Lab Exercise 5 – Model, View, Controller plus Menus, Toolbars, and Dialogs

Computer Science 2334

Due by: Friday, March 24, 2017, 11:59 pm

This lab is an individual exercise. Students must complete this assignment on their own.

Objectives:

1. To review how the model, view, and controller interact in the Model, View, Controller design pattern.
2. To review how to create a dialog box based on the **JOptionPane** class.
3. To learn how to create a menu system using **JMenuItem**, **JMenu** and **JMenuBar**.
4. To learn how to create a toolbar system using **JToolBar**.
5. To demonstrate this knowledge by completing a series of exercises.

Instructions:

This lab exercise requires a laptop with an Internet connection. Once you have completed the exercises in this document, your team will submit it for grading.

Make sure you read this lab description and look at all of the source code posted on the class website for this lab exercise before you begin working.

Assignment:

Graphical User Interfaces using the Model, View, Controller (MVC) design pattern are an important programming abstraction that shows how additional structure can be built from objects; it is also one that will be used in future projects. Carefully inspect how it works and the documentation comments included in the code.

1. Download the `Lab5-eclipse.zip` project archive from the class website. Import the project into your Eclipse workspace using the slides from Lab 2. You will submit the modified project archive when you are finished.

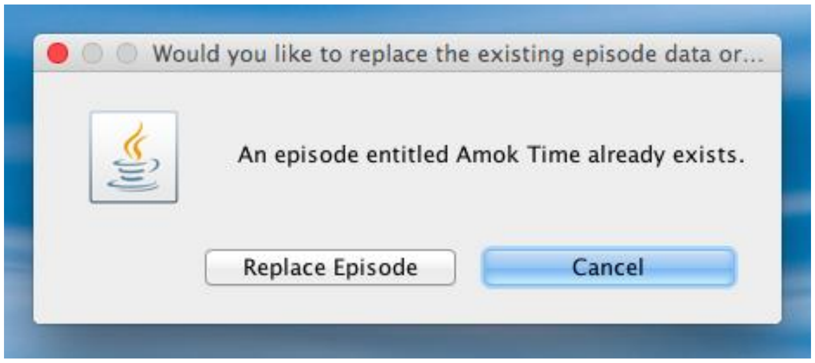
2. Review the source code for the **Lab5Driver** class. Note that it is identical (except for comments) to the **Lab4Driver** class. Also note that you won't change the driver class from what it was in Lab 4, even though this program will have new and different functionality than the previous one. This is because in MVC, the driver just creates the model(s), view(s), and controller(s) and tells them about one another. After that point, they do *all* the work.

3. Review the source code for the **Episode** and **Series** classes, which are classes for representing (some aspects of) episodes and series. The **Series** class is extended by the **SeriesModel** class. This class is the data model for the program. It extends the **Series** class by adding variables and methods, and by overriding methods, in order to deal with the GUI. You will use methods provided in these classes to complete the code for the lab. Note that this is mostly the same data model used in Lab 4, although some methods have been added to or altered in **Series** to provide or modify functionality.

4. Modify **Series** further by making it serializable and by adding methods for object IO and for printing its data to the terminal. See the "`TODO`" comments in **Series** for details.

5. Review the source code for the **SeriesInputWindow** class, which is a class that presents a GUI window to the user for adding new episodes to a series or for clearing out the collection of episodes. This window provides a view for the model data. Note that this is the same view used in Lab 4, with a few minor modifications.

6. Read through the source code for the **AddEpisodeListener** class, which is a class that is listens for the "Add Episode" button to be pressed in the **SeriesInputWindow**. In Lab 5, this class may present a GUI dialog for clarifying user intent when interacting with the **SeriesInputWindow**. In Lab 4, when a user entered an episode with the same name as one already in the episode collection, the old entry would simply be replaced by the new entry. In Lab 5, when the user enters an episode with the same name as one already in the episode collection, the user will be presented with a dialog window that will ask if the intent is really to replace the existing episode. The other option for the user will be to cancel the addition. All of these options will be presented using an object of the **JOptionPane** class, which is a new addition for Lab 5. You will create this **JOptionPane** in Step 8, below. An example of this dialog object is shown below. As you read through the source code for this class, note the "TODO" comments provided there that give hints as to what needs to be done in the program. The guide at http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html gives good examples of using **JOptionPane**s.



An episode entitled Amok Time already exists.

7. Is it necessary to create a **JOptionPane** instance? Why or why not? (Take into consideration the fact that **JOptionPane** is modal.)

Yes, since JOptionPane is modal, the information window will not leave unless acted upon and is always in focus. This makes the program not crash if the JoptionPane produces an answer.

8. Within the `actionPerformed()` method of the **AddEpisodeListener** class, use a **JOptionPane** and create **String**s for all of the information that is to be displayed to the user. These will include the messages to the user and the labels for the buttons. These components should be added to the **JOptionPane**.

9. Complete the `actionPerformed()` method of the **AddEpisodeListener** class. If the user clicks the "Replace Episode" button, this method should save to the model all of the data entered into the **SeriesInputWindow** object, replacing the data that was already there by using a mutator method provided by the data model.

10. Compile the lab assignment with your modified **SeriesController** class. You should not have needed to modify any class other than the **SeriesController** class thus far. At this point you should be able to replace episodes using your code. However, the view will not know that the model has changed, so you may not see the updated information until you take other actions, such as adding an episode with a new name.

11. To correct the lack of updates to the view, look inside the **SeriesModel** class. There you will see some mutator methods from the **Series** class that have been overridden to inform their listeners when mutation events have taken place. You should override any additional mutators necessary to notify potential listeners of any data

changes due to the additional code you added to the **SeriesController** class. When these mutators have been overridden appropriately, all changes to the model should be immediately reflected in the view. Once you can see these immediate changes, move on to the next step of building your menu system.

12. Create a menu item for each of the menu options “Load,” “Save,” and “Exit.” These are to be added to the program under a “File” menu. The type of each menu item should be **JMenuItem**. These objects should be initialized in the constructor of the class. The code for initializing each **JMenuItem** object will be similar to the following:

```
JMenuItem jmiName = new JMenuItem("Name");
```

13. Should the references to these **JMenuItem** objects be stored as class variables or variables local to a specific method? (In answering this question, consider which variables will be referenced in the `actionPerformed()` method.)

Class variables. The variables are needed in many methods.

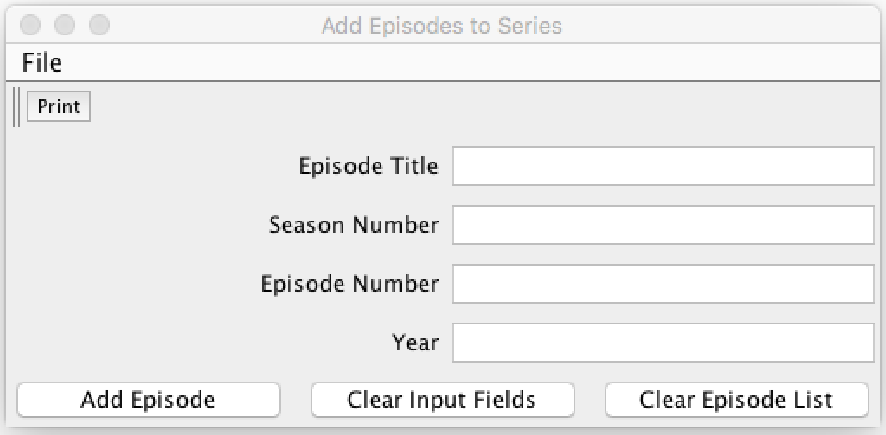
14. Inside the `setInputWindow` method of **SeriesController** you must also register each **ActionListener** on the **JMenuItem** by calling `addActionListener()` on each **JMenuItem** object. The **SeriesController** class should contain the inner classes that implement the **ActionListener** interface and act as the listeners on these menu items.

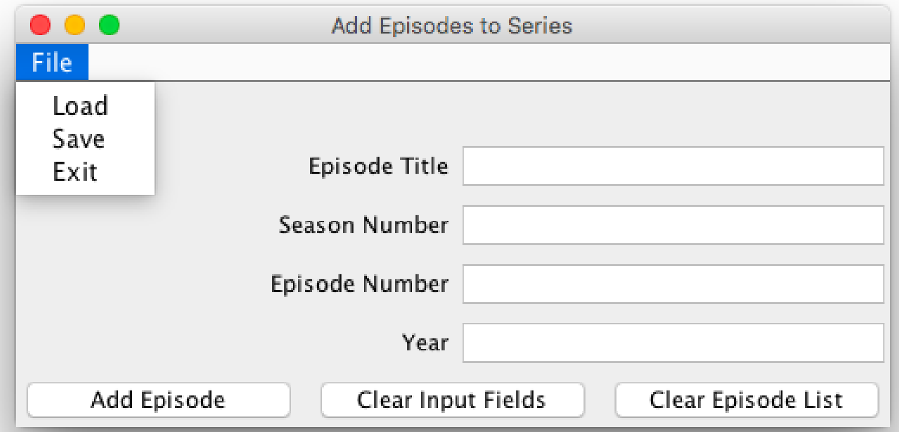
15. Create a **JMenu** object for the “File” menu. Add each menu item to the “File” menu using the `add()` method of **JMenu**. Create a **JMenuBar** object and add the “File” menu using the `add()` method of **JMenuBar**. Add the **JMenuBar** to the **SeriesInputWindow**.

16. At this point, loading the data using the “Load” option from the “File” menu should cause the **SeriesDisplayWindow** to update. This should be apparent if the loaded data is different than whatever is was displayed before the load. If this update doesn’t happen, be sure that in **SeriesModel** you have overridden any mutator methods you added to **Series**.

17. In addition to the menu, we are going to implement a toolbar, using the **JToolBar** class. This toolbar will give us a “Print” option. To create this, create a **JToolBar** by declaring and instantiating it like you would any other GUI component. Then create a new **JButton** labeled “Print” and add it to the toolbar by calling the toolbar’s `add()` method. Last, add the toolbar itself by calling `add()`.

18. All of the menu items and the button in the toolbar should be connected to methods in the **Series** class which conform to their names. You should have added these methods in Step 4, above (see **Series** for details). When everything is completed, it should look approximately as follows (shown with menu closed and open):





19. Ensure that there are no warnings generated for your code. **Do not suppress warnings.** Fix your code so that warnings are not necessary. (If you can't figure out how to fix your code to avoid the cast warning on the cloned `actionListenerList` or the one on `loadEpisodeMap` you may leave in those warnings.)

20. Submit a **completed electronic copy** of these lab instructions along with the **project archive** created following the steps given in the **Submission Instructions** by **11:59 pm on Friday, 24 March 2017** through D2L (`http://learn.ou.edu`). You do not need to turn in a paper copy of your lab.