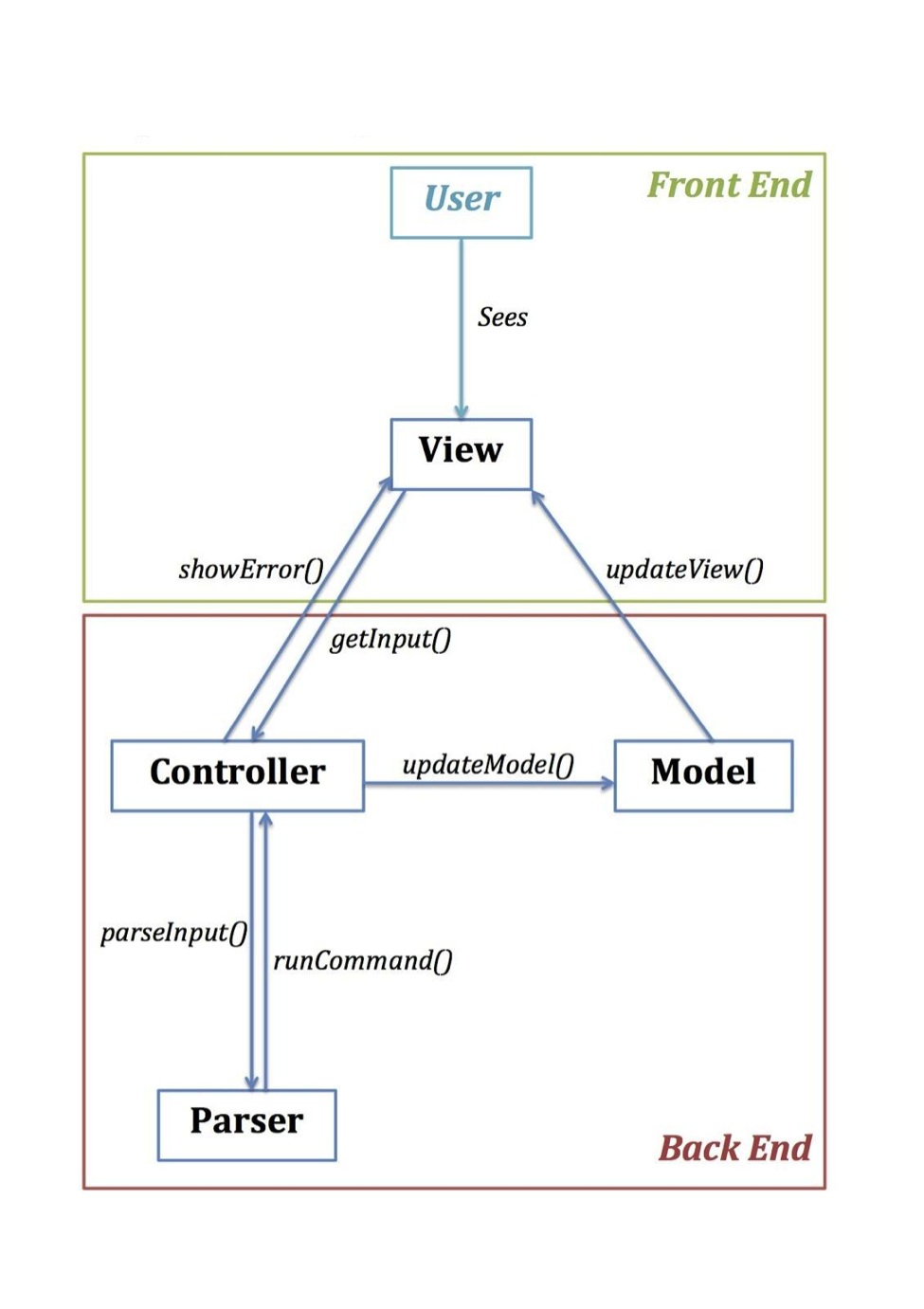
SLogo API

CompSci 308

Team 14 - Arihant Jain, Minkwon Lee, Jesse Ling, Benjamin Reisner, Martin Tamayo

**Design Goals:**

The primary goal of this design process is to intelligently break up the program into modules/sub-modules in such a way that additional Logo commands/command structures can be easily implemented to extend program functionality. The program is split into three modules: front-end, model-view-controller, and back-end. The front-end will be the grouping of classes that are responsible for the user interface, receiving user inputs, and presenting appropriate program outputs to the user. The model-view-controller will be the grouping of classes that are responsible for retaining the state of the program (within the model), calling on the back-end to parse the user inputs and then updating the model with the parsed commands (via the controller), and finally presenting the updated model through the view (which is pushed to the front-end). The back-end is the grouping of classes responsible for parsing the user input commands and returning a parsed, processed, standardized, and formatted command structure that is executable by the controller. It also will contain the classes that hold the different commands, math operations, etc. The back-end classes will be further modularized in order to maximize flexibility and extensibility, allowing for easy implementation of additional commands, operations, and command structures. The user inputs received by the User Interface is passed onto the controller in the form of commands. The controller passes the command to the parser which processes the input command string and returns a formatted structure of commands to the controller. Then controller then executes the commands and updates the state of the model. When the model is updated, the view is called to update its state as well. As the view is updated its new state is reflected in the User Interface thus the user is presented the updated state.



**Primary Classes and Methods:**

***Front-End: View Package:***

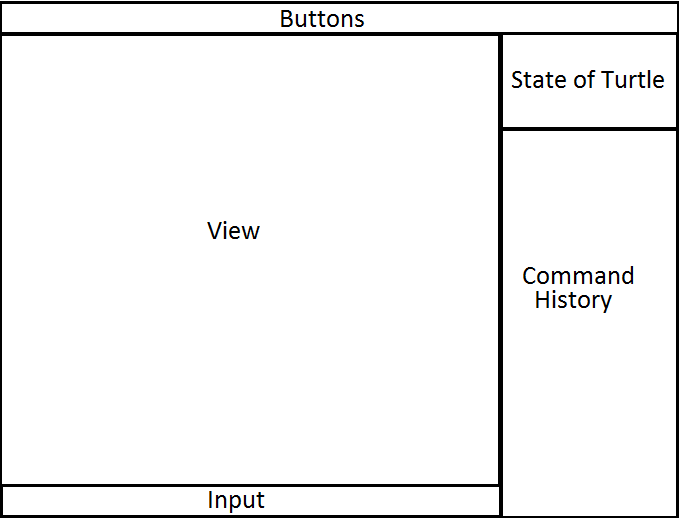


Figure 1: “ScreenShot” of the Intended View Interface

//Class that creates the GUI presented to the user upon start of the program. Will have buttons allowing user to set parameters graphically. These will function by passing specific commands to the controller input (as if a user had entered the commands). Will also contain a Workspace class, which is a grouping of panes specific to the instance of the program.

public class UserInterface

public methods:

static void main (String[] args): Main Method

//Class that groups together the panes specific to an instance of the program. Maintains the view, state of turtle, command history, and input box for a running program. This class’s main function is to hold the other panels interacting with each other.

public class Workspace

//Class that contains and displays arraylist of commands typed in.

public class HistoryPanel

//When a user selects a previously executed command from the history pane the command will be passed to the controller as a string for execution

public void toCommand()

//Allows for an error code to be send to the History Command window to be displayed to the user

public void showError()

//Class that contains a list of active turtle. Allows user to see turtle and where it is located

public class TurtleListPanel{

// This class holds the area for writing commands, a button for executing commands. The panel organizes all of its components. An instance variable of this panel is present in the WorkSpacePanel.

public class InputPanel

// Class containing the view. Shows the current state of the model and allows for the view to the updated when the state of the model changes

public class ViewPanel

//Called by the model when the state of the model changes. Update view will update the graphical display contained in view with the new state of the model.

public void updateView()

Constructor(s):

Public Methods:

public void showState(): show headings and coordinates of all the TurtleImages

// contains the buttons that create a Menu panel

public class MenuBarPanel

***Model-View-Controller***

Model:

Keeping track of Turtle (x, y positions, pen up/down, pen color)

Notifies it’s associated Views and Controllers that the Model state has changed.

Controller:

Accepts commands from user as strings.

Passes strings to parser.

View:

Grid.

Old commands.

***Back-End***

public class Parser

// Contains methods to parse strings and interpret the intended commands. Throws errors when invalid command input is encountered.

//gets a string commands from controller class, which was previously passed down from UI

public void parseInput()

public interface Commands

// Individual commands to be executed are mediated through an interface.

public class Controller

// Where commands are executed, based on the results acquired by the parser.

public void getInput() // Receives input from the user interface.

public void runCommand() // Executes (valid) input from the parser.

public class Model

// Turtle resides here. Updated based on info from controller and passes it along to the view.

public void updateModel()

**Example Code:**

“fd 50” is accepted as a string by the user interface and passed to the controller as a string.

The controller passes the string to the parser.

The parser recognizes “fd” as a command and “50” as its argument.

This information is used to call a method to update the position of the turtle in the model.

Upon execution of the method, the model checks the color of the pen and whether it is down.

The model constructs a line and passes it to the view. This line is then displayed to the user.

// User types “fd 50” into the View.

// Let X = 40, Y = 70, and bearing = 180° (down).

String userInput = “fd 50”;

myController.getInput(userInput);

myParser.parseInput(this, userInput); // Parser must know which controller to return to.

myController.runCommand(new ForwardCommand(50));

myModel.updateModel(command); // The command is stored in some variable at this point.

Line drawing = new Line(xPrevious, yPrevious, xCurrent, yCurrent); // 40, 70, 40, 20

// There will be private methods for calculating where to draw lines, etc.

myView.updateView(drawing); // This will likely be passed as an array/queue of lines.

Possible JUnit test for the Parser class:

@Test

public void testParser(){

// The following is intended to be solely conceptual.

Parser testParse = new Parser();

Exception e = testParse.parseInput("fd 27 23");

assertThat(e.getMessage(), containsString("Command fd takes 1 argument(s)"));

e = testParse.parseInput("fd");

assertThat(e.getMessage(), containsString("Command fd takes 1 argument(s)"));

e = testParse.parseInput("+ 3 7 1");

assertThat(e.getMessage(), containsString("Command + takes 2 argument(s)"));

e = testParse.parseInput("+ + + +");

assertThat(e.getMessage(), containsString("Command + takes s argument(s)"));

e = testParse.parseInput("xyftkjplp");

assertThat(e.getMessage(), containsString("Invalid input"));

e = testParse.parseInput("fd sum 67 21 ]");

assertThat(e.getMessage(), containsString("Invalid syntax"));

Command com = testParse.parseInput("fd 50");

FowardCommand forCom = new ForwardCommand();

assertArrayEquals(com.getClass(), forCom.getClass());

com = testParse.parseInput("sum 12 12");

SumCommand sumCom = new SumCommand();

assertArrayEquals(com.getClass(), sumCom.getClass());

com = testParse.parseInput("ifelse less 17 sum 12 6 fd 77 home");

HomeCommand homeCom = new HomeCommand();

assertArrayEquals(com.getClass(), homeCom.getClass());

}

**Alternate Designs:**

Our alternate design lacked part of the MVC. Instead of having the Controller as a bridge between the Turtle and the backend, where the Controller received guidelines from the parser that told the Controller how to alter the state of the Turtle, we had the parsed functions directly set the state of the turtle via the backend. For example, if the parser parsed “fd 5,” we originally had it so “fd 5” would directly move the turtle 5 pixels up once the command was parsed. Now, we have a Controller that receives some executable variation of the function “fd 5” and alters the turtle in the Model. We decided to use this implementation for a few reasons. We felt that the more we seperated the parser and classes that contain various commands/math/booleans, the cleaner our code would be and the better our design would be. We also felt that this is more extensible/malleable. If we need to change something with how the turtle is updated, it would be much easier to make necessary changes because we wouldn’t have to touch the parser or the classes that contain commands/math/booleans. If they directly altered the turtle, it would be very difficult to change each individual function that alters the turtle to fit the turtle’s new characteristics. We would only have to change the functionality of the Controller, so our changes would be focused to one narrow area. This also allows both the front end and back end to interact with the turtle without the need for either class to interact unnecessarily with the other or have multiple copies of the turtle. By utilizing the M-V-C there is now a uniformed model that contains the entire state of the program. Utilizing the controller to access the model prevents direct access to the model and allows specific definitions of what can and cannot be done to the model.

**Roles:**

Arihant Jain - Front End

Minkwon Lee - Back End

Jesse Ling - Front End

Benjamin Reisner - Back End

Martin Tamayo - Back End