QUESTION 1

Swing is the User Interface library for Java. In Swing you can set a certain Layout for a particular window, called a JFrame in Swing. Here you see some examples of different Layouts that are available in the Swing library:

The **FlowLayout** class puts components in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, the FlowLayout class uses multiple rows. If the container is wider than necessary for a row of components, the row is, by default, centered horizontally within the container.

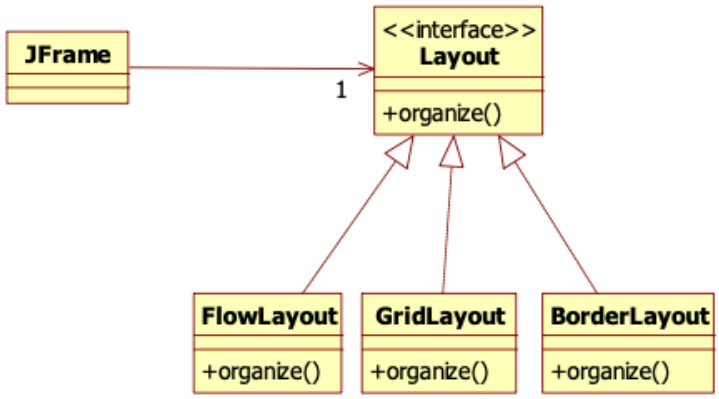
The **BorderLayout** object has five areas. These areas are specified by the BorderLayout constants:

- PAGE_START
- PAGE_END
- LINE_START
- LINE_END
- CENTER

If the window is enlarged, the center area gets as much of the available space as possible. The other areas expand only as much as necessary to fill all available space.

In Swing it is also possible to write your own Layout and use it for your UI windows.

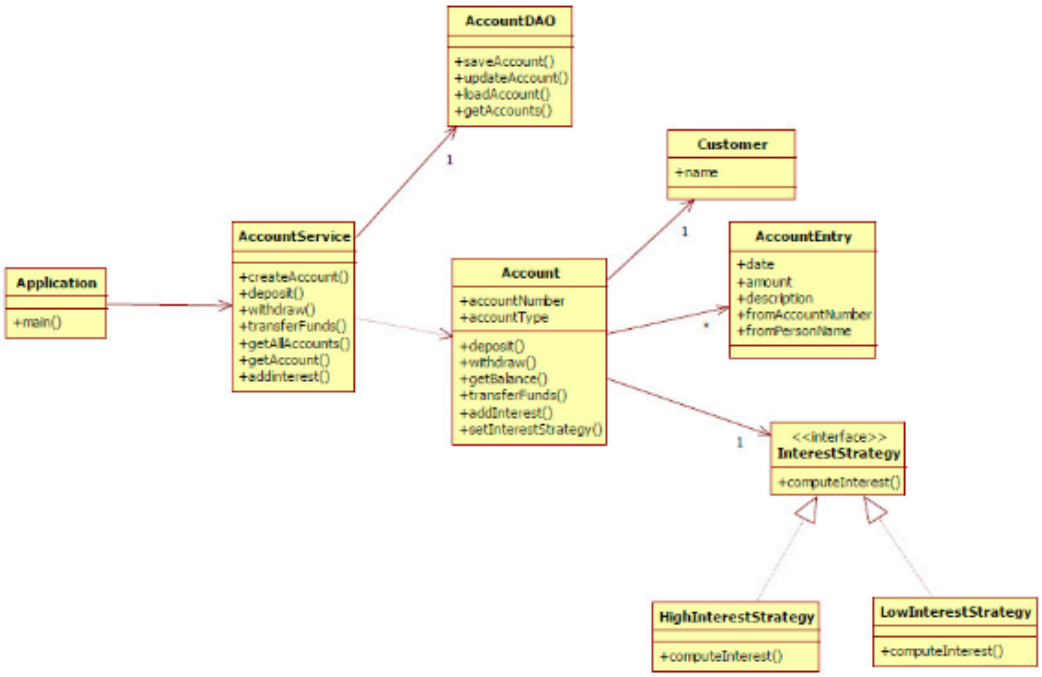
Suppose you have to design this feature of different Layouts for Swing. **Draw the class diagram that shows the important parts of your design.**



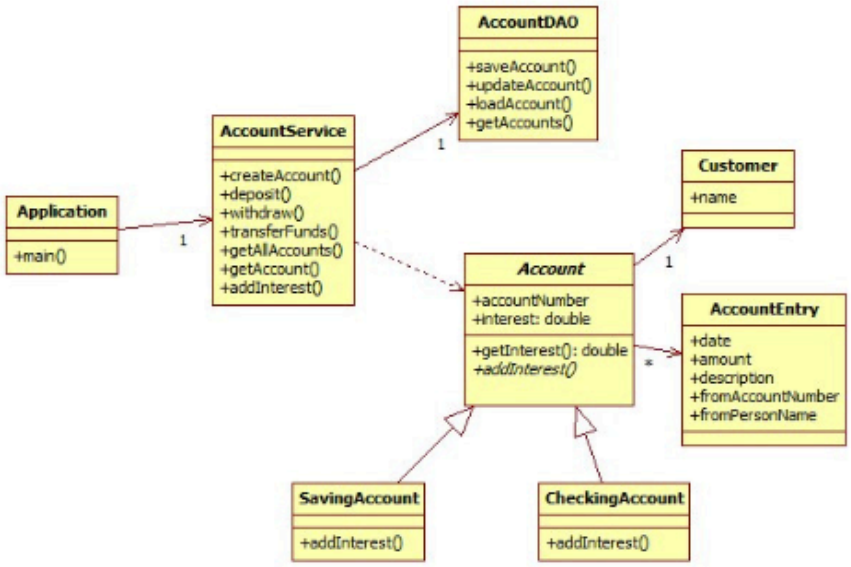
QUESTION 2

Question 2 [15 points] {10 minutes}
In the strategy pattern lab and the corresponding discussion in the forum the we saw 2 different solutions:

Solution 1:



Solution 2:



Explain clearly the advantages of one solution over the other solution.

RESULT 2

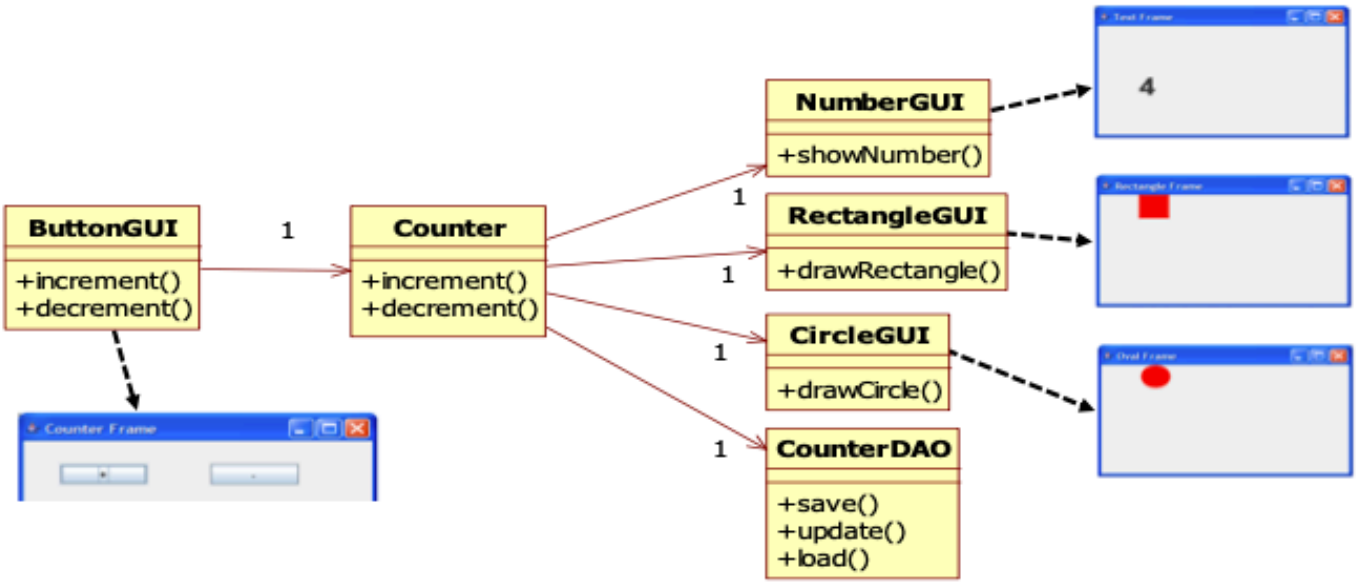
The biggest advantage of solution 2 is that we separate the interest calculation logic from the account logic. This makes it very easy to:

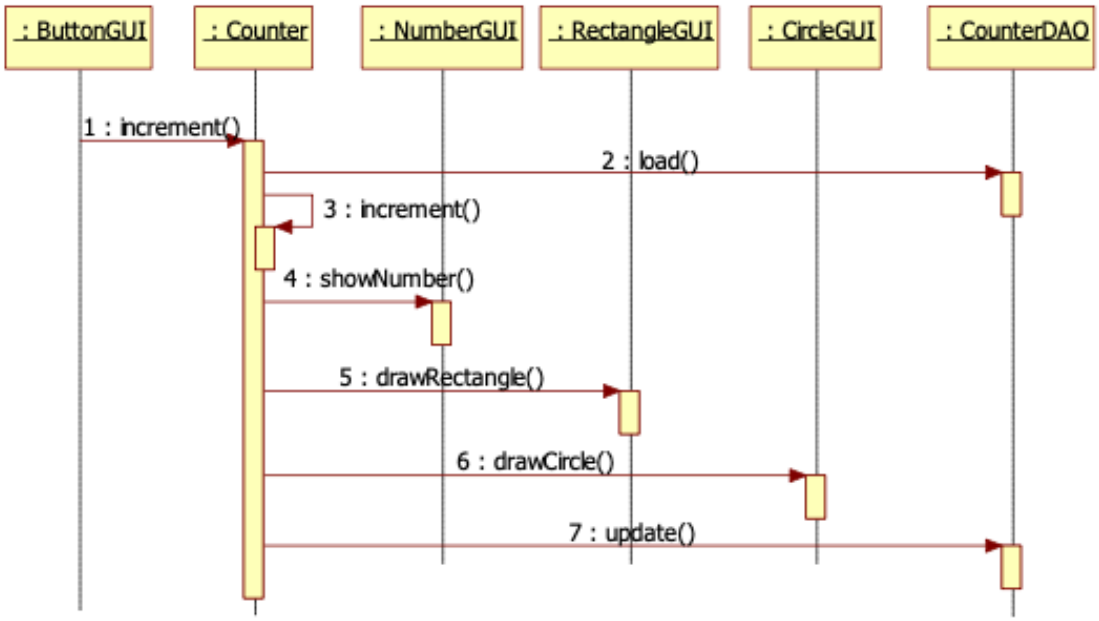
- Add new interest algorithms
- Change the interest algorithm from a certain account without changing the account (open-closed principle)
- Reuse the interest algorithm for something other than an Account or reuse a certain interest algorithm for multiple account types.

These 3 advantages you do not have in solution 1.

Question 3 [65 points] {85 minutes}

Suppose we have an application with the following design:

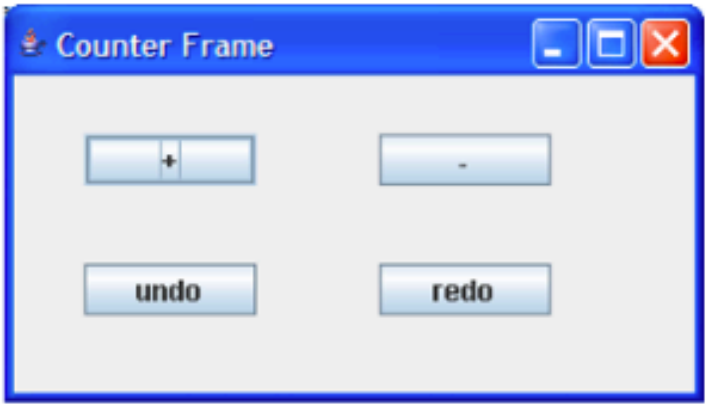




If we press the + button, we call increment() in the ButtonGUI object, which calls increment() on the Counter object. The Counter object loads the current counter value from the database using the CounterDAO, then increments the counter value, and then calls all 3 GUI viewer classes to show the counter value. The NumberGUI shows the current counter value as a number, the RectangleGUI shows a rectangle whose sides are counter value long, and the CircleGUI shows a circle with a diameter of counter value. Finally the current counter value will be updated in the database using the CounterDAO.

The current design does not follow the design principles we studied in this course.
We have to extend our design with the following requirements:

1. We want undo-redo functionality. This means the ButtonGUI will now have 4 buttons:

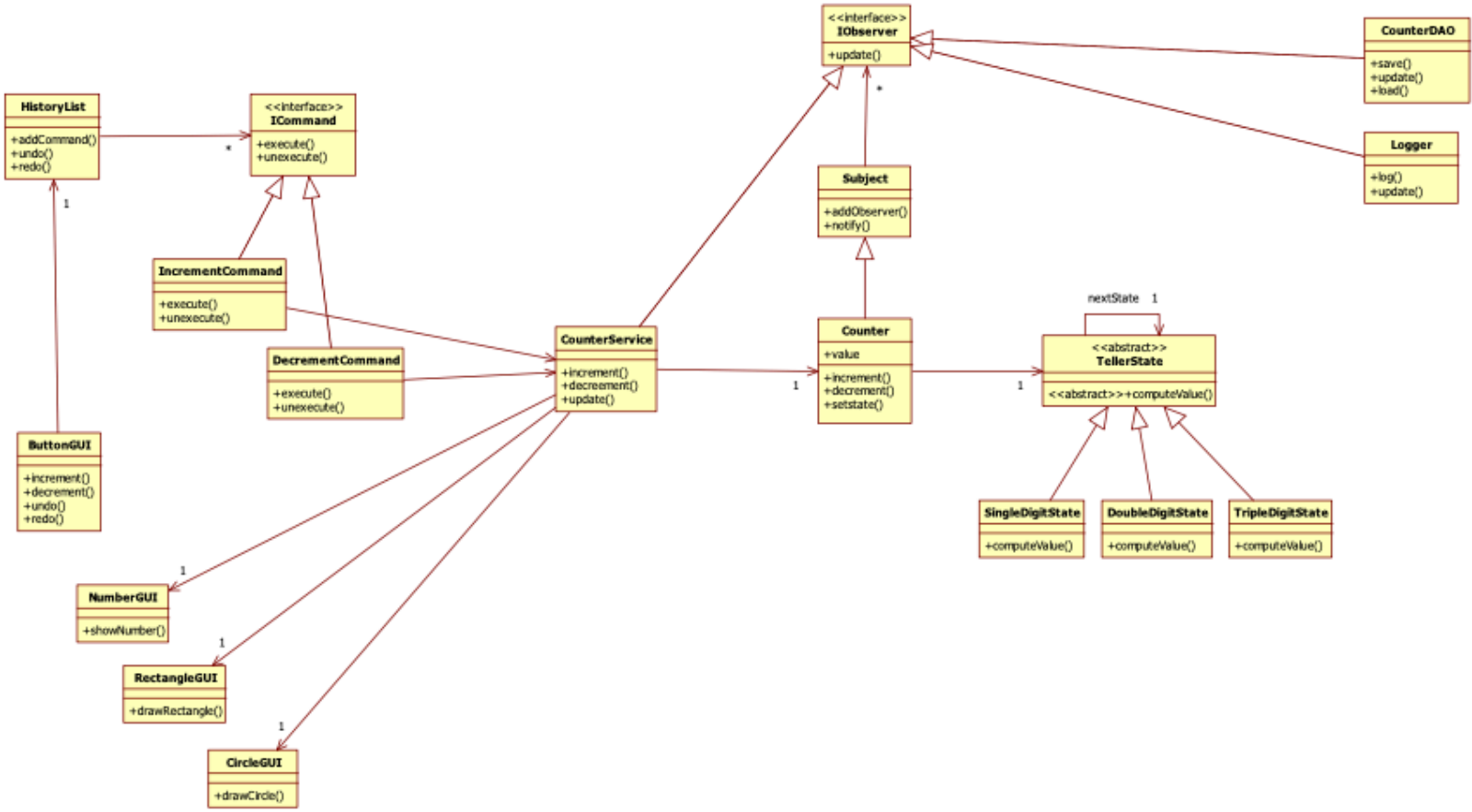


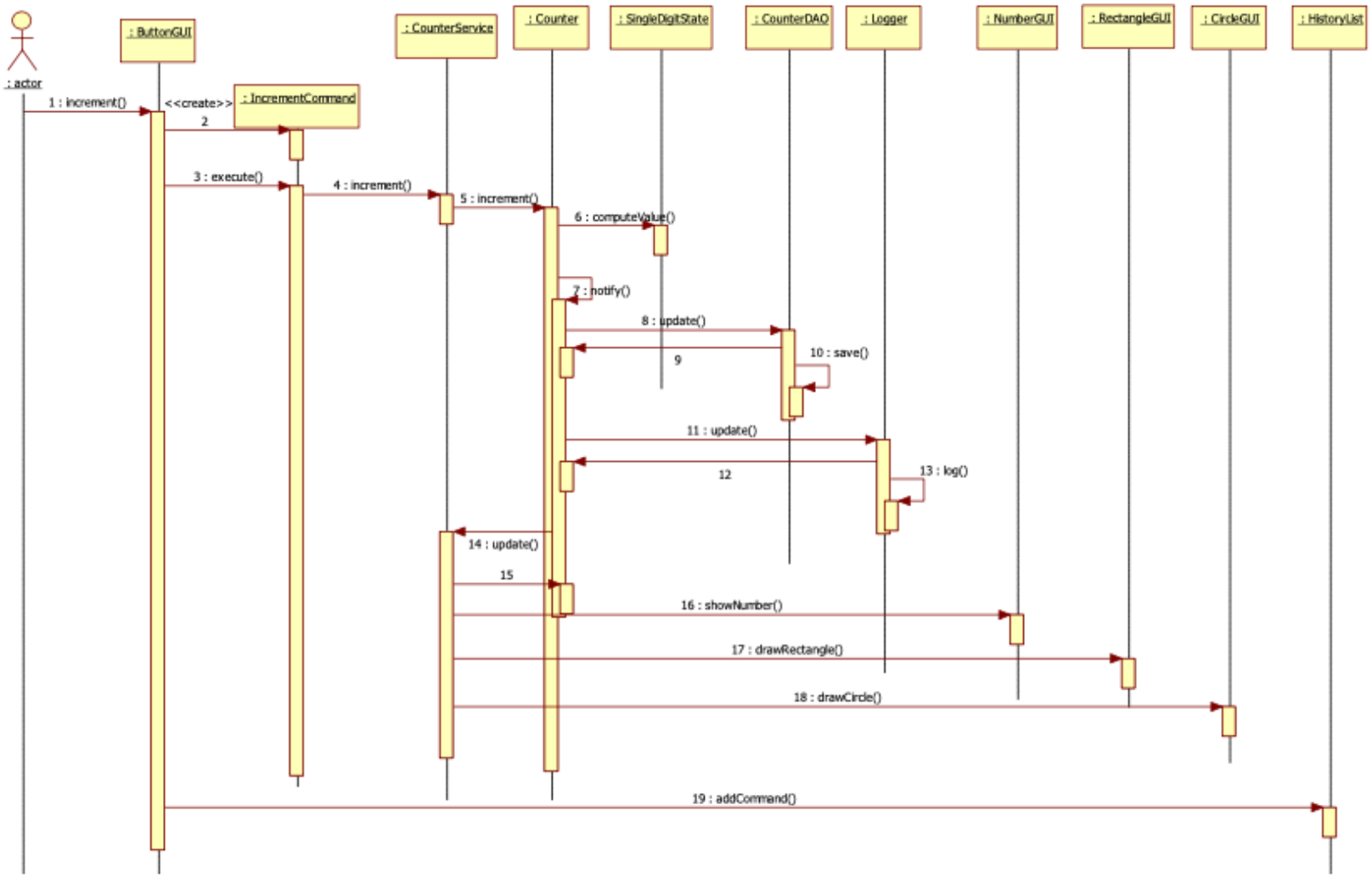
2. It should be easy to reuse the Counter class in another application.
3. We want a clear separation between UI-Business Logic-Data access
4. We want to log every action on the counter in a logfile.
5. When the teller value is a single digit number, then every button action (increment and decrement) will add or subtract 1 point from the current teller value.
   When the teller value is a double digit number, then every button action (increment and decrement) will add or subtract 2 points from the current teller value.
   When the teller value is a triple digit number, then every button action (increment and decrement) will add or subtract 3 points from the current teller value.
   It should be easy to change this algorithm.

Your design should implement the appropriate design principles we studied in the course.

a. Draw the class diagram of your design.
Make sure you add all necessary UML elements (attributes, multiplicity, etc.) to communicate the important parts of your design.
b. Draw the sequence diagram that shows exactly what happens if the user presses the increment('+') button.

RESULT 3


HistoryList
+addCommand()
+undo()
+redo()
<<interface>>
ICommand
+execute()
+unexecute()
IncrementCommand
+execute()
+unexecute()
DecrementCommand
+execute()
+unexecute()
ButtonGUI
+increment()
+decrement()
+undo()
+redo()
CounterService
+increment()
+decreement()
+update()
<<interface>>
IObserver
+update()
Subject
+addObserver()
+notify()
Counter
+value
+increment()
+decrement()
+setstate()
CounterDAO
+save()
+update()
+load()
Logger
+log()
+update()
nextState
<<abstract>>
TellerState
<<abstract>>+computeValue()
SingleDigitState
+computeValue()
DoubleDigitState
+computeValue()
TripleDigitState
+computeValue()
NumberGUI
+showNumber()
RectangleGUI
+drawRectangle()
CircleGUI
+drawCircle()


: actor
: ButtonGUI
: IncrementCommand
: CounterService
: Counter
: SingleDigitState
: CounterDAO
: Logger
: NumberGUI
: RectangleGUI
: CircleGUI
: HistoryList
1 : increment()
<<create>>
2
3 : execute()
4 : increment()
5 : increment()
6 : computeValue()
7 : notify()
8 : update()
9
10 : save()
11 : update()
12
13 : log()
14 : update()
15
16 : showNumber()
17 : drawRectangle()
18 : drawCircle()
19 : addCommand()

Question 4 [5 points] {10 minutes}

Describe how we can relate the **State pattern** to one or more principles of SCI. Your answer should be about half a page, but should not exceed one page (handwritten). The number of points you get for this questions depends how well you explain the relationship between the **State pattern** and one or more principles of SCI.