SLogo Design Document - Team 11

Describe the program's API and justify its design:

- between your sub-groups: how and what they will communicate (and thus what will remain hidden, just part of the implementation) . Focus especially on how you plan to separate the graphical interface from the interpreter and how you plan to let them communicate when necessary (especially parsing errors).
- between your sub-group and future programmers: how to extend your part to add new features (and thus what kind of code they will be expected to write and what parts of your code you expect to be closed to modification). Focus on providing paths for extension through interfaces, inheritance, and design patterns for new functionality you might reasonably expect to be added to the program.

Note, the API agreed to between the sub-groups should ***not*** be changed after this document because it affects how the other sub-group's part might work. If it must be changed, the change and its reasons must be clearly described in a separate document, API_CHANGES, that is to be included in the project's GitHub repository.

Design Goals (high-level description of the project's "vision")
Describe the goals of your project's design (i.e., what you want to make flexible and what assumptions you may need to make) by breaking it into *modules*. A module is a concept in your program that may be represented by a single class or a group of classes related in some standard way, such as set of subclasses, that has a purpose with regards to the program's functionality or collaborates with other modules.

Front End:

- The View module is the GUI for the program and is the overarching class for the front-end. It contains the TurtleWindow, CommandControl, and CommandLine, 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 TurtleImage 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 Command buttons (except for "Enter").
- The CommandLine module is a TextArea command line. It stores the user's current command in the TextArea, 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 (more detail with actual complete method signatures)
Describe the program's core architecture (focus on behavior not state), including pictures of a UML diagram to describe the Model and "screen shots" of your intended View interface

Main

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

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.

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 TextArea (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
    - Initializes all Buttons and their handlers - when pressed, each Button updates the TextArea's string to reflect the command that is chosen (add it to the command chain) using a private updateTextArea(String) method
    - Stores the Buttons in a List that can be accessed using getButtons()
    - *public List<Button> getButtons()* - returns the List of Buttons so they can be added to the GUI in View
- CommandLine
    - The command line is a TextArea that can be edited by the user
    - Able to use all of TextArea's public methods (appendText, setText, getText)
    - Separate class so we can extend for more functionality when needed
- CommandFactory
    - *public makeCommand(String s)* - Takes a String describing the desired type of command and returns a new instance of that command (a new subclass of SuperCommand)
- TurtleWindow
    - Instantiates and stores myTurtle
    - Displays the Turtle's current position and orientation in TextAreas
    - Stores color of line to be drawn
    - initialize - creates turtle and sets its image/color
    - *public void update(List<Point2D> 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 TextAreas that display the Turtle's information
- TurtleImage
    - Extends ImageView, stores a current position and orientation
    - Able to access all of ImageView's public methods (changing position, setting image)

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.getTextArea(), and the Enter button will be added to this HBox as well.

Example code (this is especially important in helping others understand how to use your API)
It should be clear from this code which objects are responsible for completing each part of the task, but you do not have to implement the called functions.

- Show actual "sequence of code" that implements the following use case:

- *The user types 'fd 50' in the command window, and sees the turtle move in the display window leaving a trail.*
- Write JUnit tests for your primary classes that show how you intend to create each class, call their primary public methods, and what return values you expect (and what their actual values should be).

Sequence of Code:

Front End

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

```
SceneUpdater update = myModel.update(CommandLinel.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(SceneUpdator update);
  - myTurtleImage.setX(update.getX());
  - myTurtleImage.setY(update.getY());
  - myTurtle.setRotate(update.getRotate());
  - this.getChildren().add(makePolyLine(update.getPoints()));
- CommandLine.clear();

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 Point2D(0,0));
        List<Point2D> newList = new ArrayList<Point2D>();
        newList.add(new Point2D(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().getTextArea());   //assert that TextArea was initialized
        assertNotNull(v.getTurtleWindow());       //assert that the TurtleWindow was initialized
        assertNotNull(v.getTurtleWindow().getTurtle());     //assert that the Turtle was initialized
}

@Test
public void testTextAreaClear(){
        CommandControl myControl = new CommandControl();
        myControl.updateTextArea("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 which also 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

Questions

1. **When does parsing need to take place and what does it need to start properly?**

When the enter button is clicked, the string is sent to the back-end to be parsed. The parser needs a string to start parsing.

2. **What is the result of parsing and who receives it?**

Parsing results in taking in the input from the user and then interpreting that information so that it can be useful to the program. This information is then passed to wherever it needs to go depending on the type of information. If the turtle needs to be moved, then a turtle class will receive the information. If the scene needs to be set at a different color, then it will be passed back to the front end. The receiver of the information that is parsed varies based on the information.

3. **When are errors detected and how are they reported?**

When the enter button is clicked, the string is passed to the back-end which will then parse the string and check for syntax errors first. When there is an error, the back-end will identify the error and notify the front-end by passing them an appropriate error message. The front-end will then display a notice to the user next to the TextArea box. The string won't get interpreted if there is an error.

4. **What do commands know, when do they know it, and how do they get it?**

Commands are classes that the back-end is responsible for. They store a string description (which is used to create the buttons and this string is part of the string that is passed to the back-end when the user clicks enter). The commands also store instance variables if necessary for example, the forward command would store "forward" and if the specific command is forward 50 then it would also store the integer 50.

5. **When are commands executed?**

Commands are executed when the play button on the GUI is pressed. This will then pass the commands to the back-end to parse.

6. **How is the GUI updated after a command has completed execution?**

The front-end gets a list of points from the back-end and we create lines between each of those points. The final direction of the of the turtle will also be passed to the front-end so we will rotate the final turtle image to the appropriate degree.

7. **How can the commands be insulated from knowing if there is only one turtle or multiple turtles?**

There will be a command that allows the user to determine which turtle they are putting the commands in for. This will allow the user to switch between any of the turtles which are labeled on the scene. The current turtle receiving commands will be highlighted.

8. **How can extra features be grafted onto initial design instead of changing it? To what extent is this possible?**

In general, create a new class. For example, if you want another command just add a new class for that command. Make sure that the class contains all the relevant information so that the back-end can interpret it.
The view has a borderpane that can have any new visual components added to it when necessary. However, elements within the BorderPane are their own classes, so adding a button for example simply requires modifying the button-containing class (CommandControl) rather than modifying the whole view class. Changing the background color would involve just modifying the TurtleWindow class.