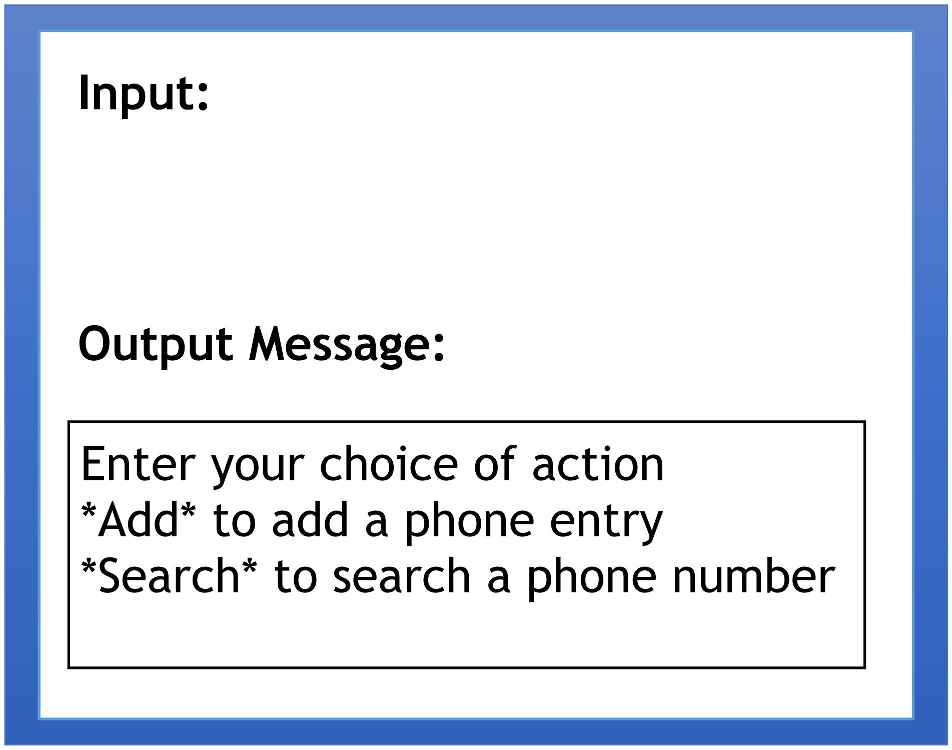
CS5551 Advanced Software Engineering

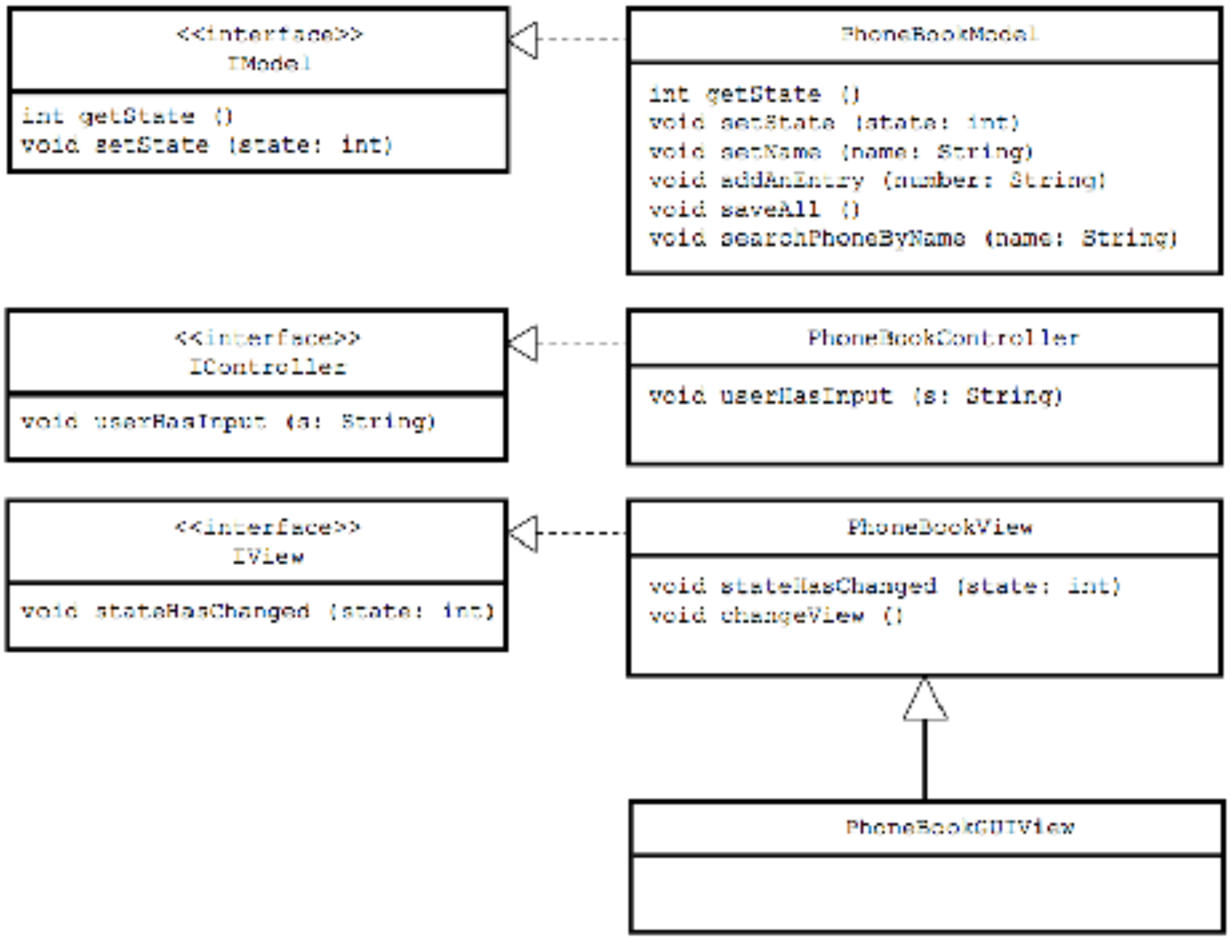
Problem Set 4 (PS-4)
Deadline: April 3 (T)
Submit a hard copy of your solutions to the instructor during the class

Name: Sneha Mishra
Class ID: 21
You are supposed to develop a simple desktop phone book application. It includes a text-based interface that allows the user to add name/phone pairs and search for phone by name. The data persist in a text file. The application's architecture is based on the popular Model-View-Controller (MVC) architectural design pattern; the Model is responsible for data processing and persistence, the View is responsible for the user interface (presentation and collection of user input), and the Controller is responsible for executing actions according to user input and current state of the application and all communications between the three components. These include both generic behavior at the abstract interfaces level and application specific behavior at the concrete classes level, as we demonstrate next.

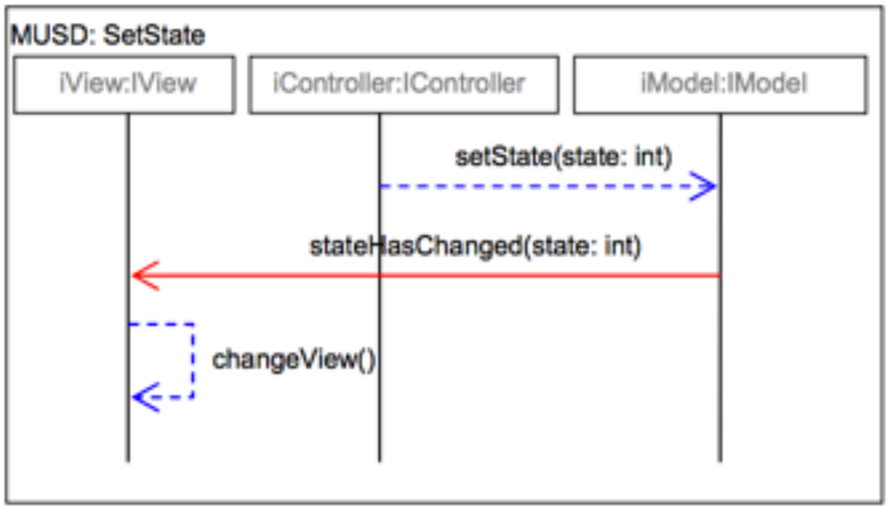
1. Draw the Phone Book GUI.


Input:
Output Message:
Enter your choice of action
Add to add a phone entry
Search to search a phone number

2. Draw the Phone Book application UML class diagram; it includes three interfaces and three concrete classes that implement them, plus another class, PhoneBookGUIView, which extends PhoneBookView

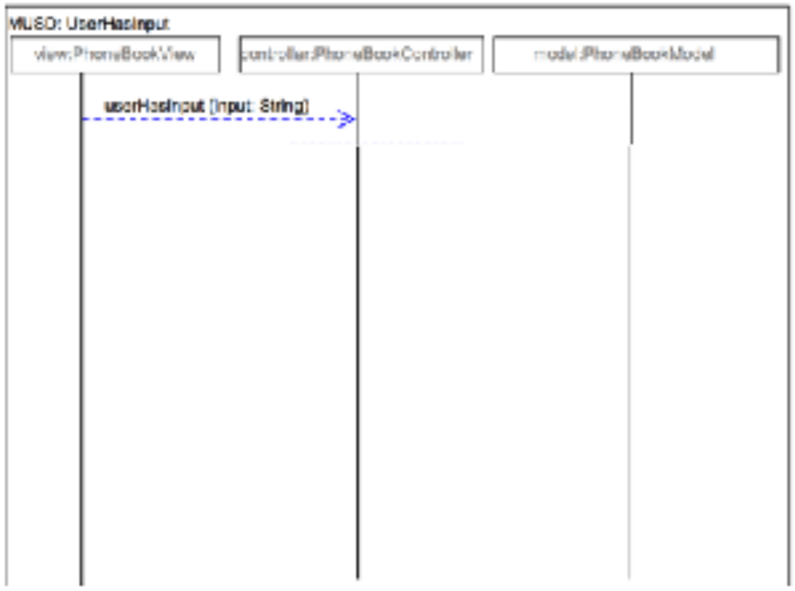

<<interface>>
IModel
int getState ()
void setState (state: int)
PhoneBookModel
int getState ()
void setState (state: int)
void setName (name: String)
void addAnEntry (number: String)
void saveAll ()
void searchPhoneByName (name: String)
<<interface>>
IController
void userHasInput (s: String)
PhoneBookController
void userHasInput (s: String)
<<interface>>
IView
void stateHasChanged (state: int)
PhoneBookView
void stateHasChanged (state: int)
void changeView ()
PhoneBookGUIView

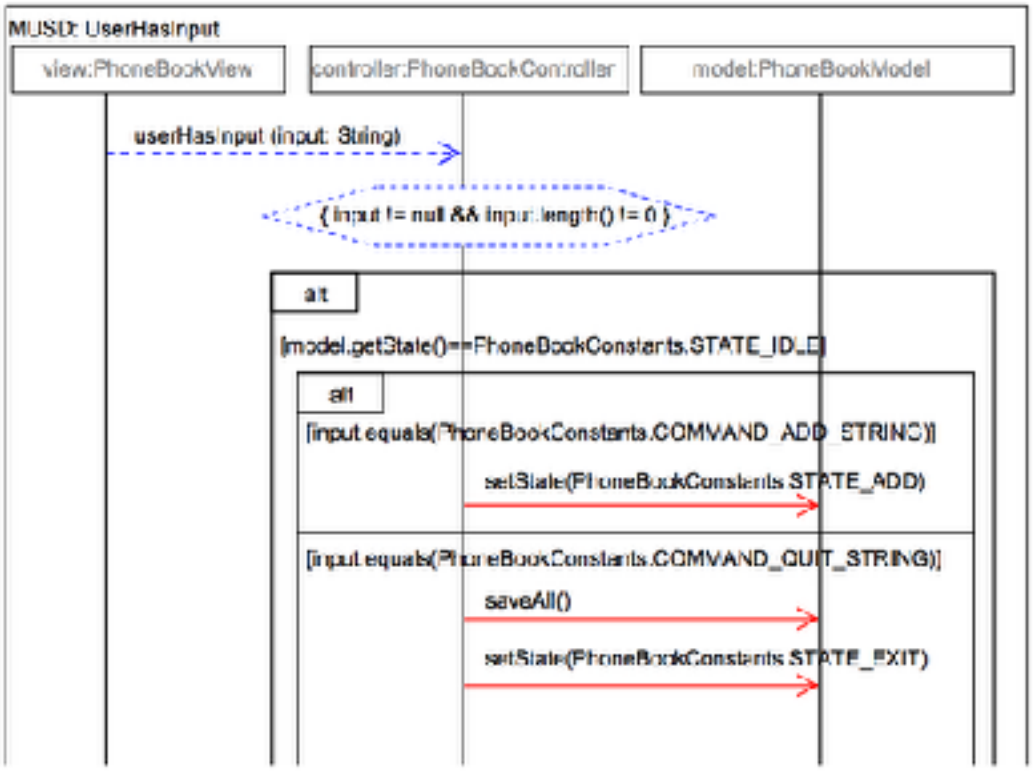
3. The SetState includes three methods: setState() (cold/monitoring), stateHasChanged() (hot/execution), and changeView() (cold/monitoring). Explain how a MVC pattern is specified in the SetState design.



It specifies that whenever the IModel method setState() is called from an IController (more precisely, whenever an object that implements the IController interface calls the setState() method of an object that implements the IModel interface), with whichever int argument, the IModel should eventually call the IView method stateHasChanged() with the same integer as argument. The IView may then call its changeView() method. The behavior is specified at the interface level since it is generic and independent of a specific MVC-based application.

4. The application specific inter-object behavior of the Phone Book is specified in the UserInput MUSD. It specifies that whenever the controller's userHasInput() method is called by the viewer, and the input is not empty, one of several alternative scenarios should happen, depending on the current state of the application and the input . For example, if the current state is STATE IDLE and the input is COMMAND QUIT STRING (defined as "quit"), the controller should call the model's saveAll() method and then call its setState() method with the new state STATE EXIT as argument. Complete the following diagram for the UserInput MUSD.




MUSD: UserHasInput
view:PhoneBookView
controller:PhoneBookController
model:PhoneBookModel
userHasInput (input: String)
{ input != null && input.length() != 0 }
alt
[model.getState()==PhoneBookConstants.STATE_IDLE]
alt
[input.equals(PhoneBookConstants.COMMAND_ADD_STRING)]
setState(PhoneBookConstants.STATE_ADD)
[input.equals(PhoneBookConstants.COMMAND_QUIT_STRING)]
saveAll()
setState(PhoneBookConstants.STATE_EXIT)

5. Describe the advantages and disadvantages of the MVC pattern in the application specific inter-object behavior of the Phone Book.

The Phone Book case study shows how S2A realizes MUSD behavioral contract enforcement and reuse across class hierarchies and interface implementations. Since the lifelines in the SetState MUSD represent interfaces, this diagram can be explicitly reused, as is, together with the three interfaces, in other applications that employ the MVC design pattern. Reusing the SetState MUSD enforces the correct use of the pattern; S2A generates the code that actually performs the required behavior and thus ensures that the behavioral aspect of the pattern is indeed preserved in the implementation. As future work we thus envision assembling a repository of reusable MUSD specifications of popular behavioral patterns, such as MVC, from which one could choose and then reuse when constructing new applications. The Phone Book application uses the PhoneBookGUIView class to implement a Java Swing interface. The main() method of the application initiates this class. Since it extends the PhoneBookView class (and thus implicitly also implement the IView interface), the corresponding lifelines in the two MUSDs bind to it at runtime. Thus, the diagrams can be reused both with new interface implementations and with subclassing. Finally, the case study shows the applicability of MUSD to popular frameworks that implement the MVC pattern, most importantly, to server-side programming frameworks of web-based applications (e.g., Struts, Spring). Specifying such a server-side application using MUSD and constructing it using S2A is a project we plan to pursue.