Comp 401 - Assignment 10: Preconditions, Commands, Threads, Animation

Date Assigned: Thu Oct 29, 2015

Completion Date: Fri Nov 6, 2015

Early Submission Date: Wed Nov 4, 2015

This assignment will give you practice with assertions, command objects, threads, and animation. You will write preconditions for certain methods of your scene object. 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 avatars. 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 the previous assignment, the key is understanding the relevant material. Once you do so, it should be straightforward. If you make a mistake, it will be difficult to debug, so program carefully.

Documentation: Assertions	PowerPoint PDF YouTube	Assertions Chapter		Assertions, Commands, Threads	lectures.documentation.assertions Package
Animation: Threads and Command Objects	PowerPoint PDF YouTube	Commands Chapter	Visualization		lectures.animation.threads_commands Packag

Preconditions and Extended Console View

Define and assert preconditions for the public methods defined by the simulation to speak, pass, approach, and fail. Let us call these methods as asserting methods. You should implement the preconditions in separate public methods that follow the ObjectEditor conventions we saw in class for these methods. Each of the asserting 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. Instead do the following to test them.

Make your scene class an implementation of PropertyListenerRegistrar (so far you were required to implement this interface only in atomic shapes). After any code in the class that changes the (value returned by the) precondition (method) of some asserting method M (from false to true or versa) fire the following precondition change event:

new PropertyChangeEvent (this, "this", MTag, newPreconditionValue)).

For example, suppose you execute some code that changes the (value returned by the) precondition (method) of the approach method from true to false. Right after that code, you should notify all observers of the scene class of the following event:

new PropertyChangeEvent (this, "this", "approach", false))

OE will automatically refresh the menu of the object when it receives this notification and grey it if the precondition is false.

If some code causes changes to return values of multiple precondition methods, then fire separate events for all of them.

Make your console view class of assignment 8 register itself as an observer of not only the atomic scene shapes as before but also an observer of the scene object. Thus, it will now print the precondition change events above so that the graders can see that your preconditions are changing correctly.

Command Objects for Parsing

Create two new command classes, tagged "SayCommand" and "MoveCommand," respectively that represent the two commands your parser understands currently.

Each of these classes implements the Java 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 objects and primitive values. 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. Thus, this command object uses the result of parsing. The order of the constructor arguments should follow the order of tokens of the associated command. Thus, the object on which the command (e.g. Avatar) should be the first argument and the parameters of the command should follow (e.g. xoffset and yoffset) should appear in that order. Similarly, the say command will take the scene object as the first argument and a String value.

Deleted: primitive and object values denoted by these tokens

The run method of a command class will execute the appropriate method on the object passed as the first argument of the constructor, passing to the method the values passed as other parameters in the constructors. For example, the run method of the move command will move the avatar based on the two int constructor parameters.

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.

The command objects you define here will not be used for animation, at least in this assignment. They will be used only for parsing, as indicated below.

Command Interpreter with Command Objects

The setter of the editable property of your current command interpreter processes two commands: the move command and the say command. The setter should call different methods for processing the two commands. *Tag these methods as "parseSay" and "parseMove" methods, respectively.*

Now make each of these methods not only parse but also create and return command objects. This means you must now convert each of these two methods to a function that constructs and returns the appropriate command object as a value. The two methods will no longer process the command directly. The caller of each of these methods (which can be the setter or some method called by the setter) will execute the run method of the returned command object to process the command. Thus parsing and processing of commands is done in two different methods. You decide what the arguments of these methods are.

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

Command Interpreter with Asynchronous Animation Methods

In this part, you will add four void parameterless methods to the command interpreter to animate Arthur, Galahad, Lancelot, and Robin in separate threads. The exact animation does not matter. For example, you can make them walk, as I did in my demo. Let us call these methods the asynchronous animation methods, respectively. Tag them as "asynchronousArthur," "asynchronousGalahad," "asynchronousLancelot" and "asynchronousRobin" respectively.

You should follow the animation pattern given in class. This means you must create one or more animating objects and command objects. Tag the classes of these objects as "Animator" and "AnimatingCommand," respectively. The command objects you create here are distinct from the ones you created above for parsing.

All command objects you create for this part of the assignment will be an instances of the same class. Similarly all animation objects you create for this part of the assignment will be instances of a single of animation class. This means, you must write a single class for animating all avatars.

Deleted: s

Deleted: these

Deleted: es

Deleted: associated

Deleted: scene

Deleted: using the

Deleted: W

Deleted: objects (for all

Deleted:)

with an animation method that takes as a parameter an Avatar. Tag this method as "animateAvatar". You can define additional animation methods that take optional animation controls such as the sleep delay. Similarly, you should define a single class for animation command objects, which calls the required animation method (animateAvatar). The class will of course be instantiated multiple times with different constructor parameters, each time an asynchronous animation method is called. Thus, your asynchronous animation methods in the interpreter will create and start threads with appropriately instances of these command objects.

Deleted: Y

Deleted: A

Deleted: Y

Extra Credit: Dynamically Enabled Buttons for Calling Asserting Methods

Add JButton buttons to your command interpreter controller for calling the pass, say, fail, and approach methods on the simulation respectively. The approach button can make the some avatar chosen by you to approach. Similarly, the say button can cause some predefined string to be said. Make the controller check the preconditions for these four asserting methods to decide if the corresponding buttons should be enabled. For instance, make it disable the say button if the precondition of the say method is not true. This means that the controller must listen to model notifications– specifically the precondition notifications.

Define a readonly property for each button that returns the button you created. Thus, you must have properties named "Say", "Pass", "Fail", "Approach" of type JButton.

You can use the setEnabled(boolean) method of a button to enable or disable it.

Extra Credit: Clapping Guard Animation

Add another asynchronous animating method to the command interpreter that makes the guard clap in a separate thread. Tag this method as "AsynchronousGuard". We will be using this method in the next assignment to synchronize the other threads. See my demo to understand how this might work and be used.

Animating Demoing Main Class

To demonstrate your work, write a main class that creates a scene object displays animations of it using the console scene view and the painting view, and displays the command interpreter user interface. Specifically, the main class:

1. Instantiates a scene object.
2. Displays it using your painting view object
3. "Displays" it using the extended console view.
4. Instantiates a command interpreter object.
5. In case of extra credit, displays the custom user interface (possibly extended with three new extra credit buttons) for the command interpreter.
6. Executes animating code to change the preconditions of each of the four asserting methods. For example it executes the approach command to change the (value returned by the) precondition of the say method, and the say method to change the

precondition of the pass method. The animating code should allow the TAs to observe the effect of the code on both the scene views and see that that the correct precondition events are fired.

7. Executes the four asynchronous animation commands in the command interpreter model.

Constraints

1. You should use animation pattern, as mentioned before
2. As before, there should be no warnings from ObjectEditor – if there are spurious warnings, let me know.

Be sure to follow the constraints of the previous assignments.

Submission Instructions

- These are the same is in the previous assignment. The TAs will run the main method to see the test cases animate.
- Be sure to follow the conventions for the name and package of the main class.

Good luck!