SLogo Design Document - Team 11

Design Goals

Front End:

- The View module is the GUI for the program and is the overarching class for the front-end. It contains the TurtleWindow and the CommandControl, which each serve their own purpose as modules of their own. The View will also have interactive configuration options for the program. All of the GUI components are part of a BorderPane, which is added to a Scene (which is in turn part of a Stage at the top level). The View returns the Scene (and all of its sub-components) so that it can be added to the Stage in Main.
- The TurtleWindow module is the visual space within which the Turtle moves. It stores and updates the Turtle object, its position, and the lines drawn on the screen based on commands inputted by the user, which are interpreted by the back end of the program.
- The CommandControl module serves as the user's interface for creating a command (or series of commands). It contains all of the Buttons (except for "Enter") and the TextField command line. It stores the user's current command in the TextField, and allows for the command to be accessed by the back end as a String so that it can be interpreted.

Back End:

- Input - The input class receives the user's input as a string, parses it. This is then passes to the Interpreter class.
- The Interpreter class - Receives the Input object, and using the instructions constructs a list of 'operations' for the turtle (location, heading, etc.)
- The Output class receives the list of operations from Interpreter and constructs a list (probably a Queue) of states for our turtle. This will probably be in the from of a que of lists containing location, heading, etc.
- The Model class is the superclass to all of these three classes. This receives the string as input, and creates the input class with the string. Then it uses the interpreter class to construct the list of operations, finally using the Output class to give the Front End package back a list of upcoming turtle states to animate. The reason the Back End is organized this way is to make it easy for the front end to pass the commands and get new states for the turtle. With this design the front end should be able to call one update() method of the Model class and get the output it needs to animate.
-

Primary Classes and Methods

Main

- Intializes both the Model and the View, sets up the Stage using the View (which in turn uses the Model)

Front-End

- View
    - Initializes and stores the TurtleWindow
    - Initializes a VBox to store the buttons (uses CommandControl to get the List of Buttons to be added)
    - Initializes an HBox to store the command line TextField (editable) and the Enter button
    - Initializes TextAreas to store previous commands entered and commands that are available
    - Includes several components for interactive configuration options - (CheckBoxes, ComboBoxes, etc.)

    - Enter button - when pressed, calls getCommand() from the CommandControl and then passes the command to the back end's update method
    - *public Scene getScene()* - returns the Scene to the Stage to set up the application
- CommandControl
    - Stores a TextField (the command line) that can be edited by the user
    - Initializes all Buttons and their handlers - when pressed, each Button updates the TextField's string to reflect the command that is chosen (add it to the command chain) using a private updateTextField(String) method
    - Stores the Buttons in a List that can be accessed using getButtons()
    - *public String getCommand()* - returns the String value stored in the TextField (accessed by View when Enter is pressed)
    - *public TextField getTextField()* - returns the TextField so it can be added to the GUI in View
    - *public List<Button> getButtons()* - returns the List of Buttons so they can be added to the GUI in View
    - *public void clearCommandLine()* - adds the current String to the TextArea containing previous commands, then clears the command line to the empty String
- TurtleWindow
    - Instantiates and stores myTurtle
    - Displays the Turtle's current position and orientation in TextFields
    - Stores color of line to be drawn
    - initialize - creates turtle and sets its image/color
    - *public void update(List<Point> list)*
        - Takes a List of points as a parameter, which represent the endpoints for each straight-line movement that the Turtle makes
        - Creates a PolyLine using these points and adds it to the TurtleWindow Pane so that the lines are added to the GUI (using makePolyLine(list))
        - Sets the current myTurtle position to the last point in the Collection (where it stops)
        - Sets the orientation of myTurtle based on the last two points
        - Updates the TextFields that display the Turtle's information
- Turtle
    - Stores a current position, orientation, and image
    - *public void setPosition(Point point)* - sets the position of the Turtle
    - *public void setOrientation(List<Point> newList)* - sets the orientation of the Turtle based on the direction of the vector between the last two points in the List
    - *public void setImageView(ImageView i)* - sets the Turtle's image representation

Back-End

- Input
    - takes in information in the form of a string in its constructor, par and passes it to the interpreter class
    - uses the parse(myString) method to separate the string input into relevant command chunks (sum, forward, integer values, etc)
    - operations(myCommands) will take the output of the parse(myString) method and passes the list of commands it constructs to the interpreter
- Interpreter
    - The interpreter takes the list of commands from the Output class in it's constructor

    - The turtleOperations(myCommands) method uses the list of commands to be performed to create a list of operations to be performed on the turtle
    - nextStates(myOperations) takes the list of operations and constructs a list (probably a Queue…) of states to pass to output. States for the turtle will consist of coordinates, heading, etc.
- Output -
    - The Output class takes the list of nextStates from the nextStates(myOperations) method sends back to the view
- Model
    - manages all of these actions by Input, Interpreter, and Output in an update() method.
    - Model.update() is what the view will call to have the back end evaluate the user input and give it back the information it needs to animate the turtle.

The goal was to have this all be handled ultimately in one superclass (Model) and have commands be dealt with using one function, the Model.update() function. The point of this is to make it easier and simpler for the View to communicate with the backend.

Setting up all of the GUI components so they work together is a little complicated. Main will initialize the Stage, Model, and View, and the Stage will receive a Scene from View.getScene(). The View will add a BorderPane to this Scene, and the BorderPane will serve as the layout for our user interface. The left space of the BorderPane will be occupied by our TurtleWindow, which extends Pane. The right space of the BorderPane will be a VBox that adds the list of Buttons using CommandControl.getButtons(). The bottom space of the BorderPane will be an HBox that adds the command line using CommandControl.getTextField(), and the Enter button will be added to this HBox as well.

Example Code

Sample Sequence of Code:

Front End

- User types 'fd 50' in TextField, presses Enter button
- Enter button's handle method then calls:

```
List<Point> newList = myModel.update(myCommandControl.getCommand());
```

Back End

- Public List<List> update() {
    - Input input = new Input(myInput);
    - inputCommands = input.parse();
    - Interpreter interpretation = new Interpreter(inputCommands);
    - commandList = interpretation.turtleOperations(myCommands);
    - nextStatesList = interpretation.nextStates(commandList);
    - Output output = new Output(nextStatesList);
    - return output;

}

Front End

- myTurtleWindow.update(newList);
    - myTurtle.setPosition(newList.get(newList.size()-1));
    - myTurtle.setOrientation(newList);
    - this.getChildren().add(makePolyLine(newList));
- myCommandControl.clearCommandLine();

JUnit Tests
Note that these tests use getter and setter methods that may not have been specified yet above, because it is hard to replicate testing the interaction between classes without using the full program implementation

```
@Test
public void testTurtleUpdate(){
        TurtleWindow myTurtleWindow = new TurtleWindow();
        myTurtleWindow.getTurtle().setPosition(new Point(0,0));
        List<Point> newList = new ArrayList<Point>();
        newList.add(new Point(2,3));
        myTurtleWindow.update(newList);
        assertEquals(myTurtleWindow().getTurtle().getPosition().getX() == 2);
        assertEquals(myTurtleWindow().getTurtle().getPosition().getY() == 3);
}

@Test
public void testInitializeMethods(){
        View v = new View();
        assertNotNull(v.getCommandControl());     //assert that CommandControl was initialized
        assertNotNull(v.getCommandControl().getTextField());   //assert that TextField was
initialized
        assertNotNull(v.getTurtleWindow());       //assert that the TurtleWindow was initialized
        assertNotNull(v.getTurtleWindow().getTurtle());     //assert that the Turtle was initialized
}

@Test
public void testTextFieldClear(){
        CommandControl myControl = new CommandControl();
        myControl.updateTextField("sample command");
        assertEquals(myControl.getCommand(), "sample command");
        myControl.clearCommandLine();
        assertEquals(myControl.getCommand(), "");

@Test
public void testModelUpdate() {
        Model model = new Model();
        model.myInput = "Sample command";
        goodNextStates = //expectedValue
        assertNotNull(model.update());
}

@Test
public void testInputParse() {
        Input input = new Input("sample string");
        assertNotNull(input.parse());
        assertEquals(input.parse(), [ideal parsed string in list form]);
}
```

Alternate Designs

Describe at least one alternative to your design that the team discussed and explain why the team choose the one it did.

We considered designing the GUI so that we had a single Grid class that stored the turtle and all the visuals and updated when necessary. However, we decided that it was not a good design to have a single GUI class that had access to everything and did all of the work on its own. We also considered having a class for every different type of button that could be clicked, but we decided not to do it this way because of the relationship between front-end and back-end. We decided that the buttons should simply serve to manipulate the command that is passed to the back-end, rather than each calling specific methods within the back end. We let the back-end handle the interpreting to determine what needs to be updated in the display, rather than trying to perform more specific individual updates as part of the GUI. To minimize the sharing of information between the View and the Model (and therefore minimize the modifications necessary for extension), we decided to keep the command hidden from the Model until we are sure that it is what the user wants to execute, and then we pass it in a single method call.

Roles

List each team member's role in the project and a description of what they expect to work on.

Ashwin - backend
Greg - frontend
Justin - backend
Rica - frontend