This was the submitted design document, if we want to edit it.

Chris Bernt, Dimeji Abidoye, Safkat Islam, Mike Zhu
SLogo Design Document

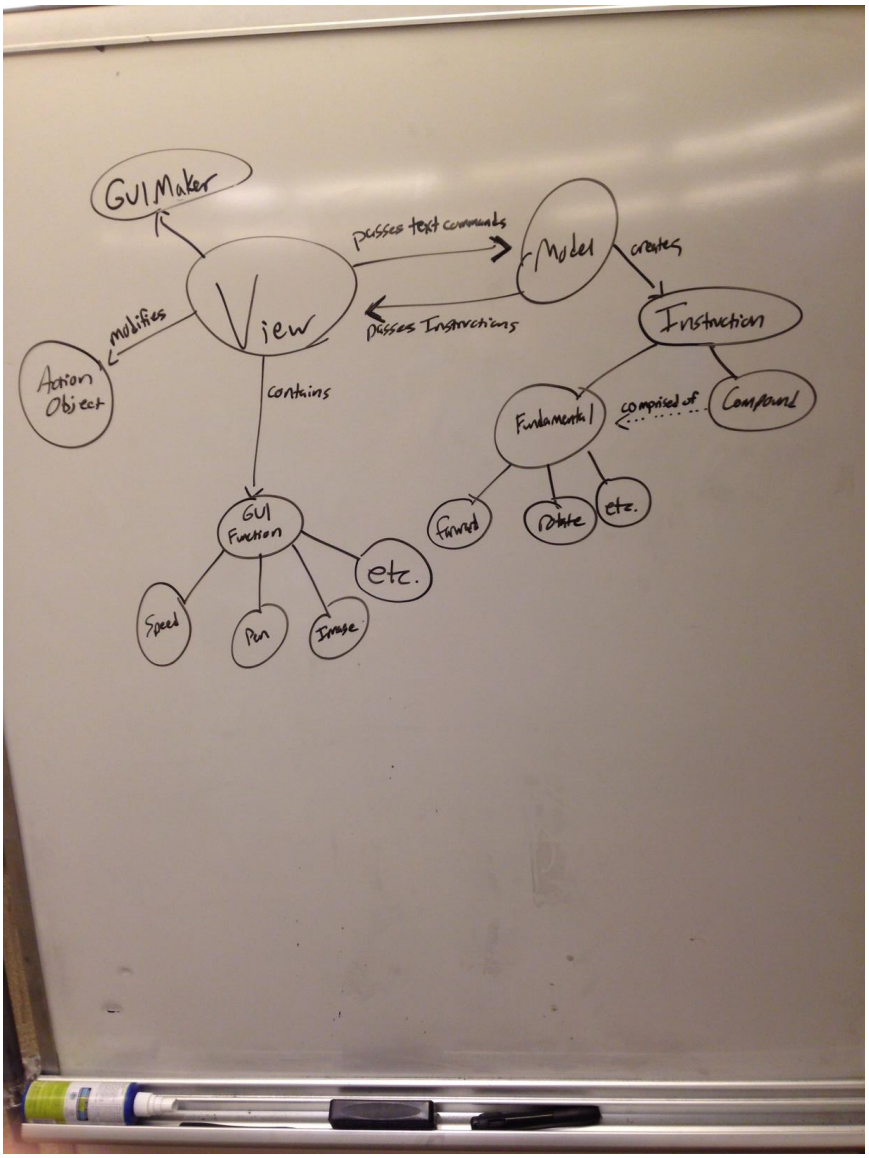
Design Goals

The primary modules of our design can be broken up into the following sections: the Model, the View, an Instructions superclass that will be divided into Fundamental Commands, Compound Commands, Boolean Commands, Query Commands, Math Commands, and Variable Commands - the purpose of this hierarchy is to keep the instructions closed to modification and even though this may blur the line between frontend and backend, this will help keep the Model and View separate in the end. We will also have an intermediary class that serves as a buffer between the View and Model and vice versa. This allows for extensibility in the future if the input from the View needs to be manipulated in some way before it is received by the Model, similar to how output commands from the Model may need to be manipulated before the View receives them. Similarly, it keeps reliance between the Model and View to a minimum so that one can be changed without affecting the other.

Primary Classes and Methods

On the backend, the abstract class Instruction acts as the superclass for all basic instructions implemented in SLogo, such as Forward, Rotate, etc. Instruction is also extended by a class CompoundInstruction that holds lists of fundamental instructions. This allows us to build complex functions out of basic building blocks (e.g., travel left implemented as the sequence of rotating 270 degree clockwise then moving forwards). The Parser class in the backend takes String inputs from the frontend representing command line inputs and converts them into lists of individual instructions that can be used to control the turtle.

The following chart represents our design at a high level:



Primary classes and methods

External backend API for backend extension

We will allow users to extend the instruction set of our program by building complex instructions from the set of fundamental instructions. Fundamental Instructions are defined as subclasses of a Fundamental Instruction, classified by the fact that they can exist on their own. Compound Instructions will also be a group of separate subclasses that are defined by a series of Fundamental Instructions. In terms of extensibility, one would only need to add a new class, either Fundamental or Compound, and the actual code that calls the action will not need to be changed. For example, a command that makes the turtle travel diagonally can be constructed from the rotate and move forwards commands.

We will grant public access to the Instruction abstract class and the basic functions, Move, Forward, and booleans.

```
/**
 * Instruction is the superclass for FundmentalInstruction and CompoundInstruction.
```

```
 *
 * FundamentalInstruction and its subclasses are private and cannot be modified.
 * There should only be a few of these (e.g., forward, rotate, penUp and penDown).
 * More complex functions (including user-defined functions) will be implemented
 * as objects of CompoundInstruction.
 *
 * Methods:
 * Constructor takes in a String code (e.g., fd, rt) and an int parameter (50, 40,
 * etc.)
 *
 * doAction() - implemented by subclasses of Instruction. doAction() calls the
 *              appropriate method in the FrontEnd using Reflection. Returns a
 *              Number object representing the parameter (e.g., fd 50 returns 50)
 *
 * @author Mike Zhu
 *
 */
public abstract class Instruction{
      public FundamentalInstruction(String code, int parameter)

      public abstract Number doAction(ActionObject obj)

      public String toString()
}

/**
 * CompoundInstruction contains a list of Instruction objects.
 *
 * The doAction() method of CompoundInstruction is implemented. When run, the method
 * goes through the object's Instruction list and runs them in sequence. Thus
 * CompoundInstructions are implemented as a series of Instructions (e.g., right 50 =
 * rotate 90, forward 50)
 *
 * Developers extend the backend by adding new subclasses of CompoundInstruction.
 * This can be done by dynamically, by defining new methods through the GUI command
 * line using SLogo syntax, or by adding new classes manually.
 *
 * @author Mike Zhu
 *
 */
public abstract class CompoundInstruction extends Instruction{

}
```

Internal Frontend-Backend API

The backend receives input from the frontend in the term of a String. This String is copied verbatim from user-inputted commands, in Slogo syntax, into the command line. The backend parses this String and returns a List of Instruction objects. Each element of this list is an object representing a given SLogo instruction (built-in or custom), with a doAction() method. The doAction() method interacts with a Turtle class in the frontend to visually represent the given Instructions. The list contains the entire set of commands input into the command line, arranged in linear order.

Internal Frontend-Backend API

The frontend then runs these instructions one by one. As an example:

Output: {PenDown object, Rotate object, Forward object}
General format: {Instruction1, Instruction2, Instruction3}

When the frontend receives these instructions, it may throw an InvalidInstructionException if the instruction received is inappropriate in the current state of the program. The backend will additionally pass information about the current state of the program. This includes a dynamic list of variables with current values and available user-defined methods. Since the variables have the potential to be updated frequently, the variable values will be continually passed every line with a command. Similarly, once the test is completely parsed, a list of all user-defined methods will be passed to the frontend to be displayed.

```
/**
 * Parser communicates between the frontend and the backend. Parser takes in a String
 * representing the command-line input from the GUI, parses it, and returns a list of
 * instructions. The frontend doesn't need to know how this parsing happens.
 *
 * @author Mike Zhu
 *
 */
public class Parser {
	/**
	 * Takes in String input from frontend, returns a list of Instruction objects
	 * to be run by the frontend sequentially. Parse should be the only method
from
	 * the backend that the frontend ever knows about.
	 *
	 * @param input == the String from command line
	 * @return == a List of Instructions
	 */
	public List<Instruction> parse (String input)
}
```

External frontend API for frontend extension

In our external API, we will have a superclass that resembles any GUI function. in this superclass, we will have the methods action() and makeTool(), which are common among all GUI features and will create the interface feature and define the action that feature does, respectively. We will also have abstract representations for each type of GUIFeature like a GUIFeatureWithButton; each concrete button will then extend this class. Every time you want to add a feature, you should simply just add a GUIFeature subclass or a subclass of a particular type of GUIFeature.

```
/**
 * This abstract class represents any sort of interaction tool that the user may have to deal with.
 * An example of this is a button or a drop down menu.
 *
 * @author Safkat Islam
 * @author Chris Bernt
 */
package frontend;
import javafx.scene.Node;
public abstract class GUIFeature {
	protected double myX;
	protected double myY;
	/**
	 * Constructor.
	 * @param x The x-coordinate of the tool on the screen.
	 * @param y The y-coordinate of the tool on the screen.
	 */
	public GUIFeature(double x, double y){
	myX = x;
	myY = y;
	}
```

```
	/**
	 * This method represents the action taken by the feature when it is
interacted upon.
	 * For example, the turtle changes color if the changeColor button is pressed.
	 */
	public abstract void action();

	/**
	 * Creates the graphical representation of the GUIFeature.
	 * @return The JavaFX Node object that is the graphical representation and
that the user interacts with.
	 */
	public abstract Node makeTool();
}

/**
 * This class represents a GUIFeature that uses a button to interact.
 *
 * @author Safkat Islam
 * @author Chris Bernt
 */
public abstract class GUIFeatureWithButton extends GUIFeature {
}

/**
 * This class represents a GUIFeature that uses a drop-down menu to interact.
 *
 * @author Safkat Islam
```

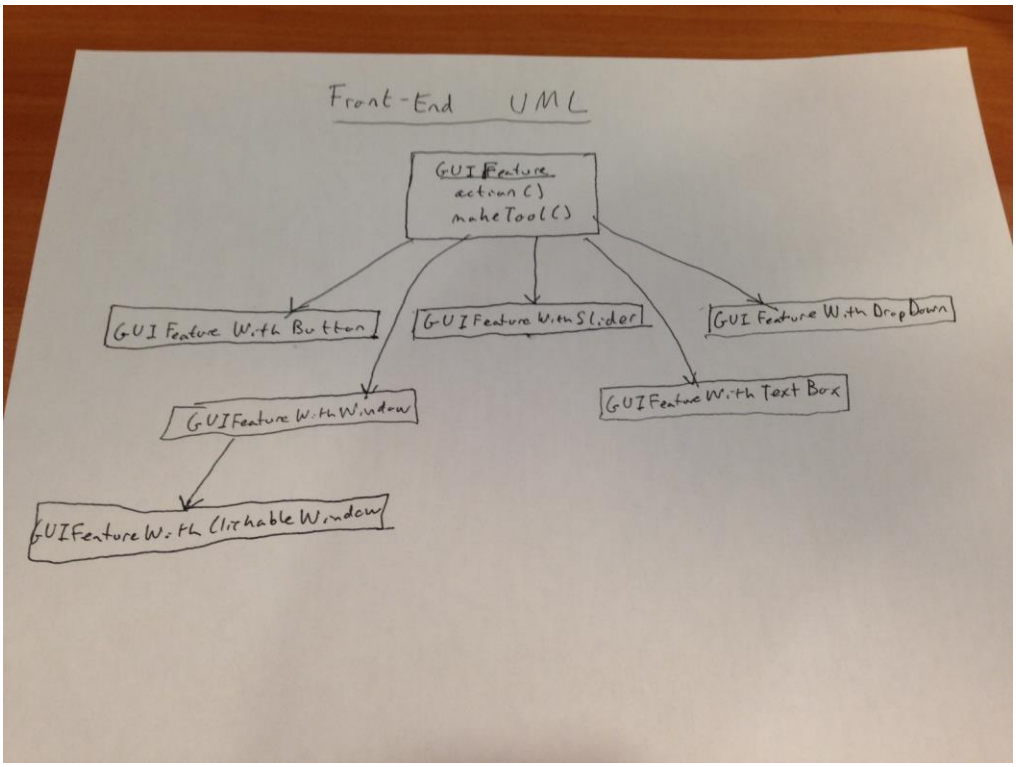
```
 * @author Chris Bernt
 */
public abstract class GUIFeatureWithDropDown extends GUIFeature {
}
/**
 * This class represents a GUIFeature that uses a slider to interact.
 *
 * @author Safkat Islam
 * @author Chris Bernt
 */
public abstract class GUIFeatureWithSlider extends GUIFeature {
}
/**
 * This class represents a GUIFeature that uses a writable text box to interact.
 *
 * @author Safkat Islam
 * @author Chris Bernt
 */
public abstract class GUIFeatureWithTextBox extends GUIFeature {
}


/**
 * This class represents a GUIFeature that uses a window to interact.
 *
 * @author Safkat Islam
 * @author Chris Bernt
 */
```

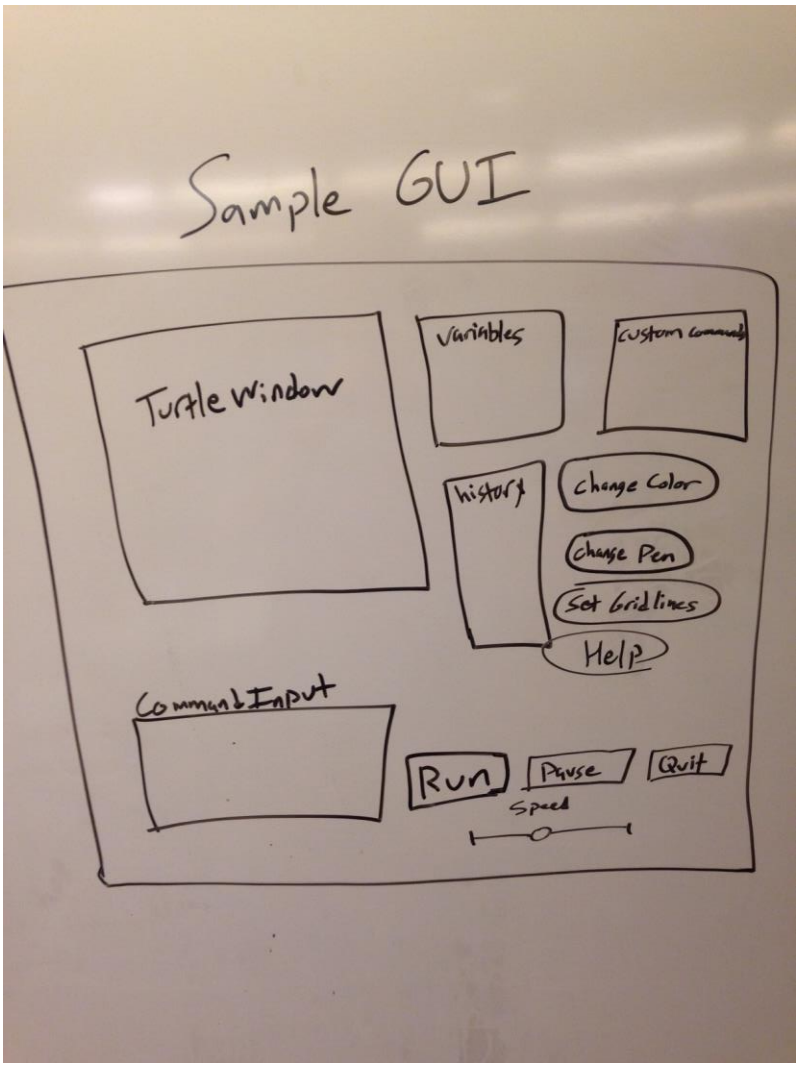
```
public abstract class GUIFeatureWithWindow extends GUIFeature {

}
/**
 * This class represents a GUIFeature that uses a clickable window to interact.
 *
 * @author Safkat Islam
 * @author Chris Bernt
 */
public abstract class GUIFeatureWithClickableWindow extends GUIFeatureWithWindow {

}
```



Sample GUI



Code Description

When a command is parsed by the backend, it is converted into a subclass of the Instruction interface, which is implemented by the abstract Fundamental and Compound Instruction. The key method in these classes is the doAction() method, which interprets the instruction for the frontend. The FundamentalInstruction class currently looks as follows:

The code structure that therefore that converts “fd 50” into turtle motion is as follows: the View takes the text and passes it to the backend parser; the parser converts it into a ForwardInstruction and passes it back to the View; the View calls doAction() on the ForwardInstruction, which results in the turtle moving and leaving a trail.

The general test plan for this API contains instantiating an action object with instructions, and checking to see whether the object continues to obey those instructions. Since they are not yet implemented, the best tests we currently have include checking whether the Instructions are not null, proper subclasses, and contain working methods. The following JUnit tests test these ideas.

JUnit Tests

The following JUnit tests check whether the Parser successfully converts String inputs into Instruction lists, and whether the TurtleModel class can successfully take parsed input and use it to move the turtle.

```
public class TestParser {

	@Test
	public void testIfForwardFiftyWorks(){
		Parser tester = new Parser();
		String input = "fd 50";
		List<String> inputList = new ArrayList<>(input.split(" "));

		List<Instruction> instructionList = tester.parse(inputList);
		assertEquals("Forward 50", instructionList.get(0).toString());
	}

	@Test
	public void testIfMultipleCommandsWork(){
		Parser tester = new Parser();
		String input = "fd 50 rt 90";
		List<String> inputList = new ArrayList<>(input.split(" "));

		List<Instruction> instructionList = tester.parse(inputList);
		assertEquals("Forward 50", instructionList.get(0).toString());
		assertEquals("Rotate 90", instructionList.get(1).toString());
	}

	public void testIfNestedCommandsWork(){
		Parser tester = new Parser();
		String input = "fd sum sum 50 50 50";
		List<String> inputList = new ArrayList<>(input.split(" "));

		List<Instruction> instructionList = tester.parse(inputList);
		assertEquals("Forward 150", instructionList.get(0).toString());
	}

	public void testIfTurtleActuallyMoves(){
		TurtleModel tester = new TurtleModel();
		TurtleModel.getParser().parse("fd 50 rt 90");
		TurtleModel.runInstructions();

		assertEquals(50, TurtleModel.getTurtle().getDistance());
		assertEquals(90, TurtleModel.getTurtle().getAngle());
	}
}
```

Alternate Designs

We initially considered storing the set of all instructions within a map variable of the Model class (the central class of the backend). This would have simplified the process of dynamically adding new instructions while the program is running: the new instruction would simply be added as a value to the instruction set mapped to the user-entered keyword. Ultimately we decided against this plan, as it required modifying source code whenever we wanted to extend the backend by adding new instructions. The final design, in which each instruction is represented by an individual class, is a far more extensible design as introducing new instruction requires adding new code, not modifying old code.

Error Handling

If there are any errors either in the back-end or in the front-end, we will throw a custom-made instruction as well as create an ErrorPopUp object that will display a new window on the screen with useful information about the error (i.e., what it actually is) and an "okay" button to close this new window.

Roles

Chris Bernt and Safkat Islam will be working on the frontend. Dimeji Abidoye and Mike Zhu will be working on the backend. For each subclass of an Instruction, the back end team will parse the text into a series of instructions that is passed to the front end. The frontend team will determine what exactly that Instruction action must be to make sense in the scope of turtle motion.