	$???$	$\{x \mapsto 0, y \mapsto 0, z \mapsto 2\}$
$l7$	$a \wedge b \geq 5$	$\{x \mapsto -2, y \mapsto 0, z \mapsto 0\}$
$l7$	$\neg a \wedge b \geq 5$	$???$

3 Question 3

```
void foo(int a, int b, int c){
    // l0
    int x=0, y=0, z=0;
    // l1
    if(a) {
        x = -2;
        // l2
    }
    // l3
    if (b < 5) {
        // l4
        if (!a && c) {
            y = 1;
            // l5
        }
        z = 2;
        // l6
    }
    // l7
    assert(x + y + z != 3);
}
```

Symbolic Execution Assume that we will apply symbolic execution on this program using symbolic inputs a, b, c. At each location l, we keep track of two things: the path condition (PC) to reach l and the program state (PS), consisting values of variables at l. Table 1 shows several locations and their corresponding PC and PS. Fill in the missing values ($???$) in the table.

4 Question 4

Consider the program below (`y` is the input).

```
{y ≥ 1}  // precondition

x := 0;
while(x < y)
  x += 2;

{x ≥ y} // postcondition
```

1. Informally argue that this program satisfies the given specification (pre/post conditions).
2. Give *three* loop invariants for the `while` loop in this program. For each loop invariant, informally argue why it is a loop invariant. Do not give the trivial loop invariants `True` and equivalent ones (e.g., if you say $x = y$ is an invariant, then do not give $y = x$ as another one).
3. **Hoare logic** Use a loop invariant to formally prove that the program is correct with respect to given specification. This loop invariant should be one of the three you gave above. You must show all the steps of the proof, including the weakest precondition, forming the verification condition, and analyzing the verification condition. The following steps are a guide to help you:
   - Determine the P, Q, and S for the Hoare triple $\{P\}\ S\ \{Q\}$ of this program.
   - Compute the weakest precondition $\texttt{wp}(\texttt{S}, Q)$ using the loop invariant you chose. This includes simplification as necessarily.
   - Form the verification condition $\texttt{wp}(\texttt{S}, Q) \Rightarrow P$ and analyze it
   - Based on the analysis, determine if the program is correct with respect to the given specification.
4. Instead of **proving** the given specifications like above, now you just want to **test** the program.
   - Provide *two* testing techniques A and B we discussed in class that you could use to test this program. For each one, clearly explain how you would use it to test this program, what you are looking for (e.g., what kind of bugs, violation of postcondition), and gives examples of test inputs that the technique would suggest.
   - Gives the pros/cons of using (i) the Hoare logic proof style, (ii) testing technique A, and (iii) testing technique B.