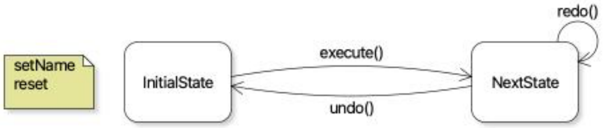
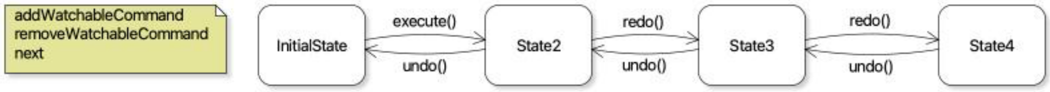
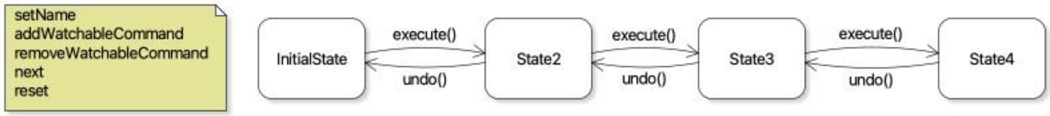
Q1(Observer Pattern):
Once a watchable object has been watched, we need to update all watchlists which contain the watched object. As a consequence, we can implement the Observer design pattern here. The watchable objects are subjects, and I will keep the watchable interface as the subject interface. Then the watchlists are observers, and I create an Observer interface to enforce the watchlist have some methods. There’re some differences between my implementation and the implementation we covered in class. The implementation we covered in class enable the clients to add observers or remove observers. However, in my implementation, when a watchable object has been added to a watchlist, it will add such watchlist as observer automatically. It’s like each watchable object will remember the watchlist which contains them. Once the watchable object is watched, it will notify all watchlists which contain it. The method watch() will include the function to notify all observers.

Q2(Inheritance):
For this part, to easily maintain our program in future, I don’t change the watchable interface. Instead, I create an abstract class called AbstractWatchable which implements the watchable interface. Furthermore, I abstract the common fields Title, Language, Studio and Tags (By checking the code, I think the Tags in Episode and Movie are same thing as the Info in TVShow) to the abstract class. Additionally, all methods related to the common fields will be abstracted as well. Furthermore, since we implemented the Observer pattern in question1, there are some common fields for subjects. As a result, I write the common part of methods watch(), acceptWatchlist() and removeWatchlist(). However, all of these three methods might be overridden in the subclass.

Q3(Command Pattern):
We need to enable the undo and redo action for this part. As a consequence, I implemented the Command pattern for this part. I create a command interface first, and each command should have execute(), undo() and redo() methods. I decided to enforce all command have the redo() method is because the redo action of some command will still change the state but some not. For example, the setName command will only change the state once. Calling redo action of the setName will not change the state. However, calling redo action of removeWatchableCommand will still change the state (still remove the watchable object from the list). In my implementation, I distinguished these two kinds of commands. If the redo action does not change the state, the undo action will directly change back to the last state no matter we call the redo action how many times. If the redo action changes the state, the undo action will only change back to the last state. I draw the state diagram to illustrate my idea.







For example, the initial name is l1. We call the setName("l2"), and then redo twice. Once we call the undo, the Name will be set back to l1 directly. If we add a Watchable object to the list, and redo twice. Once we call the undo, the list will only remove the last object we added.