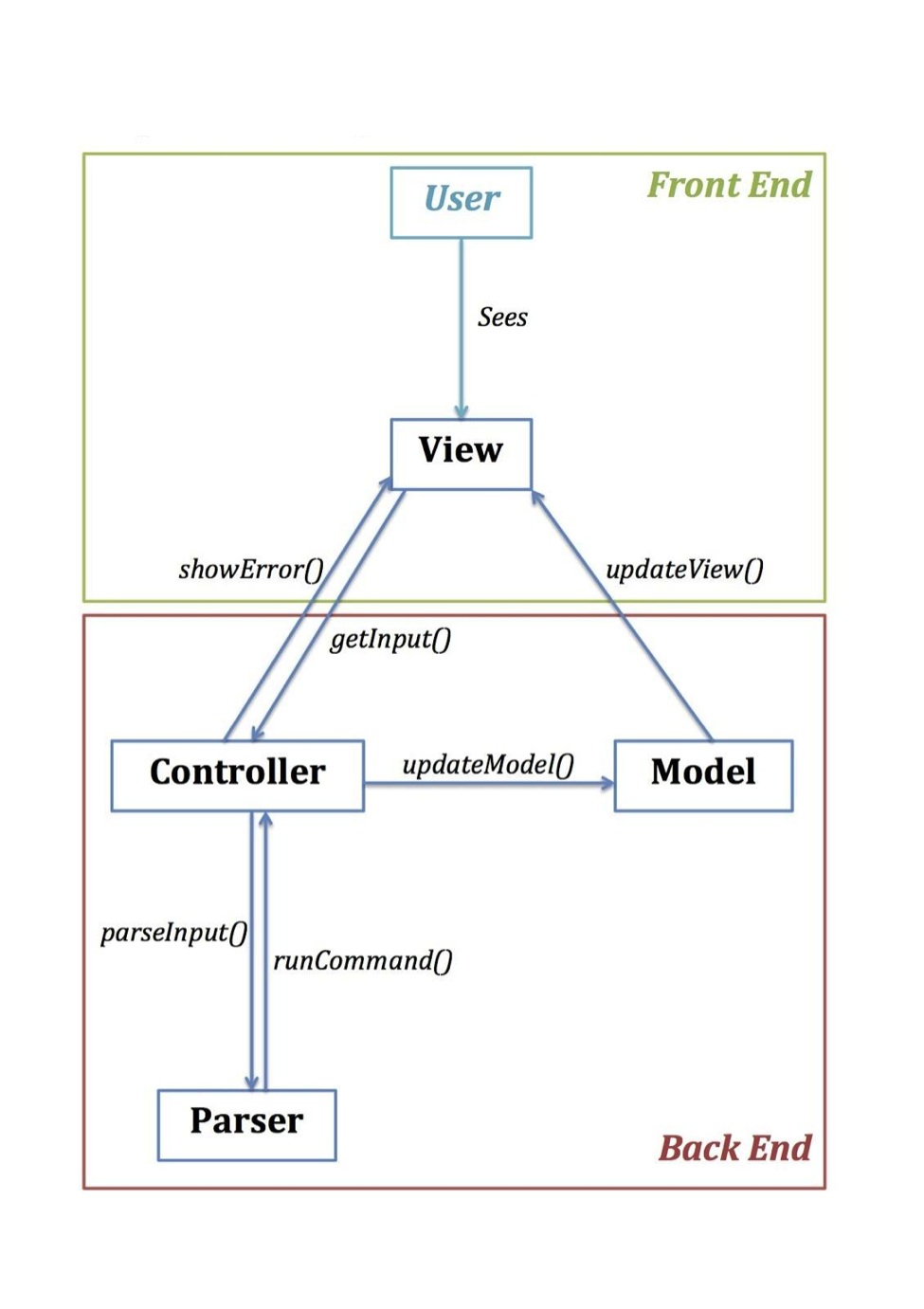
SLogo API

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:***

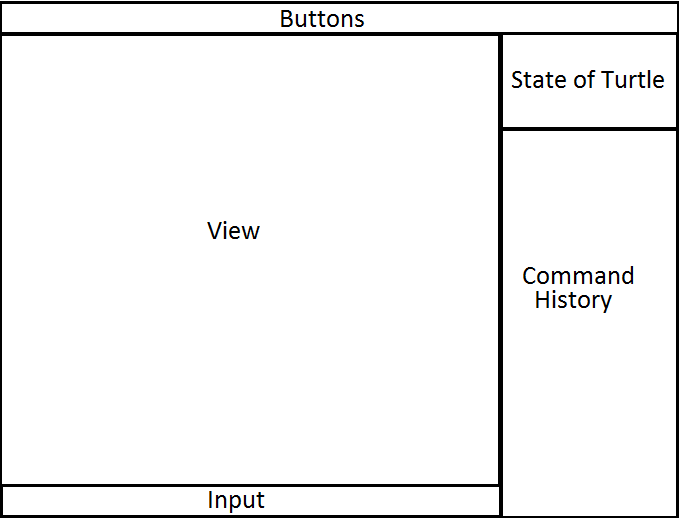


Figure 1: “ScreenShot” of the Intended View Interface

//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.

Passes commands to Model.

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:**

See example (stub) classes in the api branch.

“fd 50” is accepted as a string by a View and passed to a 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 create a Command class to update the turtle in the Model.

Upon execution of the Command, the Model checks the color of the pen and whether it is down.

Lines are constructed inside the Model to be passed to the View and displayed to the user.

**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

**Questions:**

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

Parsing takes place in Controller class, after UI passes a string of command to Controller class. To start properly, it needs Parser class, because Parser class, which takes a string of command as an argument, is able to turn the string into actually executable commands and returns a list of those executable commands to Controller class.

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

Parser class returns a collection of Command objects and Controller class receives it.

3 - When are errors detected and how are they reported?

Errors are detected in Parser class. An Error Command will be generated, returned to the Controller, and subsequently returned to the View without going through the model. The Error Command displays the corresponded error to the user upon execution.

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

They know their parameters (for example, “fd 50” represents a ForwardCommand that knows that its one parameter is 50). They know this upon instantiation in the Parser. They never learn anything else until they are used to update the turtle in the Model.

5 - When are commands executed?

Commands are executed in Model class, when executable commands are received from Controller class.

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

After Controller class updates Model class by updateModel(), Model class should update GUI (ViewPanel class) by updateView().

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

Turtles are distinguished by IDs. Commands will be carried out on specific turtles based on which ViewPanel is in use (each ViewPanel has its own Controller and Model) and based on which Turtle is currently selected in the ViewPanel.

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

The unique behaviors of a command will be coded into the command’s unique class to the fullest extend possible. Adding new commands should thus only require adding new classes for the new commands in question. To simplify this as much as possible, even buttons on the GUI can be sent to the controller as strings to parse into commands.