

Step 1: Break example user interface into less complex components

From: JLabel + JTextField

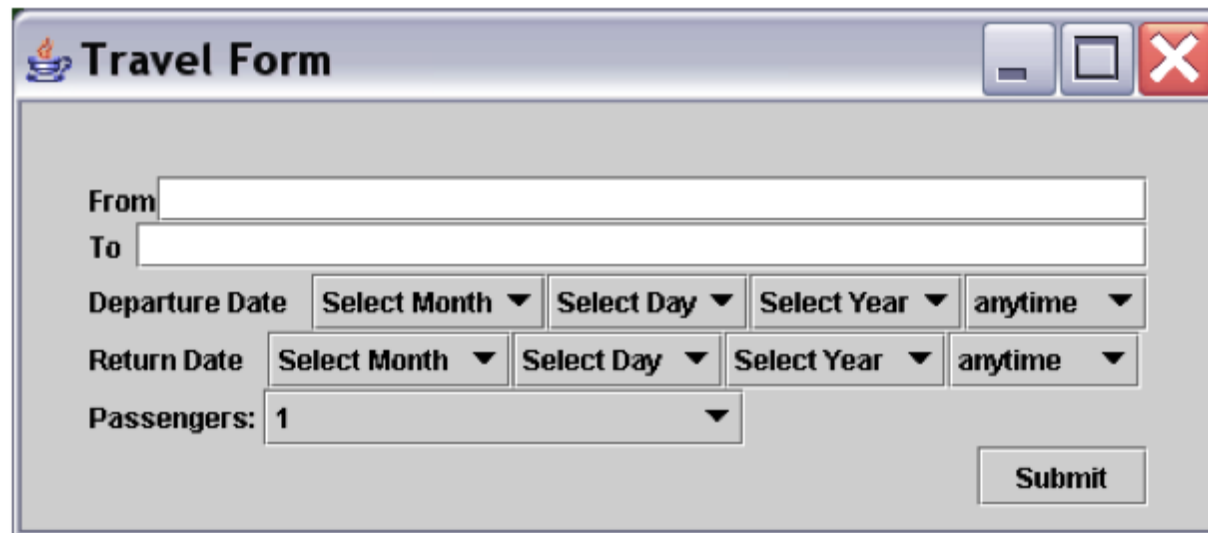
To: JLabel + JTextField

Departure date: JLabel + JPanel containing a custom date picker

Return date: JLabel + JPanel containing a custom date picker

Passengers: JLabel + JComboBox

Submit: JButton



The screenshot shows a Java Swing window titled "Travel Form" with a standard Mac OS X-style title bar (minimize, maximize, close buttons). The window has a light gray background. The form contains the following elements:

- From:** A text label followed by a white text input field.
- To:** A text label followed by a white text input field.
- Departure Date:** A text label followed by four date selection components: "Select Month" (dropdown), "Select Day" (dropdown), "Select Year" (dropdown), and "anytime" (dropdown).
- Return Date:** A text label followed by four date selection components: "Select Month" (dropdown), "Select Day" (dropdown), "Select Year" (dropdown), and "anytime" (dropdown).
- Passengers:** A text label followed by a dropdown menu showing the value "1".
- Submit:** A rectangular button with the text "Submit" located at the bottom right of the form area.

Figure 1: Screen shot of application when started

Step 2: Implement custom date picker

a. Determine what data needs to be collected and stored

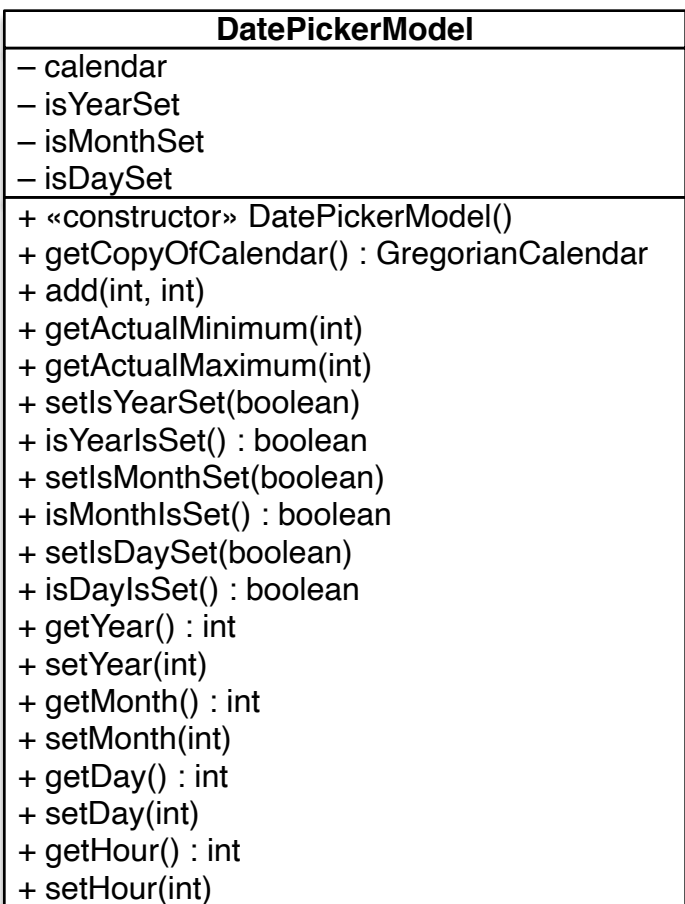
- Month, day of month, year, and time of day.

b. Is there an existing Java class that can handle this information?

- Yes, the `GregorianCalendar` class.

c. Create model

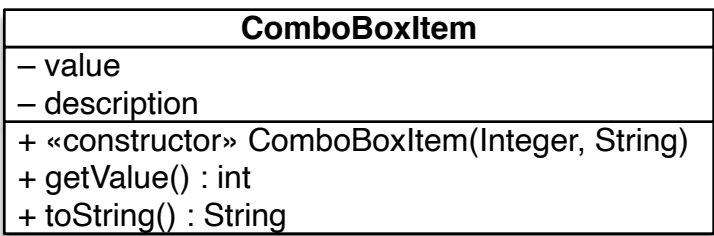
- The model needs to keep track of the date and time represented by the values selected in the UI.
- Observe that the initial values of the month, day of month, and year JComboBoxes are "Select XXX". This means that the date picker model also needs to keep track of whether each of the selected values is a valid date part, or if the "Select XXX" item is selected.
- **DatePickerModel** consists of one instance of `GregorianCalendar` to keep track of the date and time and three boolean flags to indicate whether the selected month, day of month, and year are valid date parts.



Step 2 (continued)

d. Determine how to populate the JComboBoxes

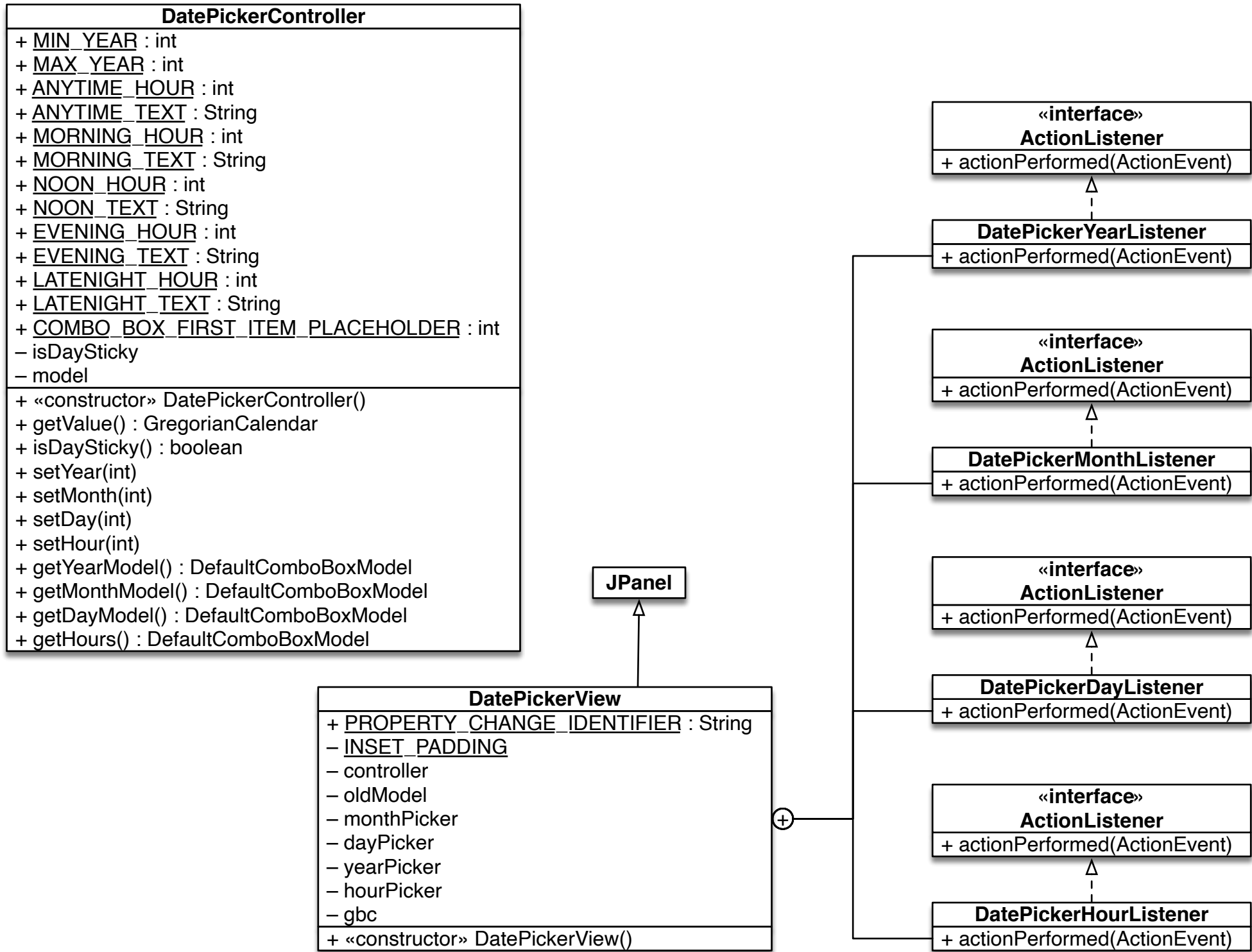
- Consult books *Introduction to Java Programming, Sixth Edition* (Liang 2007) and *Swing: A Beginner's Guide* (Schildt 2007) to learn how to populate a JComboBox. Reject using an array or vector because that seems inelegant; use a DefaultComboBoxModel instead.
- Observe that there needs to be some translation between the choice displayed by the JComboBox and the actual value that gets stored. For example, the month picker needs to display the names of the months in English, but the month value stored by the GregorianCalendar object needs to be an integer. Solution: create a class **ComboBoxItem** that stores a value and a description.



- Backing each JComboBox is a DefaultComboBoxModel that contains one ComboBoxItem for every choice.

e. Create view and controller

- **DatePickerView** consists of a JPanel that contains four JComboBoxes in a horizontal row, one each for month, day of month, year, and time and a reference to the controller. At this point the JComboBoxes are not populated and there are no listeners implemented.
- **DatePickerController** contains the business logic and mediates communication between the model and the view.
 - Implement code to populate the view's JComboBoxes.
 - Implement code to propagate changes in the UI to the model.
- In **DatePickerView**, implement ActionListeners to trigger the controller to update the model when a new choice is selected in a JComboBox.



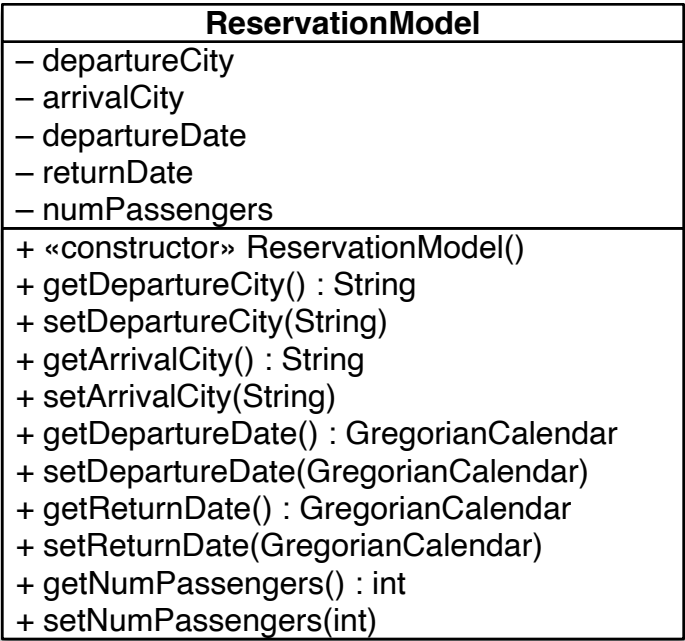
Step 3: Implement reservation maker

a. Determine what data needs to be collected and stored

- Departure and arrival cities, departure and return dates, and number of passengers

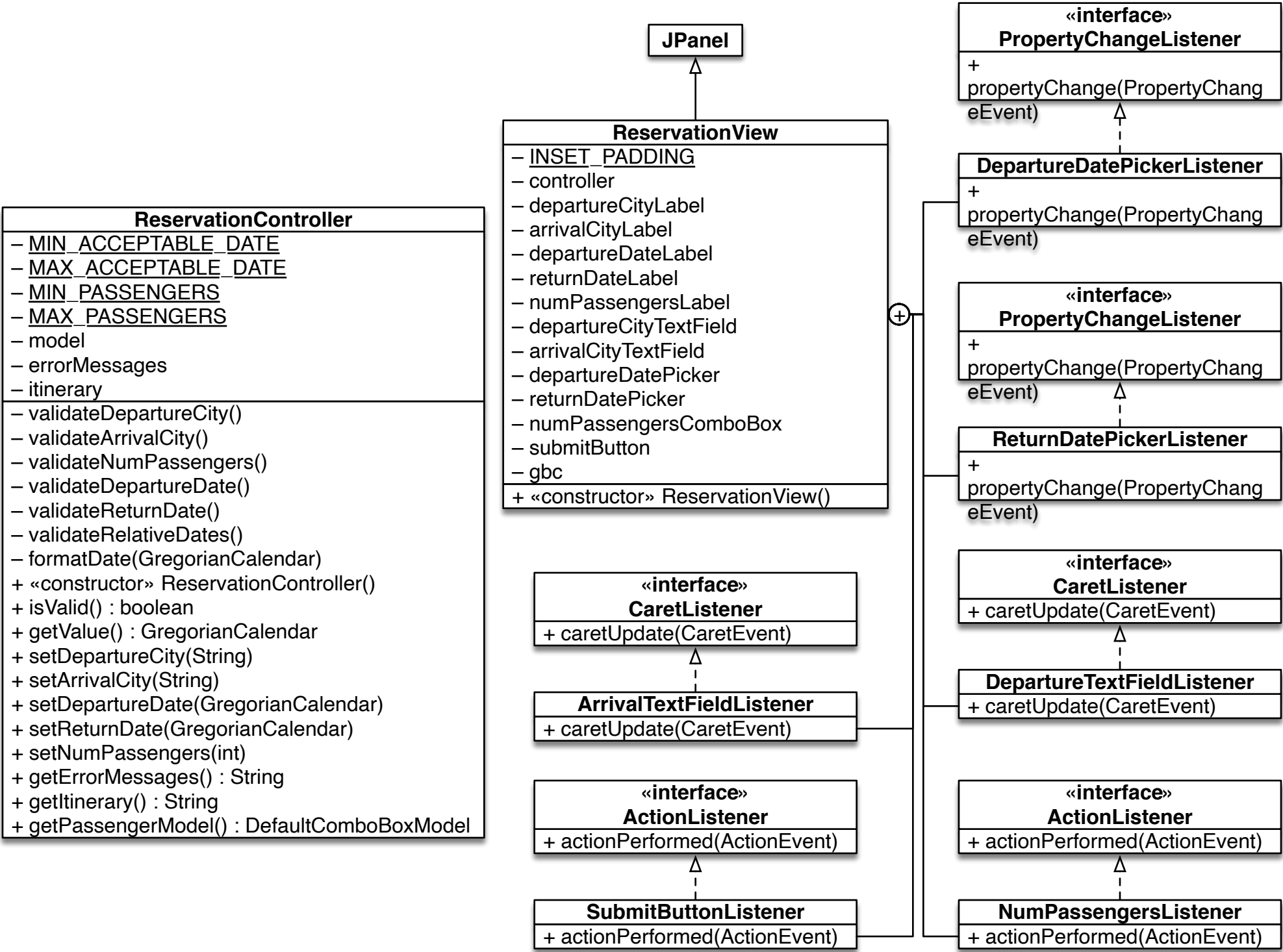
b. Create model

- ReservationModel** Stores cities as Strings, dates as GregorianCalendar, and number of passengers as an int.



c. Create view and controller

- ReservationView** consists of a JPanel that contains the UI components identified in step 1.
- ReservationController** contains the business logic and mediates communication between the model and the view.
 - Implement code to populate the number of passengers JComboBox.
 - Implement code to propagate changes in the UI to the model.
 - Observe that the **ReservationModel** needs to know when the date represented by a date picker is updated but we do not want to tightly couple the reservation model with the date picker. Need to implement some sort of observer that gets notified whenever the underlying **DatePickerView** (and its model) is updated but JPanels do not have ActionListeners. After unsuccessfully trying to figure out the answer myself, I posted a question on stackoverflow.com to find out how to update a parent JPanel when a component inside a child JPanel is updated. Thanks to @Hovercraft Full of Eels for suggesting a PropertyChangeListener.
 - Update **ReservationView** and **DatePickerView** to make use of a PropertyChangeListener to enable the **ReservationModel** to be updated any time one of the encapsulated **DatePickerViews** is updated.
- Implement code to validate the model and produce either an error message or itinerary depending on whether the model validates according to the provided business rules.



Step 4: Put everything together

- Create a Main class so people can easily figure out where the program starts.

