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

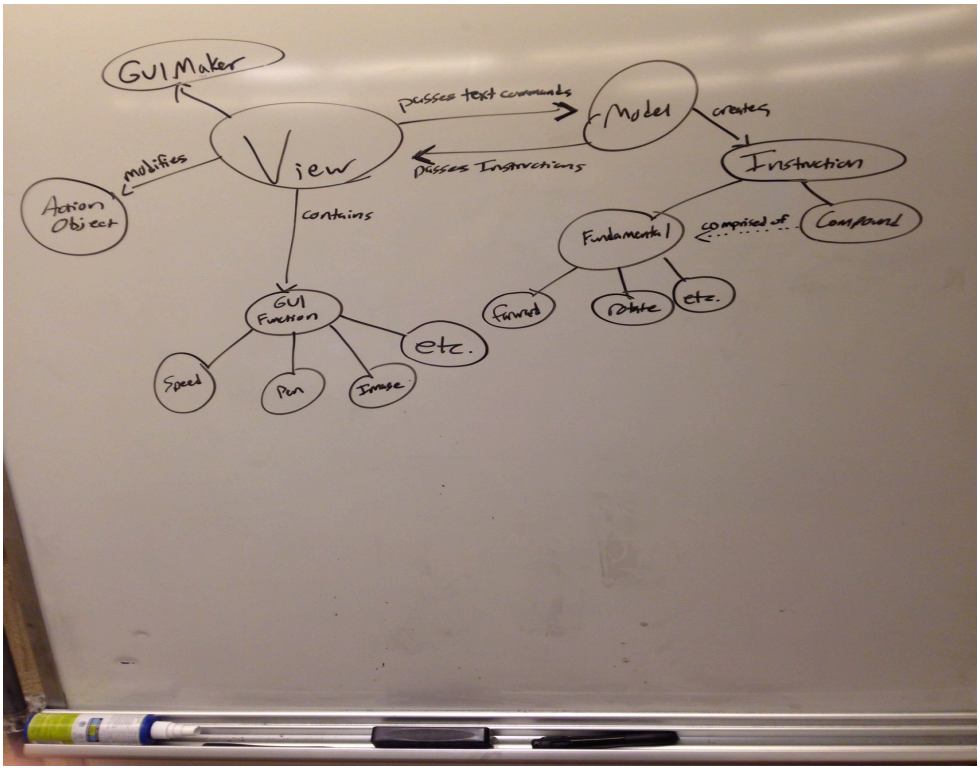
Design Goals

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:



Example Code

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:

```
import backend.Instruction;
public abstract class FundamentalInstruction extends Instruction {

	protected int myParameter;
	protected static String myCode; //Change for different languages later

	public FundamentalInstruction( int parameter, String Code){
		myParameter = parameter;
		myCode = Code;
	}


	public String toString(){
		StringBuilder toRet = new StringBuilder();
```

```
                toRet.append(myCode + " :" + myParameter);
                return toRet.toString();
        }



        public abstract Number doAction(ActionObject obj); //the crucial method

        protected int getParameters(){
                return myParameter;
        }
}
```

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.

```
public class ParserTest {

        public List<Instruction> testInstructions = new
ArrayList<Instruction>();
        public Forward forward = new Forward(0);


        @Test
        public void checkNotNull(){
                assertTrue(forward != null);
        }

        @Test
        public void checkSubclassFund(){
                assertTrue("is instance of Fundamental Instruction", forward
instanceof FundamentalInstruction);
        }

        @Test
        public void checkSubclassInstr(){
                assertTrue("is instance of Instruction", forward instanceof
Instruction);
        }

        @Test
```

```
	public void checkToString(){
		assertEquals("toString is correct", "fd :0", forward.toString());
	}

	@Test
	public void checkAction(){
		//will be better once the forward instruction is implemented
		assertEquals("doAction working", forward.doAction(), 0);
	}

}
```

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.

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.

API Design

Internal frontend-backend API:

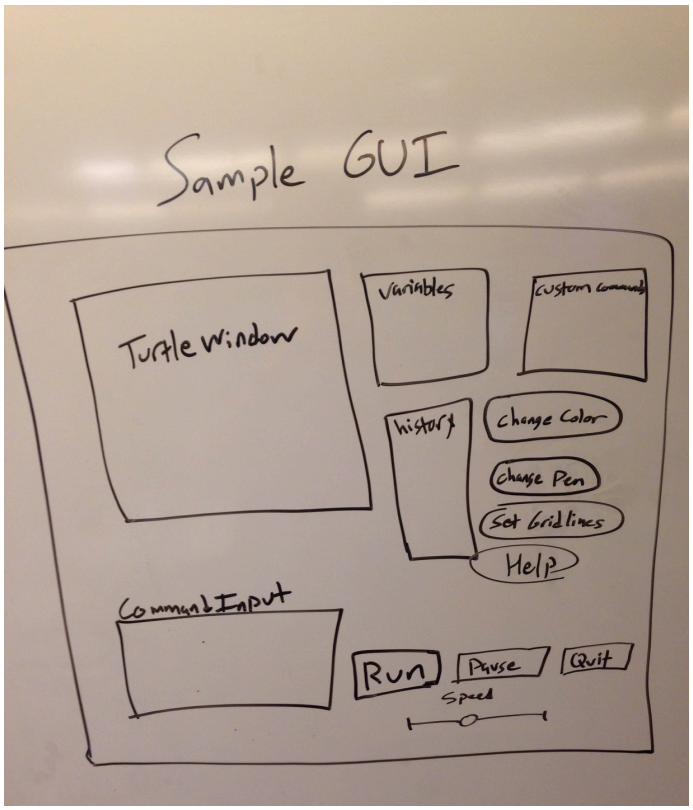
The primary object that will be passed from the frontend to the backend is the string input of instructions that a user types into a text box. When the user chooses to run the program, the text will be sent to the backend for parsing. The method that passes will throw a NoTextFoundException if the user did not actually enter any input. When the backend receives the text, the backend will throw a NoSuchInstructionException if the text represents an instruction that has not been defined.

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 methods like makeTool() and doAction(), which are common among all GUI features and will create the interface feature and define the action that feature does, respectively. Some examples of this are Pen, PenColor, BackgroundImage, BackgroundColor, BackgroundGrid, Turtle, HelpPage,

CommandHistoryWindow, VariableListWindow, etc. Every time you want to add a feature, you should simply just add a GUIFeature subclass.

The current idea for the GUI looks as follows:



Internal backend-frontend 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. 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.

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.