Design Goals

The idea behind our SLogo implementation is to have a program with a clean and intuitive interface, a flexible structure of the Logo language, and extensibility in user-set parameters. Our project is divided into front-end and back-end, where front-end deals with the user interface while back-end deals with Logo language hierarchy and logic. The front-end can be seen as the GUI, which contains the scene for drawing, the text window for user commands, history displays, and any other user features that may be extended. The back-end contains the bulk of the program logic, with the Logo instruction hierarchy, the instruction parser, class design factories, and the turtle-pen needed to draw on the scene.

The idea for the program is to keep the interface between GUI and back-end model very simple. The GUI communicates with the model almost solely through the parser, which connects both ends by processing the string input from the command window. The only other model structure the GUI has access to is the turtle-pen, whose color and other line properties are set directly from the GUI. The back-end then parses through the command string, determining what commands need to be executed as well as what variables need to be updated. The variables, such as the list of user-defined functions and the turtle image, should be objects with pointers from both the model and GUI, and hence when the variables are updated by the model, the GUI should also be able to update its view. The parser will also output a final value from the command, usually the return value of the last function call, and display it to the user.

In keeping the interface between GUI and model simple, we are hoping that extensions in the GUI will have little impact on the hierarchy of the back-end classes. Also, this leaves a large amount of leeway for back-end design, in case the current implementation plan does not work out. As for error parsing, when an error occurs in either the GUI or the model, either should be able to throw an exception or an error dialog box from a separate class of SLogo errors. SLogo errors may be a superclass which can then be extended by sub error classes specific to certain modules (i.e. parsing).

Primary Classes and Methods

Main.java

- responsible for executing the program
- SLogoView()

Model Package

- Model.java
    - takes in a string of commands from the console (passed by the front-end) and executes it
    - Instance variable: (Parser) parser
    - Methods:
        - executeCommand()
            - calls the methods in the parser

- Turtle.java

    - Instance variables: (ImageView) image, (double, double) currentCoord, (double, double) destinationCoord, (boolean) isHidden, (Vector) angle, (Node) group containing lines from GUI
    - Methods:
        - update()
        - move(), turn(), etc
        - drawLine()

- Pen.java
    - Instance variables: (Color) lineColor, (list) lines, (boolean) isPenDown,
    - Method:
        - setPenColor(), getPenColor()
        - addLineToList()
        - isPenDown()

Parser Package

- Parser.java
    - uses Tokenizer.java to split string into tokens, and turns them into usable commands
    - Instance variables: (List) commandList
    - Method:
        - convertToken()
            - checkException()
- Tokenizer.java
    - takes in a string of commands from the console from the model.java of the model package, and converts it into tokens → constant, variable, command, list
    - Instance variables: (String) regex (for each token type), (List) tokenList
    - Method:
        - createTokenList()
- UserDefinedCommand.java
    - used to parse user defined commands
- Factory Package
    - to control the nodes of the parsing tree and choose commands for the nodes
- Command.java (interface) //may change it into a package
    - TurtleCommand.java (interface)
        - classes to implement commands that affect the turtle inherit TurtleCommand.java
- OperationCommand.java (interface)
    - Math.java (interface)
        - classes to implement mathematical functions inherit Math.java
    - Boolean.java (interface)
        - classes to implement boolean commands inherit Boolean.java

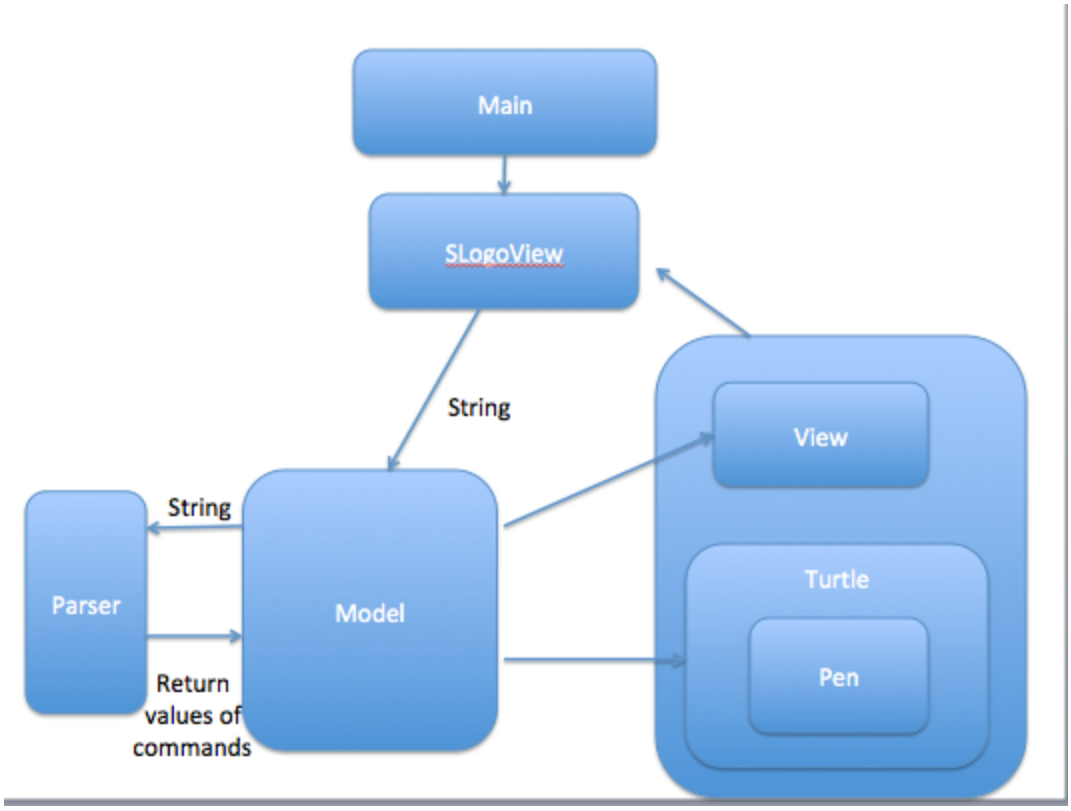
View Package

SLogoView.class

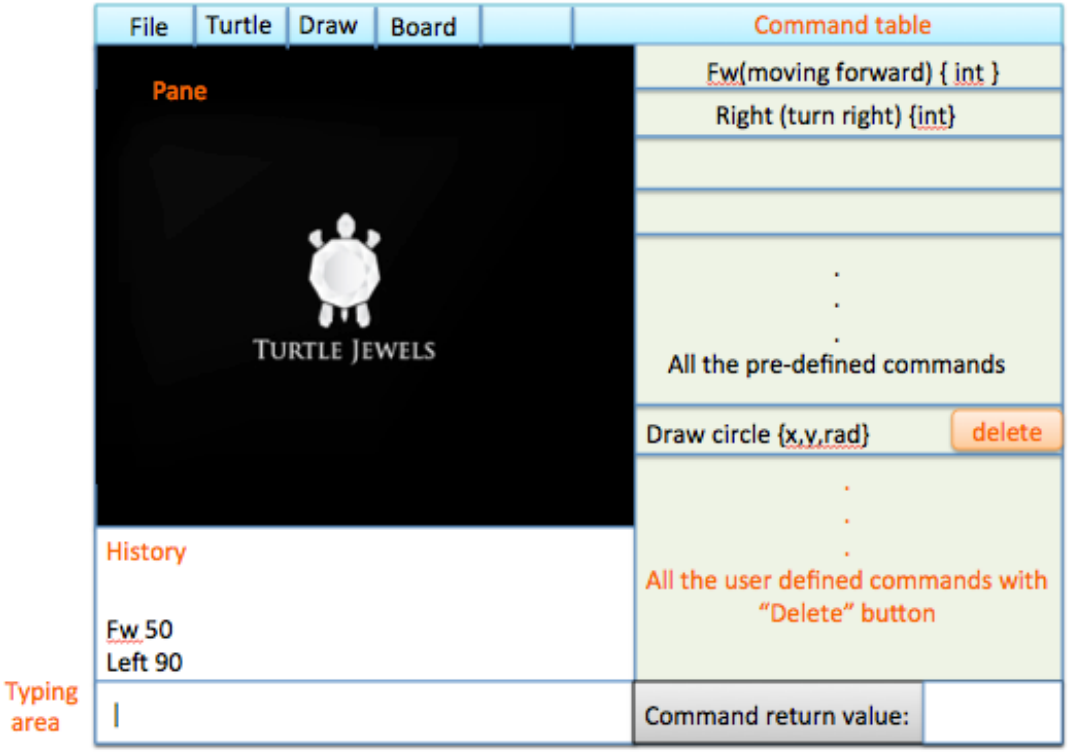
```
public static final Dimension DEFAULT_SIZE = new Dimension(800, 600);
private scene myScene;
private SLogoModel myModel;
private ResourceBundle myResources;
private Color myPenColor;
private Turtle myTurtle;
private ImageView myTurtleImage;
private Group myLines;
private Line myCurrentLine;
private List<CustomCommands> myCustomCommands;
private int myCustomSize;
private Parser myParser;
private Model myModel;
private boolean myButtonState;
private Group myHistory;

public class SlogoView {

// Initializes the model to be used and the language to run the program in.
public SlogoViewer(SLogoModel model, String language){
            myResources = ResourceBundle.getBundle("resources/"+language);
      BorderPane root = new BorderPane();
// Setting the visual nodes of the program in the appropriate places.
      root.setCenter(makeDisplayPanel());
      root.setTop((makeSettingsPanel()));
      root.setBottom(makeHistoryAndInputPanel());
      root.setRight(makeCommandPanel());
// Instantiate turtle, model and scene.
      myTurtle = new Turtle(myTurtleImage, myPenColor, myLines);
      mySlogoModel = new SlogoModel(myTurtle);
      myScene = new Scene(root, DEFAULT_SIZE.width, DEFAULT_SIZE.height);
// Instantiate parser
myParser = new Parser();

// Instantiate a model
myModel = new Model (myTurtle, myCustomCommands);
}
// Returns the scene for use in Main.
public scene getScene();
```

```
// Generates the main display that contains the following nodes - myTurtleImage, myLines, and myBackground. The main display will be overlayed with a GridPane in order to allow the showing of grid lines. Displays myReturn value in a Text node on the display.
private Node makeDisplayPanel();
// Populates the list of possible commands that can be used. Ideally includes dialog box that shows parameters for each command. Adds buttons and also CustomCommand nodes that can be deleted. Each button has an event handler that writes out the function into the InputPanel.
private Node makeCommandPanel();
// Creates CustomCommand that includes a delete button. Can be added to Command Panel.
private Node makeCustomCommand();
// Creates Settings Panel that allows for the changing of myPenColor, myTurtleImage, myBackground using buttons and drop down menus
private Node makeSettingsPanel();
// Calls makeHistoryPanel and InputPanel to generate both.
private Node makeHistoryandInputPanel
// Scrolling list of Commands that have been used previously. These commands will be buttons that when allow the command to be run again. They are added to the group myHistory and displayed in the Panel.
private Node makeHistoryPanel();
// Creates a TextField that can be take in commands as well as a submit button that sends the text as a String into the model. When submit is clicked, a button should also be generated that is added to Histo
private Node makesInputPanel();
// Method that generates buttons for use in the other panels.
private Button makeButton(String name, EventHandler<ActionEvent> handler);
//Deactivates Buttons based on myButtonState
private void changeButtonState(Button button);
//Updates the scene by adding new commands to the Command Panel by calling makeCustomCommand. Used as an action event for every timeline frame.
private void checkandUpdateCommands();
```

UML diagram:



Interface design:



Alternate Designs

1. Make a pane with every grid as a pixel. Pros: Same implementation as in CellSociety. Cons:Each grid has to be very small and the implementation is a waste of memory.
2. When drawing the line, get the starting point and the ending point and let the turtle jump from the original location to the target location. Pros: Faster to process. Cons: Unable to see the process of drawing and no way to change the speed of drawing.
3. The front-end gives list of string the user types into the parser and executes the commands at the back-end. Then the back-end returns a list of commands to the front-end and the front end will execute all the commands to update turtle or add in user-defined commands. Pros: The turtle class will be totally at the front-end and will not involve in command execution at the back-end. Cons: The line between front-end and back-end gets vague as part of command execution which related to the turtle is at the front-end. It also makes the parser hard to decide when to return the command. As we decide command like "forward 50" returns a value 50 and "forward forward 50" equals "forward 100", it is hard for the back-end to process the commands with several layers and decide if the commands have been simplified to those only related to the action of turtle.
4. We used to decide to put the error checking of front-end and back-end into the same class and show the error through the GUI. Pros: Group classes of same function together to enhance the readability of the codes. Cons: The front-end and back-end have to share the access of GUI, increasing the visibility and influence between two ends. Solution: Use boundary limit or ask the user to choose from a selection instead of typing in information to eliminate the error in GUI. The

back-end will throw the error from the pop-up dialog window to avoid accessing and changing elements in GUI.

Example Code

- *The user types 'fd 50' in the command window, and sees the turtle move in the display window leaving a trail.*
    - 1) The string ‘fd 50’, which is typed into the console, is passed by the SLogoView class (front-end) to the model class of the Model package (back-end).
    - 2) The model class executes ‘fd 50’, using its executeCommand() method. The model class has the parser as its instance variable; the executeCommand() method calls the methods in the Parser class of the parser package.
    - 3) The Parser class uses the Tokenizer class, which is has the method createTokenList() responsible for converting such string of commands from the console into tokens.
    - 4) The Parser class takes such list of tokens from the Tokenizer class and converts them into usable commands → use of parsing tree
    - 5) Model class finally executes commands and returns the results to the SLogoView to be displayed.

JUnit Tests

```
public class ForwardTester {
        public void executeTest() {
                myTurtleImage = new ImageView();
                myLines = new Group();
                ForwardInstruction tester = new ForwardInstruction(50);
                Turtle turtle = new Turtle(myTurtleImage, Color.BLACK, myLines);
                tester.execute(turtle);

                //Test to see if the turtle's coordinates changed appropriately.
                assertEquals("The x coordinate should be 50",  50, turtle.getXCoord());
                assertEquals("The y coordinates should still be 0", 0, turtle.getYCoord());
                //Test to see if line was generated
                assertEquals("myLines should have one children", 1, myLines.getChildren().size());
                //Test to see if the line is the correct.
                assertEquals("The line generated should have the right start x coord", 0,
myLines.getChildren(1).startX);
                assertEquals("The line generated should have the right end x coord", 0,
myLines.getChildren(1).endX);
                assertEquals("The line generated should have the right start y coord", 0,
myLines.getChildren(1).startY);
                assertEquals("The line generated should have the right start y coord", 50,
myLines.getChildren(1).endY);

        }
```

```
}
```

```
	public void testTurtleCommands() {
		Instruction forward50 = new Forward(50);
		Instruction back39 = new Back(39);
		Instruction left21 = new Left(21);
		Instruction right55 = new Right(55);
		Instruction setHead24 = new SetHeading(24);
		Instruction penUp = new PenUp();
		Instruction forward100 = new Forward(Forward(50));

		assertEquals(50, forward50.evaluate());
		assertEquals(39, back39.evaluate());
		assertEquals(21, left21.evaluate());
		assertEquals(55, right55.evaluate());
		assertEquals(24, setHead24.evaluate());
		assertEquals(0, penUp.evaluate());
		// Turtle moves 100 but method still returns 50
		assertEquals(50, forward100.evaluate());
	}
```

```
	public void testMathOperations() {
		Instruction sum = new Sum(2,50);
		Instruction diff = new Difference(4,1);
		Instruction remain = new Remainder(20,6);
		Instruction minus = new Minus(4);
		Instruction sin = new Sine(43);
		Instruction power = new Power(4,3);

		assertEquals(2 + 50, sum.evaluate());
		assertEquals(4 - 1, diff.evaluate());
		assertEquals(20 % 6, remain.evaluate());
		assertEquals(-4, minus.evaluate());
		assertEquals(Math.sin(43), sin.evaluate());
		assertEquals(Math.pow(4, 3), power.evaluate());
	}
```

Roles

Kevin/Meng'en - Working on the Front End (SLogoViewer). Will move onto backend parsing and model class as soon when finished.
Jennie/Sandy - Working on backend code including the parser class, the model class as well as Instruction Hierarchy.