Comp 401 - Assignment 10: Preconditions, Command Objects and Animation

Date Assigned: Wed Oct 31, 2012

Completion Date: Fri Nov 9, 2012

Early Submission Date: Wed Nov 7, 2012

This assignment will give you practice with assertions, command objects, threads, and animation. You will write preconditions that will be used by ObjectEditor make the avatars and the scene appear and disappear. You will create command classes for the two commands your interpreter processes. You will change your command interpreter to parse each command into a command object before executing it. You will write one or more animating objects that provide animating methods to animate Dorothy, Oz, and Scarecrow. For each of these animating methods you will write animating command objects that call the animating methods with appropriate arguments. Finally you will provide methods in the command interpreter to create and start threads that, in turn, execute or run these animation command objects.

Like previous assignments, the assignment may be changed in minor ways in response to student questions. So please look for tracked changes before you submit.

The following new material is relevant to this assignment.

Documentation: Assertions (10/24, 10/29)	PowerPoint	PDF	Assertions Chapter			lectures.documentation.assertions Package
Animation: MVC (10/29)	PowerPoint	PDF	Commands Chapter	Video		lectures.animation.mvc Package
Animation: Threads and Command Objects (10/29,	PowerPoint	PDF	Commands Chapter	Video		lectures.animation.threads_commands Package

Like the previous assignment, the key is understanding the relevant material. Once you do so, it should be straightforward.

Scene Precondition and Adapting Methods

In your Scene model, create a table that maps each of the four scene object: background, Oz, Scarecrow, and Dorothy, to a Boolean value that determines if the object should be part of the scene or not. For each scene object, write two parameterless scene adapting methods (such as dorothyEnters() and dorothyLeaves()) that set the Boolean value to true and false, respectively. Using OE conventions, write a precondition method for the getter of each of the four Scene objects that returns true only if the corresponding object is part of the scene. The getter method should invoke the corresponding precondition method in an assertion. However, you do not have to turn assertion checking on to test this part of the assignment.

If you are not doing extra credit, your Scene object should not have an addPropertyaddPropertyChangeListener(PropertyChangeListener aListener) method in it. As mentioned earlier, only atomic objects should notify; so this will not cause your previous work to fail. The reason for this constraint is that when a scene adapting method changes the status of an object, no property is fired. At this point we want ObjectEditor to automatically refresh the scene object. However, if the scene object is an observable, ObjectEditor does not do an automatic refresh. So do not make the object an observable.

If your Scene class is a subclass of some class that is makes it an observable, then simply do a manual refresh from the user interface to see the effect of the preconditions; the TAs will do the same. Alternatively, download the new version of oeall21 I uploaded at ~~6:45~~12:09pm pm on 11/2~~1~~. Make your scene class a PropertyListenerRegistrar, and in each scene adapting method, fire the change : propertyChangeSupport.firePropertyChange(new PropertyChangeEvent (this, "this", null, null)). OE will automatically refresh the object when it receives this notification.

Command Objects and New Interpreter

The setter of the editable property of your command interpreter processes two commands: the move command and the say command. Hopefully, the setter does not does not process them directly and calls different methods for processing the two commands. If you have written a monolithic setter, change it so that there is a separate procedure for parsing and processing each command. Let us call these procedures the say and move parsing methods, respectively.

Now convert each of these two methods to a function that returns a command object as a value. The two methods will no longer process the command. Instead, they will construct and return command objects. The caller of each of these methods (which can be the setter or some method called by the setter) will execute the returned command object to process the command. Thus parsing and processing of commands is done in two different methods. The two

functions will return instances of two different command classes. Let us call them the say and move command classes, respectively.

Each of these classes implements the Runnable interface. This means they implement the run() method defined by this interface. They will define constructors that take parameters that specify the associated user command. These parameters are not tokens but instead primitive and object values denoted by these tokens. For instance, the move command class will define a constructor that takes a parameter that specifies the avatar object and two int parameters that specify by how much the avatar should be moved in the X and Y directions. (As the constructor does not take tokens as parameters, it can be used in contexts other than parsing. In addition, each time its run() method is called, no conversion of tokens to Java objects/primitive values is done.)

As mentioned above, the run method will be invoked by the caller of the two parsing methods to process the associated command.

You can change the interpreter directly. You do not have to subclass the existing interpreter.

Animation and Threads

In this part, you will add three methods to the interpreter to animate Dorothy, Oz, and the Scarecrow in separate threads. The exact animation does not matter. Let us call these methods the asynchronous animation methods, respectively.

You should follow the animation pattern given in class. This means you must create one or more animating objects and command objects. For regular credit, you can write a single animating object with a single (synchronous) animation method that takes as parameters the avatar to be animated and optional animation controls such as the sleep delay. Also for regular credit you can define a single animation command object for calling this (synchronous) animation method on this avatar with the animation avatars. Your asynchronous animation method in the interpreter will create and start a thread with an appropriately constructed instance of the command object.

Extra Credit: Precondition-checking Manually Implemented Oz View

Change the Oz view(s) you wrote in the previous assignment to check the precondition of the getter of each of the four scene objects before deciding if the object should be displayed.

If no property change is fired by a scene adapting method, then if you invoke the method in the OE view, your own view gets no notification. To overcome this problem I have associated new semantics with property changes. If your object sends a notification with the property name "this" (the string "this", not the object this), OE refreshes the object. This property can be received also by your view. You are free to encode in the old and new values additional information that your view can use to efficiently hide and show specific properties. For instance the property name can be in the old value and the visibility status of the property in the old value.

To get the benefit of these semantics, you will have to download the new version of oeall21 I uploaded at ~~6:45~~12:09pm pm on 11/~~21~~. Next make your scene Object a PropertyListenerRegistrar, and in each scene adapting method, fire a change of the form: propertyChangeSupport.firePropertyChange(new PropertyChangeEvent(this, "this", “Dorothy”, true));

Extra Credit: Preconditions of Scene Adapting Methods

A scene adapting method will not always be invokable. For example the dorothyEnters() method should be invoked only if Dorothy is not part of the scene. Write a precondition method for each of the scene adapting methods that determines if the scene adapting method is invokable.

Extra Credit: Dynamically Enabled Buttons/Menu Items

Create a controller for the OZ scene model that adds buttons or menu items to the OZ scene frame for invoking the scene adapting methods (e.g. dorothyEnters()).

Make the controller check the preconditions for these methods to decide if the corresponding buttons/menu items should enabled. You can use the setEnabled(boolean) method of a button/menu item to enable or disable it.

After a button or menu item is activated, check the precondition methods of the model to determine the enabled status of all buttons and menu items. Alternatively, make your scene class a PropertryListenerRegistrar, fire property change events in your scene adapting methods as described above, and make your controller a listener of these events. This will allow your buttons/ menu items to have the same enabled state as the corresponding ObjectEditor menu items.

Extra Credit: Move and Rotate Postcondition Assertions

In the method for moving an avatar, write post condition assertions that say that the relative distances among the avatar parts do not change after the method is executed, and none of the components of the avatar rotates. Similarly, write post condition assertions in the method for rotating the avatar that say that none of the components of the avatar moves.

Create an alternative method for moving/rotating an avatar that also asserts the same post condition as the correction one but deliberately fails to show that the assertions work.

Extra Credit: New Command Interpreter Buttons

Add a controller to your command interpreter that provides buttons to invoke the asynchronous animation methods.

Extra Credit: Multiple Animation Methods

Write three different animation methods for Oz, Dorothy, and Scarecrow, respectively. The Oz animation method should make (the Wizard of) Oz clap on each beat (sleep call) and the

Scarecrow and Dorothy animation methods should make the corresponding avatars do two different dances to the same song. That means, on each beat they should say the same thing, but do different movements. To invoke these methods in separate threads, you will have to write three different command ~~objects~~classes, one for each method.

Extra Credit: Command List

Modify your interpreter to add all executed commands to a command list it receives from the main class. This means that the method(s) that execute a command object should also add the object to this list. Follow OE conventions to create the list. If you do so, the main method can display the list in an OE window. You should be able to right click on each command display to pop up a run command menu item, which you can select to re-execute the command.

A command object will not have any properties. ObjectEditor displays an object with no properties as the value returned by its toString() method. You should override this method to give details you think are appropriate to show for a command. If the toString() method is not called by OE, associate its class with the annotation DisplayToString(true).

Extra Credit: Observable Command List

Follow the OE collection conventions to make the command list an observable.

Main Class

For regular credit, write a main class that creates and visualizes an instance of the Oz scene and the command interpreter using OE.

Depending on which extra credit features you have done, you may also have to call the erroneous move and rotate methods to violate assertions, create and display the command list, and display the manually implemented interpreter and Oz scene.

As in the last assignment, the TAs will interact with the controllers manually – your main method should call only model methods.

As in the last assignment, do not call the OEFrame refresh method.