Design Goals

FRONT-END
Flexibility:
- Workspace appearance i.e. images/colors/workspaces
- Extensibility in adding multiple instances of existing components (workspaces, turtles, etc…)
- Adding exceptions/changing console output
- Adding new functionality through menubar items and mouseListeners

Assumptions:
- Keyboard interactions (no built-in functionality for directing turtle other than through keyboard interactions, but such features should be somewhat easy to implement.)
- Minimum window size needed for the GUI
- Moving tabs (user cannot move tabs around like they would be able to in an average web browser)

Primary Classes
1. SlogoFrame
    a. WorldGraphicsPanel
    b. StatsPanel
    c. ConsolePanel
    d. MenuBar (classes inside not mentioned)

Primary Methods
1. paint
2. populateDisplays

BACK-END
Flexibility:
- adding commands
- changes in parsing algorithms
- changing languages/command mapping
- adding exceptions/changing console output

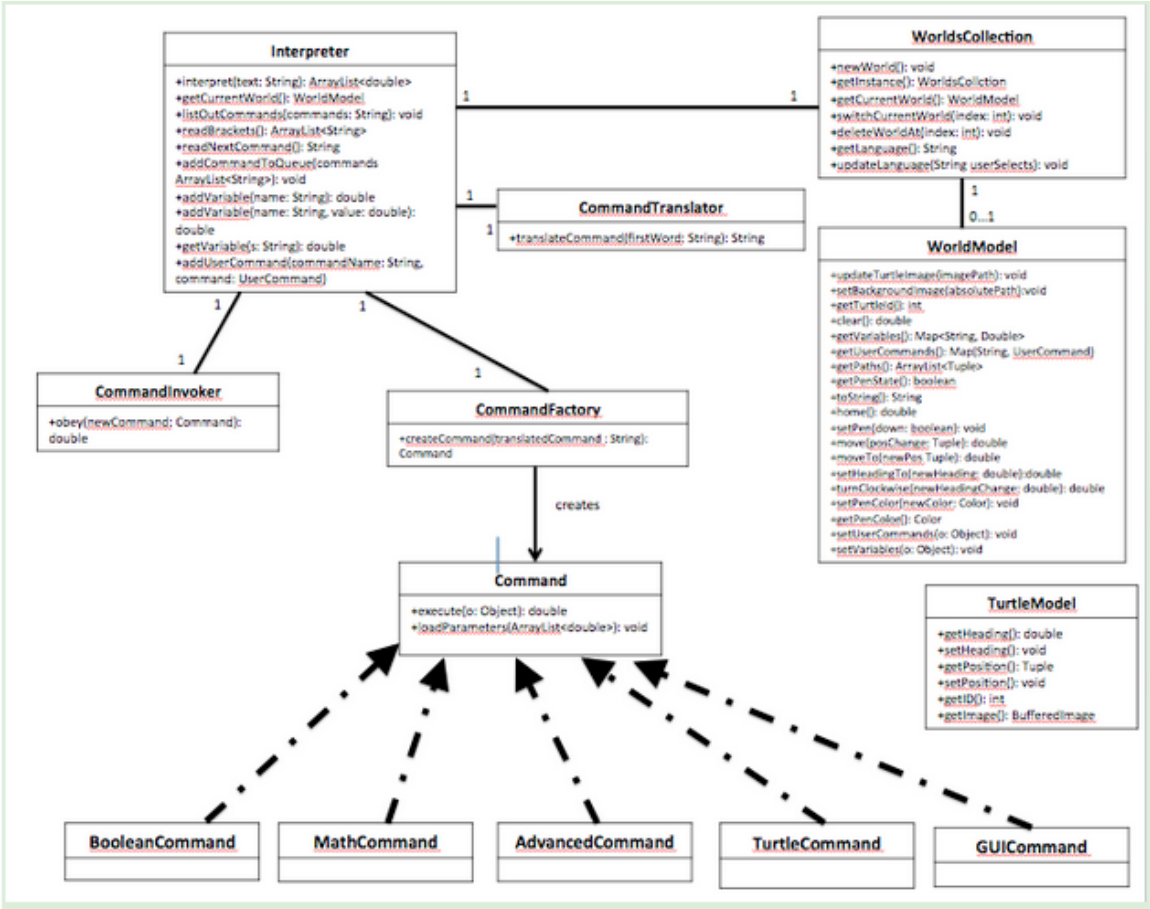
Assumptions:
- Whatever happens in an individual world is limited to that world. (world independence)

Primary Classes
1. Exception (and its descendents)
2. Interpreter
3. Parser
4. TurtleModel
5. WorldModel
6. WorldsCollection

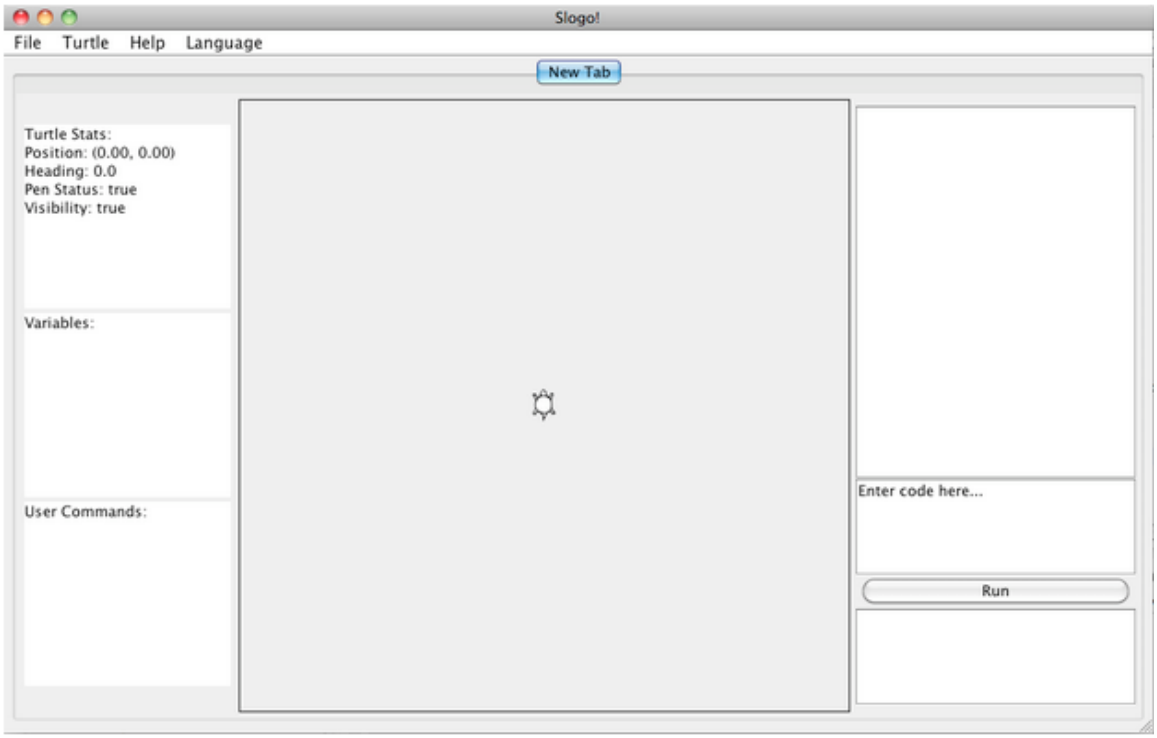
Primary Methods

1. interpret
2. obey
3. evaluateCommand
4. readBrackets
5. execute
6. loadParameters

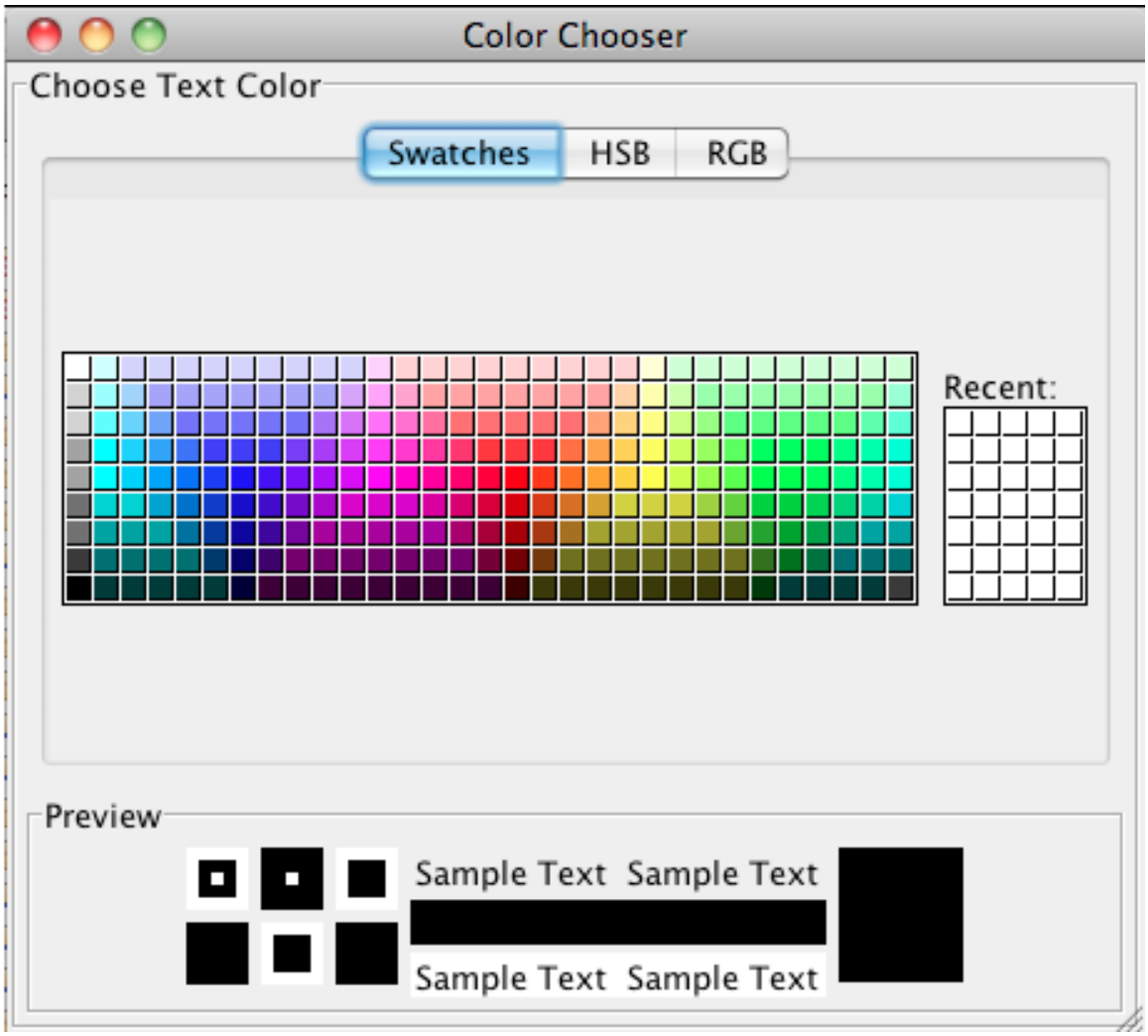
The core architecture of our program was as such: We had an interpreter which would receive the strings from the View interface, and then interpret the strings. From these strings, we would then create Commands using the CommandTranslator and CommandFactory. Then, the CommandInvoker executes the commands in the list.



Shown below are screenshots that highlight some of our view's aspects. The first image is the main screen. On the left side, there is a stats panel that updates with the turtle's information as new commands are entered and sections that list user made variables and commands. In the middle is the main turtle panel, which displays the results of entered commands. On the right side, are panels that display the command history, the input text box, the run button, and the output of the most recently entered command. At the top is a menu bar that includes functionality to complete tasks such as creating new tabs, closing old tabs, setting turtle images, setting pen color, bringing up help documentation, and changing the language.



The next image is our ColorChooser window that is brought up when the user sets the pen color from the main screen's menu bar in the turtle section. Inputting a color from any of the three methods will change the color of the line that the turtle draws.



Example code:

New commands can be added by creating a command class within the appropriate command package. The command class will extend the appropriate Command superclass, and call the super constructor with the number of parameters it wishes to take in. For example, the SumMathCommand class is as follows:

```
public class SumMathCommand extends MathCommand {
	public SumMathCommand() {
		super(2); //2 parameters
	}
	@Override
	public double execute(Object o) {
		return parameters.get(0) + parameters.get(1);
	}
}
```

Special parsing (i.e. within brackets) are handled by the interpreter but must be called through the execute method in the command class. Then, the command must be added to Commands.properties and other language properties files to map it to a Slogo command. The sum

command in the Commands.properties file is added with this line:
"Sum=commands.mathCommands.SumMathCommand".

Implementation of 'fd 50' command

When this command is run, the ActionListener for the 'Run' button grabs the text from the inputTextArea (text box for commands), and passes it to the Interpreter's interpret method and copies it to the command history text box. The interpret command breaks down the string into a list of commands, which is evaluated by the evaluateCommands method. This method begins from the front of the list, identifies the type of command, and returns the appropriate values by recursively evaluating parts of the list of commands. In the 'fd 50' command, 'fd' is identified by evaluateCommands as a command. This results in the CommandFactory being called to create the command via reflection. The createCommand method in the CommandFactory pulls the class name from a map of commands parsed from the properties files and creates the appropriate command. The command created here would be ForwardTurtleCommand. Then, evaluateCommands takes the command and parses the appropriate number of parameters from the list of commands passed to it. After parsing the parameters (50 in this case), they are loaded into the command using loadParameters method and the command is executed using the execute command, all belonging to the command. The execute command method calls the turtle's move method. The move method takes in a Tuple, which embodies changes in the turtle state. This tuple is passed to the WorldModel, sets the turtle's visual position and then refreshes the drawing of the turtle.
The code snippets responsible for executing 'fd 50' is below:

GUI:

```
try {
                                        consoleOutputTextArea.setText(""
                                                +
interpreter.interpret(inputTextArea.getText()));
                                } catch (SlogoException e1) {
                                        consoleOutputTextArea.setText("" + e1.getMessage());
}
```

interpret:

```
listOutCommands(text);
while (listOfWords.size() > 0) {
                        try {
                                evaluatedValues.add(evaluateCommand(listOfWords));
                        }
```

listOutCommands:

```
                listOfWords = new ArrayList<String>();
                String[] words = commands.split("[\\s\\n]+");
                for (String word : words) {
                        listOfWords.add(word);
                }
```

evaluateCommand:

```
if (isConstantValue(firstWord)) {
```

```
                return Double.parseDouble(firstWord);
}
else if (isCommand(firstWord)) {
ArrayList<Double> parameters = new ArrayList<Double>();
                Command newCommand =
commandFactory.createCommand(commandTranslator.translateCommand(firstWord));
                for (int i = 0; i < newCommand.NUM_OF_PARAMETERS; i++) {
                        parameters.add(evaluateCommand(wordList));
        }
newCommand.loadParameters(parameters);
commandInvoker.obey(newCommand);
}
```

loadParamters (in Command):

```
this.parameters = parameters;
```

execute (in ForwardTurtleCommand):

```
        WorldModel t = WorldsCollection.getInstance().getCurrentWorld();
        return t.move(new Tuple(0, parameters.get(0)));
```

move (in WorldModel):

```
moveTo(Tuple.sum(actualPosChange, turtle.getPosition()));
```

moveTo (in WorldModel):

```
Tuple onScreenPosition = Tuple.subtract(Tuple
                        .mod(newRawPosition, new
Tuple(WorldGraphicsPanel.SCREEN_DIMENSION.width,

        WorldGraphicsPanel.SCREEN_DIMENSION.height)), new Tuple(553.0 / 2,
                                                        553.0 / 2));
```

Tuple instance variables:

```
public double x, y;
//Several methods for tuple manipulation such as sum, mod, and distanceTo
```

Alternatives

Parsing - regex,
Display - jgame, JavaFX
Commands - lots of different ways

We chose to not use regex because for larger programs we don't want to store everything, so using a word by word parser uses less memory. Our implementation basically operates like an input stream.

For the display, we chose not to use jgame to lessen the dependencies of the project. Also, integration of multiple graphic systems is very troublesome and hard to seamlessly integrate while maintaining good readability.

Commands - We initially thought it was best to write a class for every command. For example, our original plan was to have a ForwardCommand be created and then added to a queue, and then the Engine would handle running that ForwardCommand to manipulate a Turtle Object.

Team Roles

Our team originally split up distinct front-end and back-end roles. Zanele was responsible for developing the parser and working on the data structures. Dan was responsible for working on the data structures and the advanced commands. Jeff was responsible for working on the view and testing. Cody was responsible for the GUI. Over the course of the project, however, the roles were blurred and the team ended up working together on many features. For instance, Dan implemented the turtle commands, and Zanele used the model for command implementation to add math commands. A simple version of the GUI was originally developed by Jeff and flushed out by Cody. The commands were created by Zanele and Dan. The interpreter structure was defined by Jeff. Multiple workspaces and other extra features in the latter part were implemented by all of us. Jeff and Zanele refactored much of the model-view paradigm. This type of teamwork allowed us to develop a holistic view of the project and for us to work on the design together.