Eric Chen
Michael Deng
Michael Ren
Nick Widmaier

CS 308: SLogo Planning

<u>Design Goals:</u>

Frontend:

Our goal for the front end is to not have it be exposed to any of the logic that goes into updating our display. To do so, we want the front end to pass to the backend the command string and a list of UI elements that we want to be updated from that command. The backend is where the logic for executing the commands will take place, and once the appropriate values are provided back to the front end, we can use these values to update each UI element accordingly. Thus, we separate the view from the logic that goes into updating the state of our view, and update this state once we have the correct data. We want to also be able to add UI elements as easily as possible to our view,will then parse and execute the command, giving back the correct data as a result of performing the correct logic.Our main main module will launch the program, our graphics module will populate the initial scene and set the functionality of all the UI components. We then want to have a UI Element factory, thus allowing greater flexibility to our front end if we decide to add additional UI elements. This factory allows updating each and any UI element to be much simpler, and we won't need to change any methods used in updating it. In all, we can add new UI functionality that the user can see without having to worry about altering the logic of the commands in the backend.

Backend:

The backend will take care of interpreting a string representing a command and returning the correct updates to the UI elements that were passed as a list to the backend along with the command. 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 an universal object that contains the items for the front end to update. The front end then can call upon its UI elements to draw and update based on this data. This is beneficial in the sense that adding new functionality to the backend, means that no work to the front end is needed, as long as the data is returned according to this set format.

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. The backend also will be where the errors are checked for, mostly involving improper syntax, and as soon as they occur, we have a call to a separate command which handles the error and also gives back an output in the same format as any other command.

Frontend backend relationship:

The relationship between the front end and the back end is to essentially have it allow for passing the command and the graphical components that need to be updated, and get back the data object that results from the command logic that is performed. This data object is sent right back to the front end, and the front end uses it to update the appropriate elements to be displayed to the user. Again, it is separated from the logic, transferring the needed data back and forth between the front end to the backend.

Primary Classes and Methods:

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).

Frontend:

General Model:
The front end contains a IView class serving as a gatekeeper interface that has two methods. init() simply starts up the UI and sets all the UI elements in place for the user to see and interact with as the initial starting point. It also has and error(String message) method which is used for notifying the view that an error occurred, with the error message itself as a parameter. The front end uses this to update all its UI elements appropriately.

UI Elements:
We have a IUIElementFactory, which has a method makeElement(String element). This allows that allows the UI to create a new IUIElement for the UI of a certain type, which is contained in the element String, and returns this IUIElement. An IUIElement is simply an element that appears in the UI, and it has draw() and update(String s) methods. The update(String s) method takes in the changes to be made and updates the state of the element, and the draw() method returns a Node based on these changes to the IUIElement so that they can be visually represented to the user.

We also anticipate having a Main class, a Graphics class, and a Turtle class. These are simply for the front end to worry about. 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. The turtle class will be used to provide the image the user sees in regards to their location in the display window.

Backend:

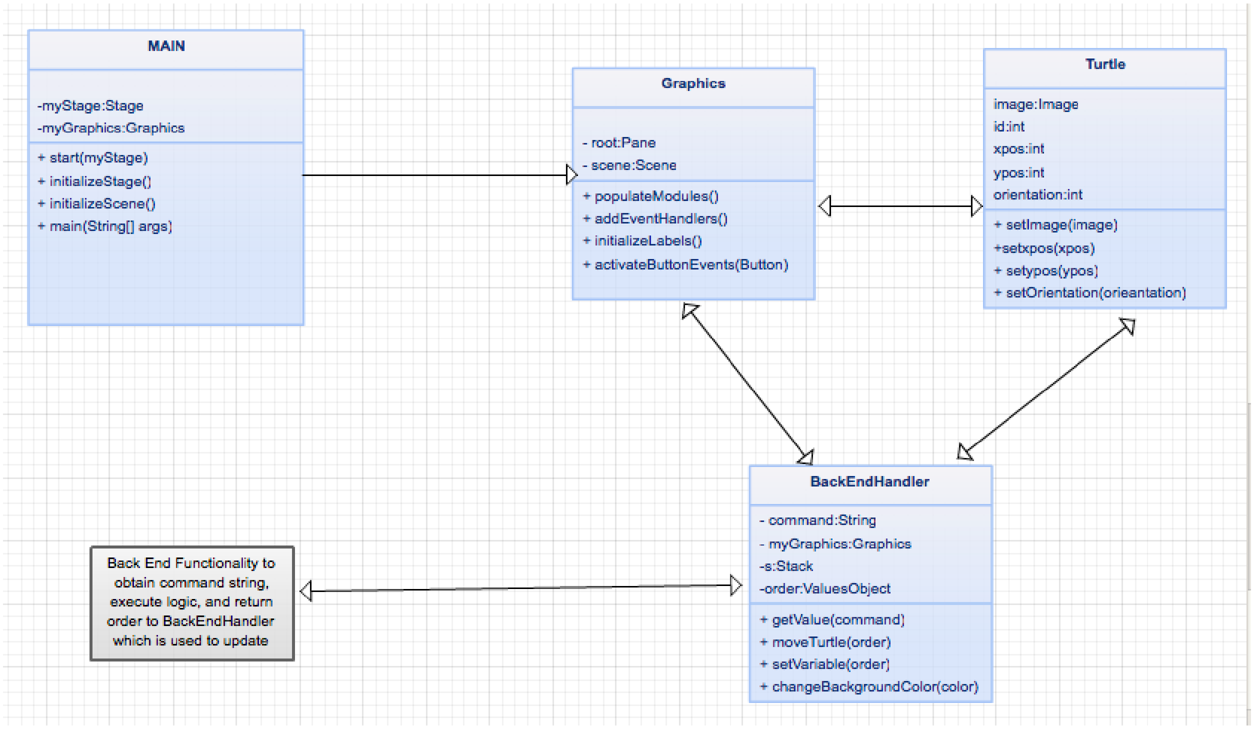
General Model
The backend contains a gatekeeper interface called IModel which contains a single method update() which transforms a string and an array of dependencies into a key and the updated state.

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.

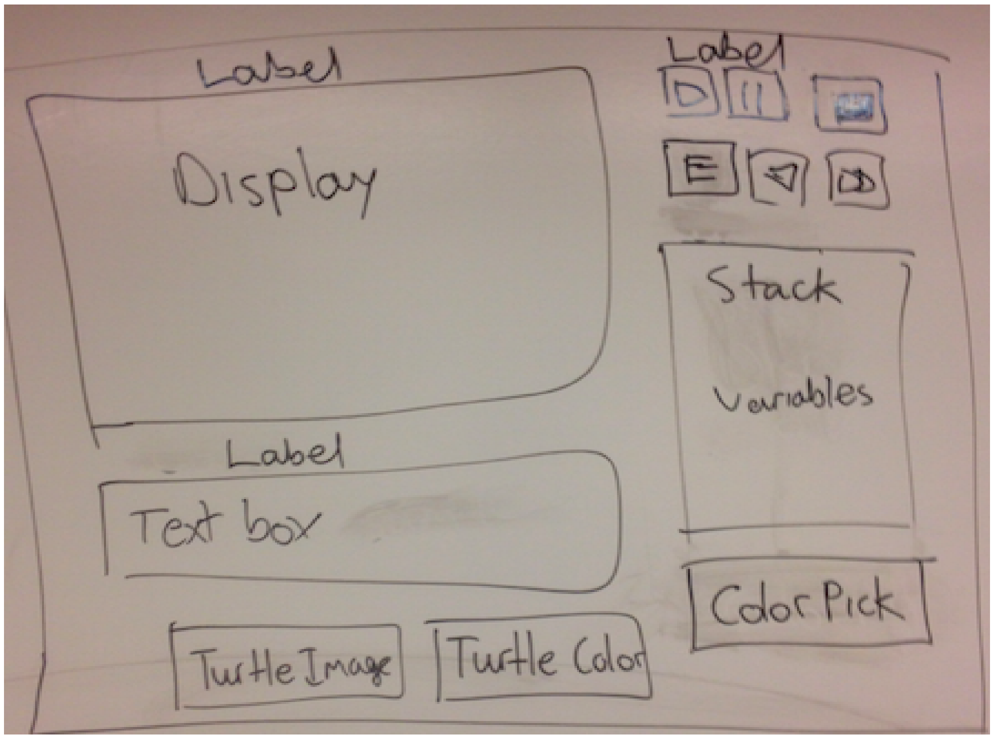
Parsing
The Parser API has a similar structure to Tokenization; IParser corresponds with ITokenizer, IGrammarRule corresponds with ITokenRule, and IASTNode corresponds with IToken. Basically, IParser.parse() takes in a list of tokens, e.g. from ITokenizer.tokenize(), and it emits a syntax tree (the root 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.

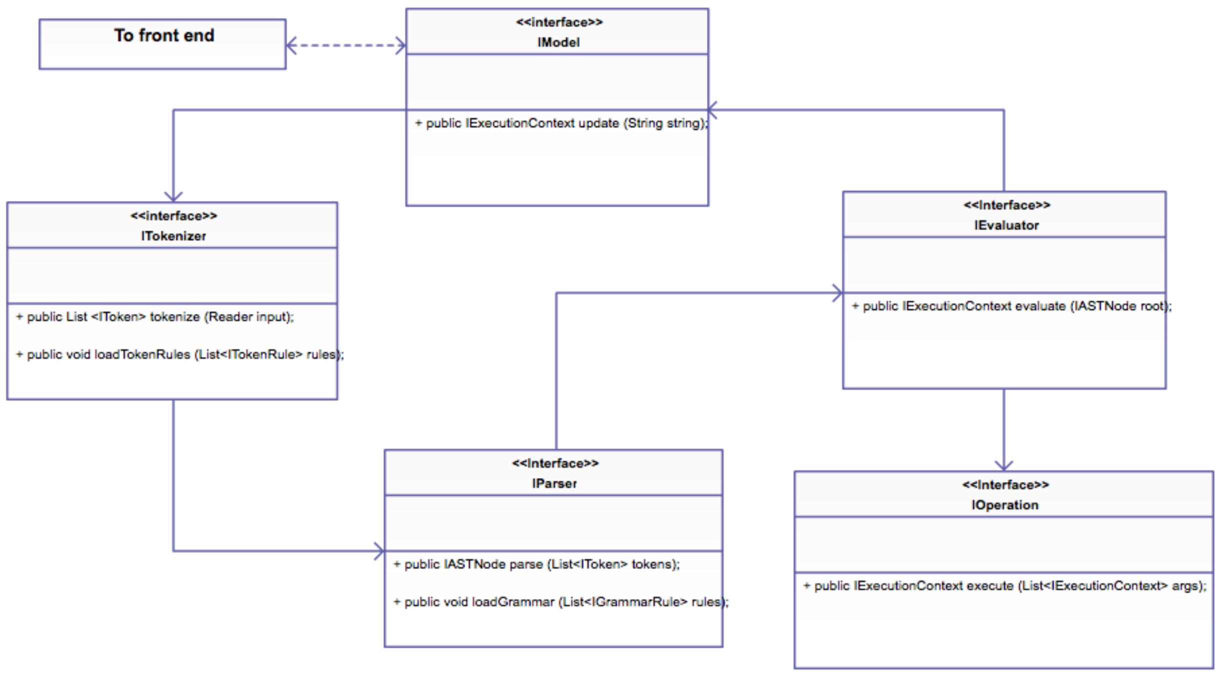
Frontend UML:



Visual Display:



Backtend UML:


To front end
<<interface>>
IModel
+ public IExecutionContext update (String string);
<<interface>>
ITokenizer
+ public List <IToken> tokenize (Reader input);
+ public void loadTokenRules (List<ITokenRule> rules);
<<Interface>>
IEvaluator
+ public IExecutionContext evaluate (IASTNode root);
<<Interface>>
IParser
+ public IASTNode parse (List<IToken> tokens);
+ public void loadGrammar (List<IGrammarRule> rules);
<<Interface>>
IOperation
+ public IExecutionContext execute (List<IExecutionContext> args);

<u>Example Code:</u>

Frontend:

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

Backend:

```
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() {
                IModel myModel = new Model();
                myModel.execute("FW 50");
                Collection <String> elements = new ArrayList<String>();
                elements.add("position");
                elements.add("turtleVisibility");
                elements.add("pen");
                Map map = myModel.getData(elements);
                assertEquals("0,50", map.get("position"));
                assertEquals("true", map.get("turtleVisibility"));
                assertEquals("up", map.get("pen"));

                }

        @Test
        public void testUIElement() {
                IUIElement turtlePane = new TurtlePane();
                Collection<String> turtlePaneDependencies = new ArrayList<String>();
                turtlePaneDependencies.add("turtleVisibility");
                turtlePaneDependencies.add("pen");
                turtlePaneDependencies.add("position");
                assertEquals(turtlePaneDependencies, turtlePane.dataDependencies());
        }

        @Test
        public void testUIElementFactory() {
                IUIElementFactory factory = new UIElementFactory();
                assertEquals(new TurtlePane(), factory.makeElement("TurtlePane");
                assertEquals(new ErrorPane(), factory.makeElement("ErrorPane");
	}

	@Test
	public void testOperation() {
		IOperation sum = new Sum();
		IExecutionContext oneOne = new Constant(1);
		IExecutionContext oneTwo = new Constant(1);
		List<IExecutionContext> args = new ArrayList<IExecutionContext>();
		args.add(oneOne);
		args.add(oneTwo);
		IExecutionContext result = sum.execute(args);
		assertEquals(result.environment.get(“returnValue”),”2”);
	}
```

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.

Another design choice we considered was what exactly was to be returned to the view. Originally, we just wanted to have a return value in the format of a number. However, we realized that this would have created a large if tree because we would have had to determine the format of that number. We ultimately decided to, instead, return an object that has a key and an output. The object would be given to a method in the view class, where the key being returned would be equated to another key in a map, which would then run another method mapped to that key. The method would take the output of the backend as an input. Both the output and the input would be of the same format, thus removing the need for a large if tree and making our code a lot more modular.

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.

Another thing we thought about was what input we wanted to give to our backend. Our first design was the simply pass in the command from the user as a string. We assumed that all the information we would need from our input could be found in the command, so we would not need to pass in anything else. But after brainstorming for several hours, we decided to not only pass in the command as a string, but to also pass in a list of parameters that helped to define the configuration of the game. Our logic behind this decision was that our commands would would have different effects on the frontend based on certain configurations, such as whether or not the path the turtle takes is being hidden, or if the grid the turtle traverses is 2D or 3D. We figured that even though a few of these parameters could have easily just been toggled using a bit of front end coding, the best way to implement additional functionalities to our code would be to just add new parameters to our list instead of adding new methods and new if statement checks to the rest of our classes. In this way, the backend is constantly aware of the current state of the front end and can react accordingly.

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.