Model [external] [accessible to the View]

* parse input [returns the action] [regardless of language]
* parse file
* get history [commands and variables]
* store commands [save the commands]
* store workspace [save workspace]
* get command help page
* setting the language
* appendAction

Model [internal]

* get action [have a set of actions]
* create new actions [linked list of pre-existing actions]
* catch errors [other side of parse input]
* store/update the history

View [external]

* applyActions
* location of the turtle
* display error
* boundaries

View [internal]

* set background
* turtle image
* set pen color
* penUp
* penDown
* draw
* draw function [drawing the physical line]
* add workspace
* reset/clear
* moveTo

**Classes and Methods**

* **Model**
  + Commands [Back-end parsing]
    - Turtle commands [make actions → forward, left, right]
    - Math commands [+, -, /]
      * Arithmetic
      * Boolean
    - Program commands [for loops, if trees]
  + Action [takes turtle in as input] [front-end]
    - Forward
    - Back
    - Right
    - Left
* **View**
  + Drawer class
    - Turtle
  + Canvas
  + Workspace [exposed to the model]
    - Get saved commands
    - Has drawer
    - Has canvas
    - Has history

UNDO

* Undo commands
  + Forward → Backwards
  + Left → Right
* Have global action change list
  + Even changes in the view will change the global list

Save command → Return list of actions

* Holds hashmap of all commands

Storing history

* Parsing → Command to string
* Contains a group of strings

Workspaces

* Turtles
* Canvas
* Saved commands
* History

Discussion about saved workspaces

* Instead of saving workspaces, save chain of actions that got us to that state
  + Something that isn’t in our action change [clicking, color]

Turtle keeps track of color, pen up/down, hide/show

View receives list of actions from the model and it knows which turtle it’s associated with

* Tough if there are multiple turtles

View handles turtle selection

Only one turtle moves at a time [for now]

Having the turtle live in the view → Model doesn’t care

Workspace should be a class in the view

Should the Model care about the workspaces?

Array for store histories

Workspace should store history, language, commands

Model [external] doesn’t need to catch error because the view doesn’t need to call it

Store history/sequence of actions as a linked list [Loops]

Location of turtle (external v. internal)

Separate moveTo [turtle] and draw [pen] functions

Separate turtle class [create multiple turtles]

* Pen up/down
* Current location

View deals with all basic turtle commands

Model does all of the math and comparators

List of commands

* Return coordinates?
  + Issues → Drawing paths [sine/cosine waves], Pen up/Pen down

Turtle [moving cell] → Doesn’t change state, has information about itself

* Is waiting for actions to execute

Model sends actions to View and View applies action to the turtle

Model → Parse actions → List of actions → View → Updates

Action list helps with undo-ing actions

DESIGN DECISION:

* Location for turtle - stored in model or view or workspace
* Using nodes/linked lists to represent an action, link them together (vs. another class)
* how the workspace is stored

**Design Goals**

The purpose of SLogo is to create an integrated development environment (IDE) that supports users to write SLogo programs. In designing this IDE, we split up the work into two areas, a back end and a front end. The front end takes care of all the graphics and visualizations, sending user input to the back end and receiving information from the back end on what actions to perform or information to display.

The front end consists of two modules, the View class and the classes associated with the Workspace. The View displays all the information and handles action events for user input. The Workspace class has variables for Turtles that draw on the canvas, a Canvas, a command History, and a SavedCommands class. The View knows how to switch between Workspaces, display the appropriate information, and pass information to the back end. This enables greater flexibility with user input options and

The back end is broken up into two modules, the Model class and the Node-based classes. The Model takes care of parsing the user input (as strings) to obtain the core commands that should be performed in the Workspace. The Model can perform some of the commands like loading and saving files, but the parsing and evaluation of commands is largely assisted by the Node-based classes. As the strings of commands from the front end are parsed, new nodes are created. Nodes are broken up into several subclasses such as mathermatical expressions or Turtle commands, each of which knows how to evaluate itself based on its specific type of node and children. As a given line is parsed, an entire tree of nodes is formed, and from this tree, a list of Actions are created. Actions are essentially actions that should be performed by a Turtle or a Workspace, and live primaily within the front end, but are populated within the back end.

This architecture allows us to keep classes shy and eliminate repeative code while separating funcitons enough for multiple people to work on. The front end modules will collect user input and pass it to the back end, which will parse the input and populate a set of actions for the front end to perform wiht respect to the display.

**Primary Classes and Methods**

**Front End:**

***\*\*\* See UML Diagram for more details\*\*\****

Our Front end is composed of a central class View, which will facilitate our externally facing API functions, including:

**+ updateTurtles(List<Action> actionChain): null**

**+ getTurtleLocation(Location loc) : return**

**+ displayError (String errorMessage) : void**

**+ getCanvasBoundaries() : return**

The view will have member Workspaces. The apply actions externall facing API function will allow the Model to pass in a chain of **Actions**, represented as a **List<Action>** to our View. Our View will then handle these Actions by passing it onto the **currentWorkspace.**

A Workspace contains a **Turtle**, a **Canvas**, a **History**, and a **SavedCommands**. A Workspace has the following public methods:

**+ affectTurtle (List<Action> actionChain ) : null**

**+ display () : Group**

**+ saveCommands(String name, String commands) : null**

**+ getSavedCommands(String name) : String**

**+ getHistory(): List<String>**

**+ saveCommand(String): null**

Most importantly, the **updateTurtles()** method will accept an **actionChain**, and then apply this to the Workspaces member Turtle.

A Turtle is very simple, and is intended to be as dumb as possible. The Turtle contains the following member variables:

**- myIsPenDown : boolean**

**- myLocation : Location**

**- myImage : Image**

**- myPenColor : Color**

As you can see, the Turtle doesn’t know much besides it appearence, its location, and whether or not its pen is down. The Actions are applied to the Turtle, so the Turtle does not need to know how to handle them.

**Back End:**

***\*\*\* See UML Diagram for more details\*\*\****

Our back end is composed of a central class Model, which will facilitate our externally facing API functions, including:

**+ parseInput(String inputString): return List<Action>**

**+ parseFile(File inputFile): return List<Action>**

**+ storeCommands(String commands): void**

**+ storeWorkspace(Workspace workspace): void**

**+ getCommandHelpPage(): return Object**

**+ setLanguage(String language): void**

**+ appendAction(Action action): void**

The model will have associated node classes that are instantiated to represent code segments to be evaluated. After all the parsing is complete, a list of actions, represented as a **List<Action>**  will be sent to our View. Our View will then handle these Actions by passing it onto the **currentWorkspace.**

Each SLogoNode will extend Node and be able to get its children, get its parent, set its children, and evaluate itself:

**+ evaluate() : return SLogoNode**

Depending on the type of SLogoNode, it will have a different implementation for evaluation. FOr exmaple, SLogoNode will likely be an abstract class extended by several subclasses such as mathermatical expressions, programming tools (like loops), and Turtle operations. Each of those will in turn be extended as concrete classes for real operations like boolean operators, moving a turtle forward, or looping through code. This design will require many classes, but each with very little code. This will make it easy to test and implement.

**Example Code**

**Code Runthrough:**

\*View sends input to Model to be parsed\*

List actionsToDo = myModel.parse Input(input)

\*Model parses math and programming commands into linked list of basic Turtle Actions\*

List actionsToDo = new List

Action action = getAction(eachInputCommand)

actionsToDo.add(action)

\*Model sends Actions to View for visual updating\*

for (Action action : actionsToDo){

action.update(myCurrentTurtles)

}

**Front-End example JUnit Tests**:

//test that commands history exists

@Test

public void testCommandHistory(){

assertNotNull(myWorkspace.getHistory())

}

//test that list of saved commands exists

@Test

public void testSavedCommands(){

assertNotNull(myWorkspace.getSavedCommands())

}

**Back-End example JUnit Tests**:

//test to see if the parseInput() method returns the right type

@Test

public void testParseInput(){

String sampleCode = “fwd 50”;

Action expectedAction = new Action();

List<Action> actionList = new ArrayList<Action>();

assertNotNull(parseInput(sampleCode));

assertEquals(parseInput(sampleCode),actionList);

}

//test to see that getCommandHelpPage() does not return null

@Test

public void testCommandHelp(){

Object helpPage = new getCommandHelpPage();

assertNotNull(helpPage);

}

//test to see if commands execute properly

@Test

public void testCommands(){

String sampleMult = “\* 5 4”;

Command

assertEquals(node.evaluate(), 20);

}

//test appendAction()

@Test

public void testAppendAction(){

Action a = new Action();

List<Action> actionList= new ArrayList<Action>();

actionList.appendAction(a);

assertNotNull(actionList);

assertEquals(actionList[0], a);

}

**Alternate Designs**

One of our design decisions was where we should keep all information about the turtle. At first, we discussed whether the model or the view should contain this information. The view would need informations about the turtle in order to be able to draw it properly. We also thought that the model would need information about the turtle in order to execute the actions. We decided that it would be best if the model would just return actions to the view, without caring what was actually doing the action in the front-end. Because of this, we realized that it would pointless if the turtle lived in the model because the model would not care about the turtle. It would only pass to the view what needs to be done. After deciding that the turtle belongs in the view, we decided that the workspace class would be the one to keep track of the turtle[s].

We also discussed how to keep track of actions for the history. A list of actions seemed ideal, but with actions such as pen up/down and hiding/showing the turtle, where the turtle isn’t actually moving, the list of actions didn’t seem to work as well. Especially when taking into consideration having the ability to undo commands, we were not quite sure how we would keep track of the actions that don’t cause the turtle to move. Because of this, we decided that the turtle would contain all the information about such actions. The turtle will contain information about whether it is showing or hidden, what the pen color is, and whether or not the pen is up or down, that way, we will still be able to use a list of actions to keep track of what has been done so far.

**Roles**

**Front End:** Nick Balkissoon and Wesley Valentine

* Nick and Wesley will be working on the View, Workspace, Turtle, and Action classes

**Back End:** Monica Choe and Pranava Raparla

* Pranava and Monica will be working on the Model and Node-based classes.