Eric Chen
Michael Deng
Michael Ren
Nick Widmaier

CS 308: SLogo Planning

Design Goals:

Frontend:

Our goal from the front end is to not have it be exposed to any of the logic that goes into updating our display. We want the backend to take in the commands and parse them appropriately, and then pass the values to our frontend-backend interface which will then use these values to update our visual display for the user. Our front end then is not exposed to any of the command logic. In essence, our main module will launch the program, our graphics module will populate the initial scene and set the functionality of all the UI components, and our frontend-backend interface module updates our graphics based off the values returned from the backend. We want this so that when you can add new commands to the backend, without really altering any of our front end components. The visual display will still update correctly, as long as the logic was implemented correctly in the backend and updates are passed back to the interface in the correct format. Also, you can add additional UI functionality to the view, like buttons, or new windows, which will not affect the backend functionality.

Backend:

The backend will take care of interpreting a string representing a command and returning the state of the turtle and its environment after the command's execution. To do so, the backend will engage in three sequential steps: tokenization, parsing, and evaluation. Tokenization will take the string passed in by the frontend and break it up into meaningful chunks that can be recognized by the parser. The parser will then take these chunks, or tokens, and construct a syntax tree representing the command to be run. Evaluation will take this tree and walk through it, performing the operations associated with each individual node in the tree. This evaluation step will then return the turtle's context and then pass it to the front end.

We want to design the backend such that it is extensible in terms of adding new operations (e.g. divideAndThenMultiply, etc.), new syntax, and different human languages. The latter will be achieved by feeding a different set of rules into the tokenizer.

Frontend backend relationship:

The goal of this frontend-backend interface is to essentially have it be passed a set of values that represent the appropriate updates needed for our graphics module. Initially, the handler will take in the command string from the Graphics textbox, and then pass it to the back end classes to be parsed. After this command is parsed, a set of values corresponding to appropriate display updates is passed back to this interface handler, and the methods to update the turtle and other relevant UI components are executed based on these return values. Thus, this handler in essence acts to pass the command to the backend where it is parsed, and passes what's returned back to the display. It does not need to know the exact logic of the commands, as long as it gets back values from the backend that are of a correct format so that it can handle updating the visual display.

Primary Classes and Methods:

Frontend:

For the Frontend we anticipate having a Main class, a Graphics class, a Turtle class, and a BackEndHandler class. The Main class will be where Slogo will be launched and have a start method to initiate a new instance of our Graphics class. This Graphics class will be the scene displayed to the user, and will hold all the UI elements the user will see - the buttons, the labels, the text box for the user to input commands, a text box to display previously entered commands, and a drop down menu to select a color.There will be methods in the Graphics class to create all of these items and set their functionality, allowing the user to interact with them appropriately. The Graphics class will also have an initialize method that will set the initial scene for the user once the program is launched. The turtle class will be used to provide the image the user sees in regards to their location in the display window. Since one of the requirements is to allow the user to choose an image for the turtle, this class will have to have in essence a setImage method that sets the image that the user will be shown based on their input. The last class in the Front end will be our BackEndHandler class, which is needed for our interaction with the back end so that the display can be updated correctly. It will have a method getValue which will take in the command string from the UI textbox that is inputted, and pass it to the backend to be calculated. Then it will have methods moveTurtle and setVariable which will take in the returned values from the backend and update all the graphical components appropriately.

Backend:

API reference:
For a complete reference of backend interfaces, please consult the branch "api_design" inside our group's repository (https://github.com/duke-compsci308-fall2014/slogo_team03/tree/api_design).

The interfaces should be roughly complete, but documentation is not at 100%, and some method signatures may need to be changed (and written down / communicated) later.

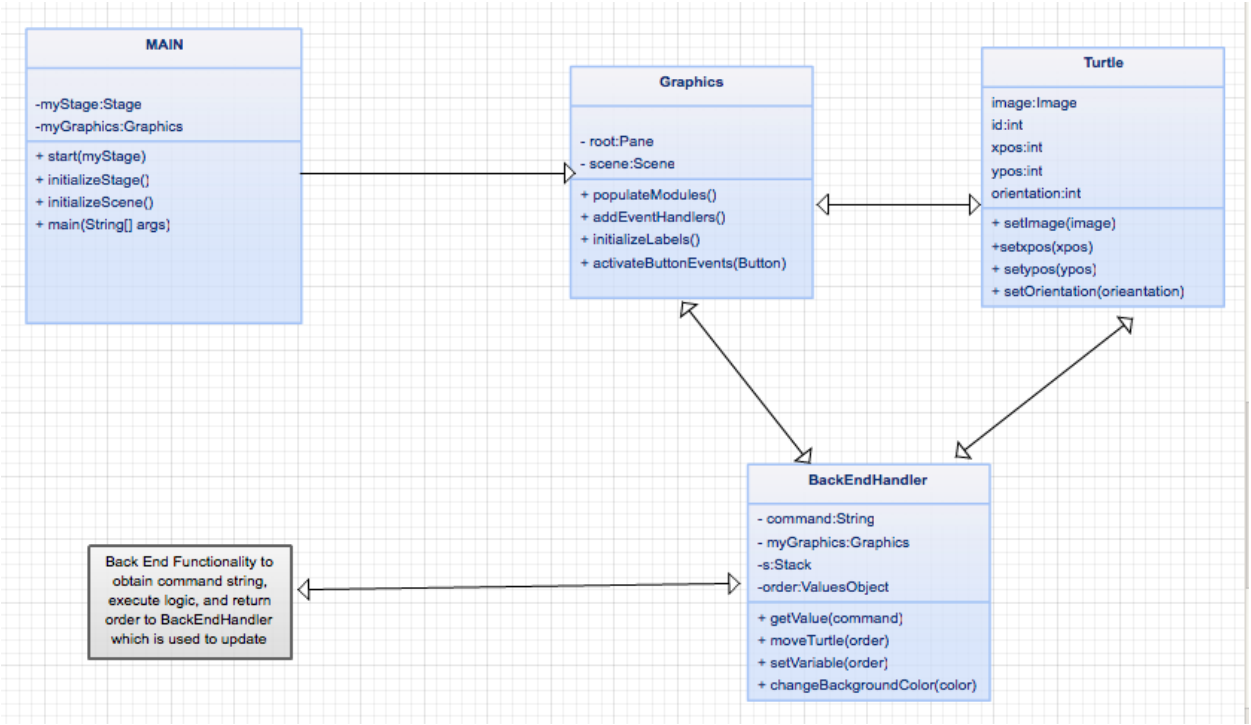
General Model
The backend contains a gatekeeper interface called IModel which contains a single method update() which transforms a string into the updated state.

Tokenization
We have an interface IToken which contains information specific to particular token, such as its location in the source, the associated text, and its token type, e.g. identifier, addition, etc. These tokens are generated when ITokenizer calls tokenize() on a Reader; tokenize() will use the rules loaded via loadTokenRules() by calling ITokenRule.match() to generate ITokens. ITokenizer.tokenize() will generate a list of tokens to be fed to the parser.

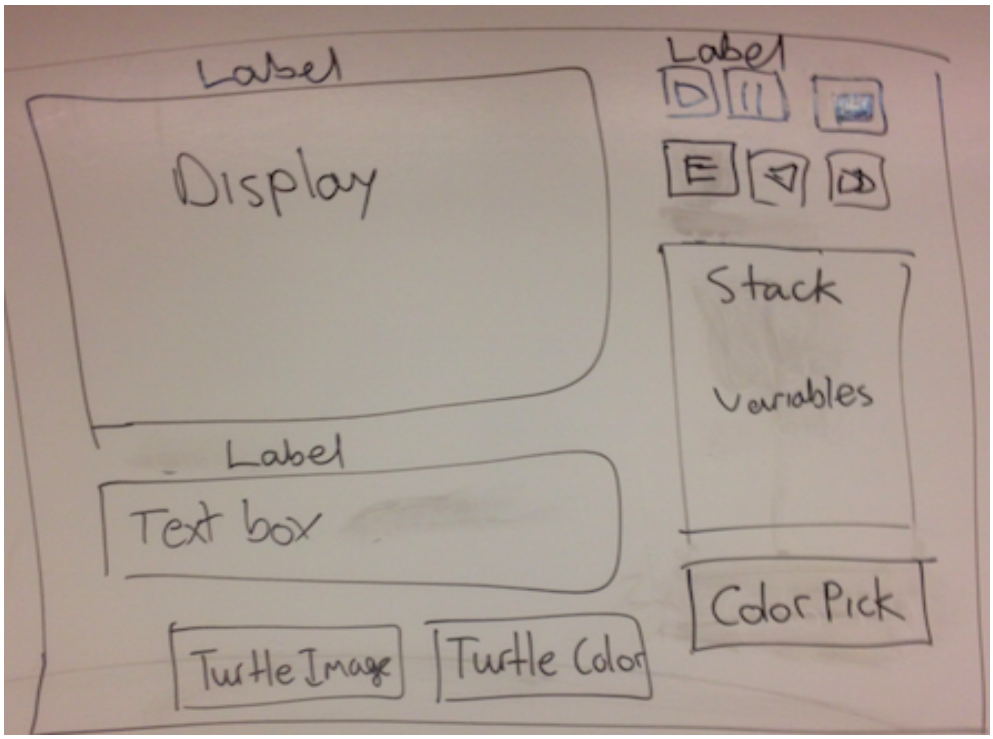
Parsing
The Parser API has a similar structure to Tokenization; IParser corresponds with ITokenizer, IGrammarRule corresponds with ITokenRule, and IASTNode corresponds with IToken. Basically, IParser.parse() takes in a list of tokens, e.g. from ITokenizer.tokenize(), and it emits a syntax tree (the root IASTNode), applying grammar rules by calling IGrammarRule.produce(). Each IASTNode contains an associated IOperation that is defined by the grammar rules in IGrammarRule.produce().

Evaluation
IEvaluator has a single function, evaluate(), which takes the syntax tree and evaluates, bottom-up, the operation on each node, using its children as input. This will merge all nodes until the only the root node remains; the result of the root node's evaluate() is then passed back to IModel, and then to the client. The standard format for representing the state after execution is IExecutionContext, which contains information on the turtle and the variables used, for instance the last return value. ISerializer also allows serialization from the syntax tree back into a string representation. Whether there is data lost in deserializing from string to syntax tree is left as an unresolved question.

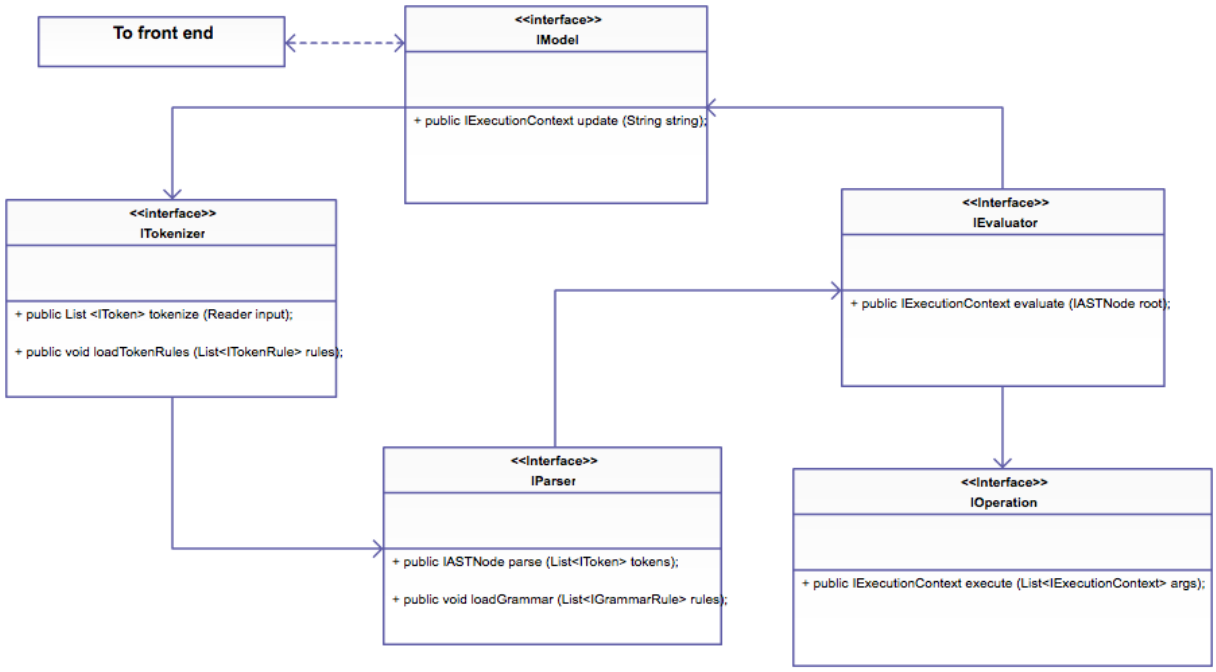
Frontend UML:



Visual Display:



Backtend UML:



Example Code:

Frontend:

```
Class Main() {
	void start(primarystage) {
		InitializeStage();
		InitializeScene();
		Graphics.PopulateModules();
		Scene.show();
	}
}

Class Graphics () {
	void PopulateModules() {
		Label label = InitializeLabels();
		TextBox tb = InitializeTextBox();
		Button btn = InitializeButtons();
		Colorpicker cp = InitializeColorPicker();
		AddEventHandlers();
	}
	void AddEventHandlers() {
		ActivateButtonEvents(btn);
```

```
            ActivateTextBoxEvents(tb);
            ActivateColorPickerEvents(cp);
        }
        Label InitializeLabels() {
            Label label = new Label();
            label.setColor("Red");
            label.setFont(4);
            label.setXPosition(100);
            label.setYPosition(100);
            return label;
        }
        ActivateButtonEvents(Button btn) {
            btn.setOnAction (new EventHandler<action>) {
                Public action {
                    run action;
                    run event;
                }
            }
        }
}

Class BackEndHandler() {
        Stack s;

        Command GetValue(String command) {
            Command order = Backend.calculateValue(command);
            return order;
        }
        void moveTurtle(Command order) {
            if (order.instruction == TurnRight) {
                Turtle.TurnRight();
            }
            s.push(order);
        }
        void setVariable(Command order) {
            Var x = order.value;
            Textbox.add(order.name);
            s.push(order);
        }
        void changeBackgroundColor(Color color) {
            Display.changeColor(color);
        }
}
```

JUnit Test:

```
Class Test() {

        @Test
        void BackEndHandlerTester() {
                BackEndHandler BEH = new BackEndHandler();
                Command ForwardCommand = createForwardCommand();
                Command Sum = createSumCommand();

                assertsEquals( FowardCommand, BEH.GetValue(“Fd 100”);
                assertsEquals( Sum, BEH.GetValue(“Sum 10 10”);
                assertsEquals( True, BEH.GetValue(“1 > 0”);
                assertsEquals( False, BEH.GetValue(“100 < 67”);
        }
}
```

Backend:

```
public interface IModel {

public IExecutionContext update(String string);

}

public interface ITokenizer{

public List <IToken> tokenize (Reader input);

public void loadTokenRules (List<ITokenRule> rules);

}

public interface IParser {

public IASTNode parse (List<IToken> tokens);

public void loadGrammar (List<IGrammarRule> rules);

}

public interface IEvaluator {
```

```
public IExecutionContext evaluate (IASTNode root);

}

public interface IOperation {

public IExecutionContext execute (List<IExecutionContext> args);

}

public interface IExecutionContext {

public Map <String, ITurtleStatus> turtles();

public String returnValueString();

public Number returnValueNumber() throws NumberFormatException;

public map <String, String> environment();

}

public class Model implements IModel {
	public Model(){
		this.tokenizer = new Tokenizer();
		this.parser = new Parser();
		this.evaluator = new Evaluator();
		this.serializer = new Serializer();
		this.tree = new ASTNode();
	}
	public IExecutionContext update(String string){
		tree = parser.parse(tokenizer.tokenize(string));
		return evaluator.evaluate(tree);
	}
	public Reader save(){
		return serializer.serialize(tree);
	}
}
```

JUnit Test:

```
Class Test() {
```

```
	@Test
	void ModelTester() {
		Model myModel = new Model();
		ExecutionContext context = myModel.update("FW 50");
		Map<String, TurtleStatus> map = context.turtles();
		TurtleStatus status = map.get("turtle");
		Coordinates coor = status.turtlePosition();
		Direction dir = status.turtleDirection();
		assertsEquals( 0 , coor.getX());
		assertsEquals( 50, coor.getY());
		assertsEquals( 0, dir.getDegree());
		}
}
```

Alternate Designs:

Frontend:

The two designs we were debating between were whether or not to have one large class that handles all the front end interaction and implementation, or to have a multitude of different classes that would be responsible for different functionalities and tasks. Because the backend for the project does the majority of the heavy lifting, we were debating have just one class that handles everything called Main. Main would be responsible for populating the main screen with all the different modules, activating all the modules' functionalities, and handling all the events fired by the different modules. Main would also be responsible for giving inputs and taking outputs from the backend. We first reasoned that this would be easier since the design for the frontend was so straight forward. However, we gradually came to the conclusion that this implementation would be cluttered and very difficult to read and make additions to. We ultimately decided to go with the design where we have three classes that handle three different tasks for the the frontend. These three methods would be Main, the module populator class, and the class that interacts with the backend and manipulates the frontend. We decided that this way would be a much better design because it would lead to both modularity, flexibility, and understandability. Firstly, the code would be much easier to read because it would be divided up into different purposes. Our code would also become a lot more shy because all Main would have to do is call run on these classes to have them work, which would allow changes to be made to any one of these classes without affecting the other classes.

Backend:

The first design we thought of was having one model that contained all of the different functions in it. We figured that having a generic model/view implementation would be the best choice that that functions could be easily organized in the model class with different methods. But then, after listening to Professor Duvall talk on Wednesday, we decided to switch our implementation drastically to have a lot more function classes. We had not realized how annoying and inflexible the code would have been if what we had to do was continuously add more methods to the model every time we wanted to add in another function. We also realized, after hearing Prof. Duvall speaking, that our code would have become very long very fast and would not have been as shy as we would have liked. We decided to go with an implementation similar to way Duvall showed in class, where each function has its own little subclass and we have two larger classes called Animation and Math that handle each of these functions when called upon. This not only makes our code a lot more readable by dividing up all the different sections of the backend based on functionality, but it also keeps the backend shy and modular. Now, any new additions to the program's functionality does not involve any changes to the existing code and one broken class does not affect the others.

Roles:

Michael Deng and Nick Widmaier:

Michael Deng and Nick Widmaier will be dually responsible for the creation of the front end design and implementation. Both will work together on the classes and methods implemented and will divvy up the work equally as the tasks role in.

Michael Ren and Eric Chen:

Michael Ren and Eric Chen will be responsible for the design and implementation of the backend. We will cooperate with Michael Deng and Nick Widmaier to integrate the back end with the front end.