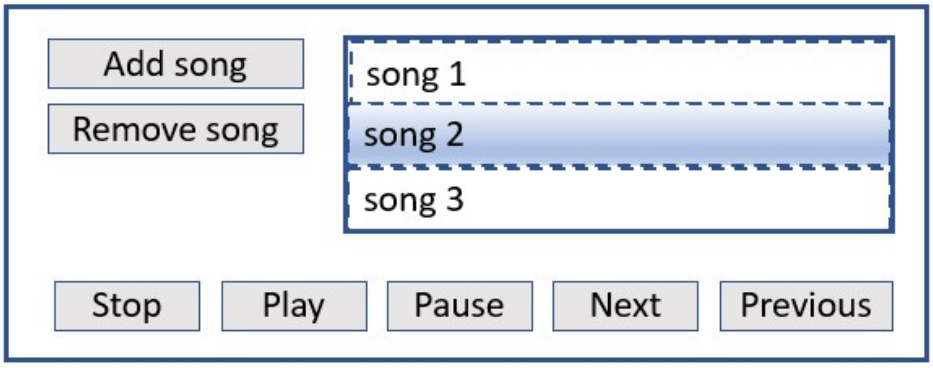
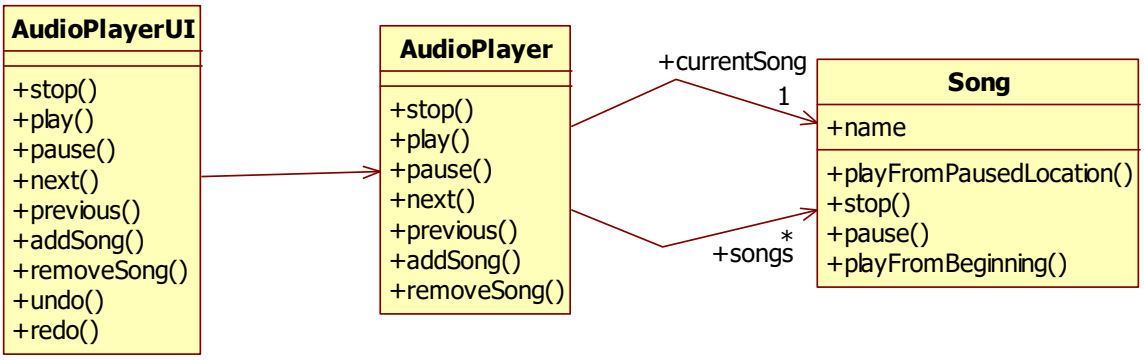
Question 1 [45 points] {50 minutes}

Suppose you need to design and implement an audio player that has the following user interface:



You can add and remove songs to a list, and the buttons **Stop, Play** and **Pause** operate at the selected song. You can change the selected song with the **Next** and **Previous** buttons.

Your first design looks like this:



The problem with this design is that the AudioPlayer class contains a lot of conditional logic. For example if we click the Play button, it depends on the current state of the audio player what will happen. The play() method looks like this:

```
public void play() {
  if (currentState.equals("stop")) {
        currentSong.playFromBeginning();
        currentState = "play";
  } else if (currentState.equals("pause")) {
        currentSong.playFromPauzedLocation();
        currentState = "play";
  }
}
```

We learned that if we use the State pattern, we can get rid of this complex conditional logic.

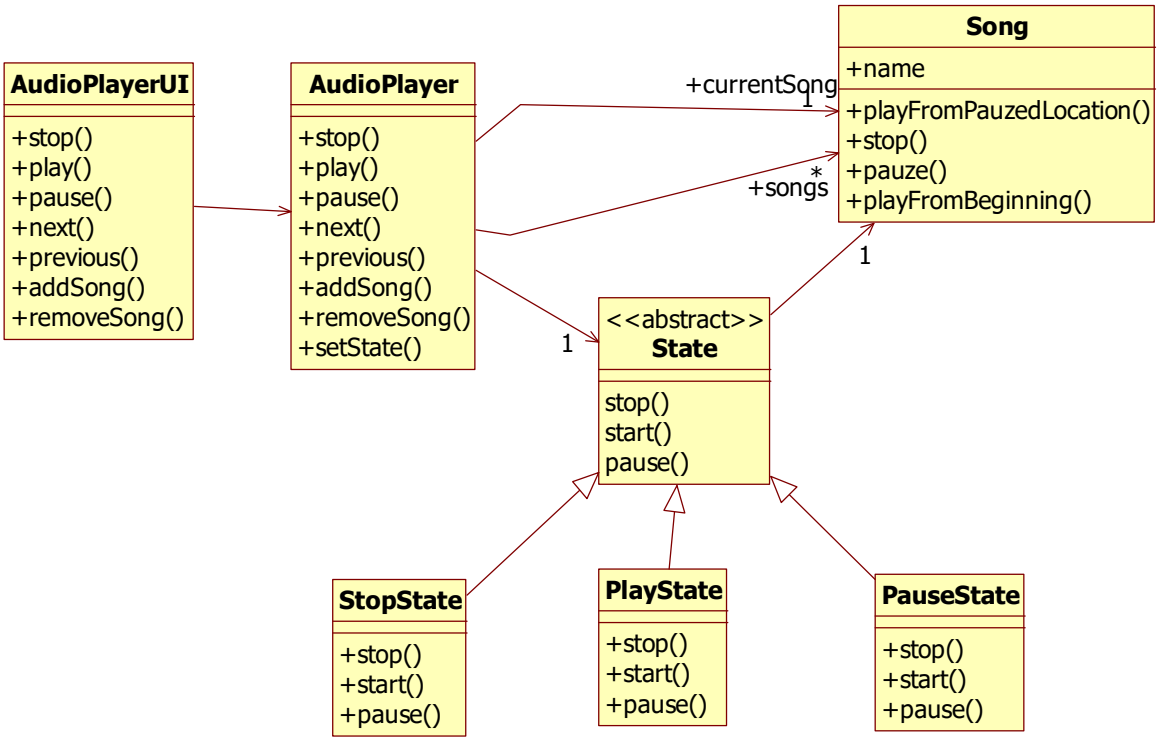
a. [5 points]

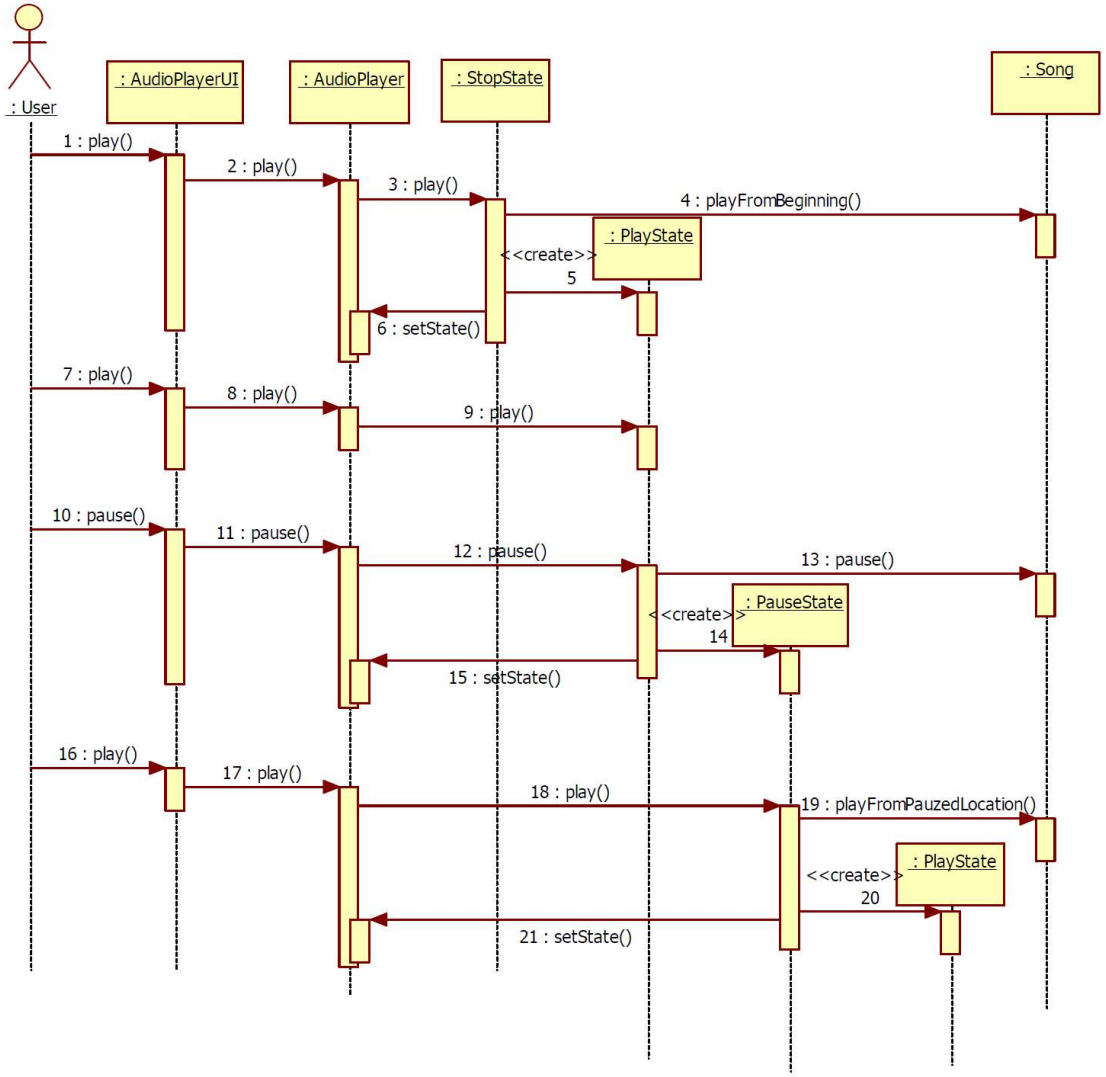
Draw the class diagram of your design with the state pattern applied

b. **[35 points**]

Draw the sequence diagram that shows clearly how your design works if you use the state pattern in this application. The initial state of the audio player is stop, so assume the audio player is already in the stop state when the sequence diagram starts. Show the sequence diagram of the following user actions:

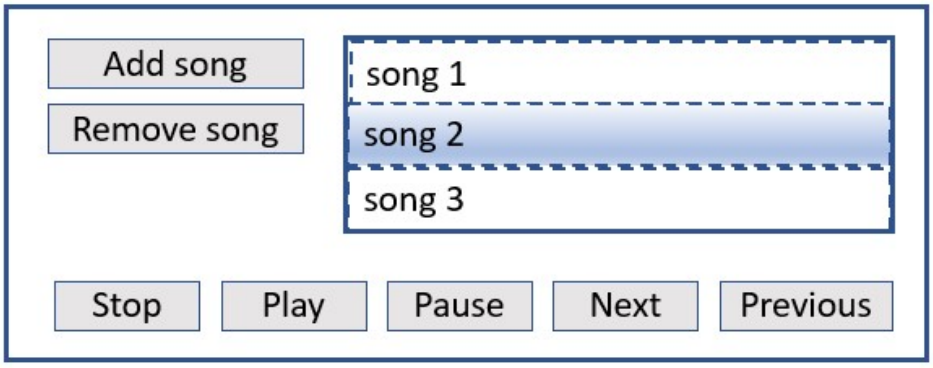
1. The user presses the play button
2. The user presses the play button again
3. The user presses the pause button
4. The user presses the play button




: User
: AudioPlayerUI
: AudioPlayer
: StopState
: Song
1 : play()
2 : play()
3 : play()
4 : playFromBeginning()
: PlayState
<<create>>
5
6 : setState()
7 : play()
8 : play()
9 : play()
10 : pause()
11 : pause()
12 : pause()
13 : pause()
: PauseState
<<create>>
14
15 : setState()
16 : play()
17 : play()
18 : play()
19 : playFromPauzedLocation()
: PlayState
<<create>>
20
21 : setState()

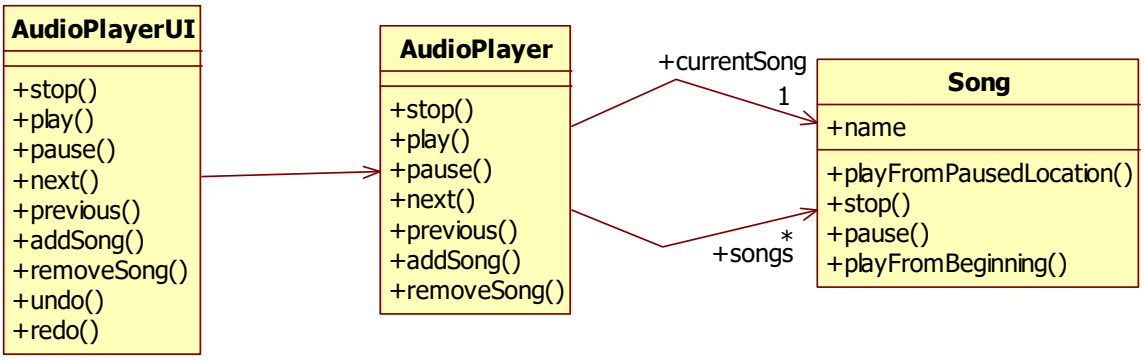
Question 2 [40 points] {50 minutes}

Suppose you need to design and implement an audio player that has the following user interface:

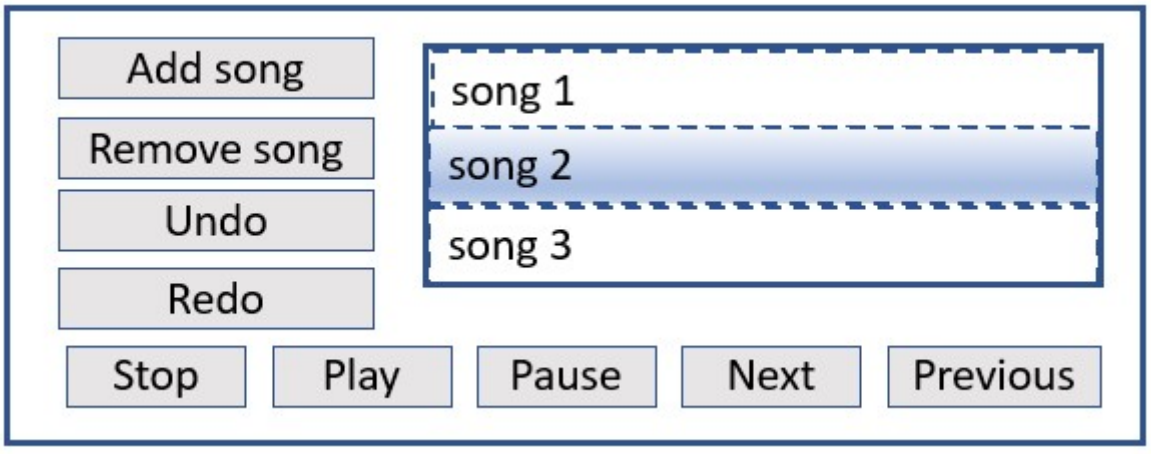


You can add and remove songs to a list, and the buttons **Stop, Play** and **Pause** operate at the selected song. You can change the selected song with the **Next** and **Previous** buttons.

Your first design looks like this:



But now you get a new requirement that the audio player also need undo/redo support.



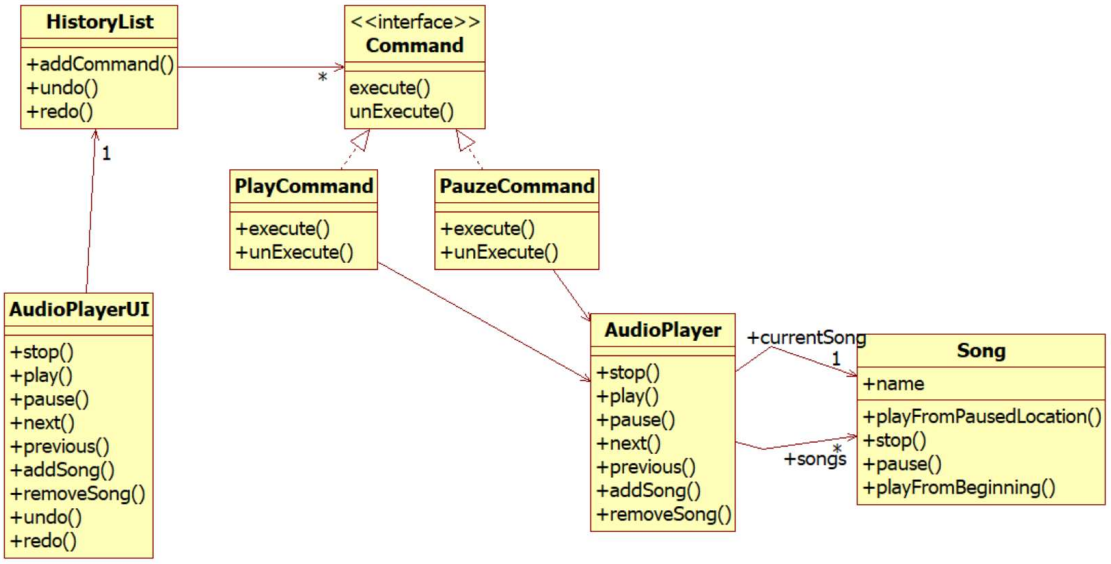
In question 1 we saw already that the state pattern can be used to get rid of the complex conditional logic in the AudioPlayer class. For this question you can assume that we don't care about this conditional logic. For this question, do **NOT apply the state pattern**.

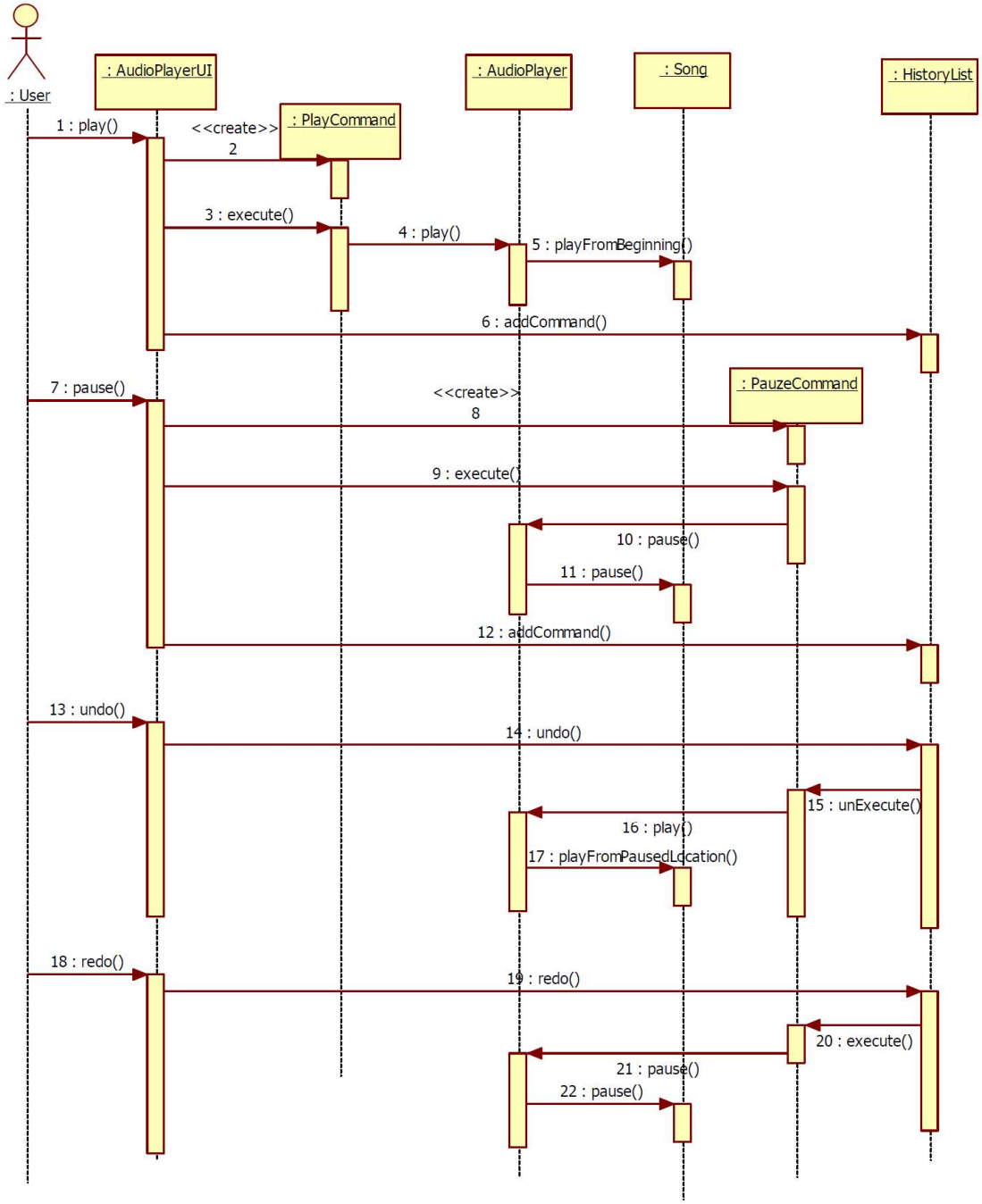
For this question we only need to implement the **undo/redo** functionality.

c. [5 points]
Draw the class diagram of your design so that your design supports undo/redo functionality.

d. **[35 points**]
Draw the sequence diagram that shows clearly how your new design works. Suppose the last action that is done is clicking the Stop button. So your sequence diagram starts when the audio player is in the Stop state. Show the sequence diagram of the following user actions:

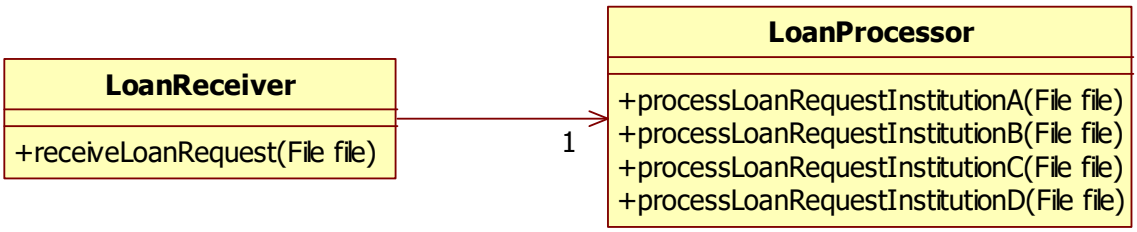
1. The customer clicks the Play button
2. The customer clicks the Pause button
3. The customer clicks the Undo button
4. The customer clicks the Redo button




: User
: AudioPlayerUI
: PlayCommand
: AudioPlayer
: Song
: HistoryList
: PauzeCommand
1 : play()
<<create>>
2
3 : execute()
4 : play()
5 : playFromBeginning()
6 : addCommand()
7 : pause()
<<create>>
8
9 : execute()
10 : pause()
11 : pause()
12 : addCommand()
13 : undo()
14 : undo()
15 : unExecute()
16 : play()
17 : playFromPausedLocation()
18 : redo()
19 : redo()
20 : execute()
21 : pause()
22 : pause()

Question 3 [10 points] {10 minutes}

Suppose we have an existing loan application with the following design:



The loan application receives requests from different financial institutions, creates a loan offer and send this load offer back to the institution that submitted the request. The basic problem is that the loan offer request we receive from every institution is different. They all send a file, but they use different file formats, different data formats and the data is structured in a different way for every institution. The current application works fine because we receive loan requests only from 4 different institutions (A to D).

Now you get a new requirement that this application will have to support loan requests from many different institutions. In the short time that might be 100 different institutions, but the number of institutions might grow to thousands in the long term.

It is your job to create a better design than the existing application. Your design should support processing loan offers from many financial institutions and it should be easy to add support for processing loan offers from new financial institutions.

Give the **name of the pattern** you are going use to improve your design.

Chain of responsibility.

Strategy is not solving this problem. With the strategy you still need conditional logic to decide which strategy clas to use.