SLOGO Design Document

Friday October 3rd, 2014

Austin Kyker, Allan Kiplagat,

Steve Kuznetsov, Stanley Yuan

Design Goals:

The major goal of this design is to separate the front-end (the view) from the backend (the model). To do this we plan on building an MVC architecture. MVC stands for Model, View, and Controller. The Model is the data and should not contain much logic. The view is the front-end (the Graphical User Interface) and should also not contain much logic. The controller, on the other hand, has the ability to display a view, listen for events on the view (such as the user inputting a command or clicking a button), and make decisions based on these events. The controller may interact with the model to make calculations, parse information, or get and set data. For simplicity, we will also refer to the view as the Graphical User Interface (GUI). We will also alternatively refer to the model as the backend. In order to keep the GUI separate from the model we will define external API's for both of these components. These API calls will be facilitated by a controller. This way the front-end does not have direct control to the backend, and the backend does not have direct control to the front-end. By defining the clear APIs, our group can split into pairs (half working on the GUI, half on the backend) and assume that an API call to the other end will result in the correct action (as specified by the contract of the API). Internally within each end (front-end and back-end) our goal is to write closed and extensible code. Adding new features should be as easy as extending a superclass or implementing an interface and overriding a couple methods. Finally, we will pride ourselves on writing clean, concise code that makes use of the standard java naming conventions.

Primary Classes and Methods:

Due to our primary goal of this project being a separation of the front-end and back-end, we will introduce the primary classes and methods of each of the MVC components separately. We will begin with an examination of the controller.

Control

The control class will be called SlogoControl. This class will implement two interfaces. The first interface will be SlogoGraphics. This interface will require the implementation of methods that will be called when the user performs an action (Types a command and hits return or clicks a button). Some of these methods will include setBackgroundColor(Color c), toggleRelativeGrid(), parseCommandFromString(String command), etc. Some of these methods will not require interaction with the backend. For instance, when the user wants to change the color of the background, we do not need any of the parsing capabilities of the backend to do this. In this case, the SlogoControl implementation of this method will simply make a call to a ComponentDrawer class (a front-end class). In contrast, a method like parseCommandFromString() will result in the control making an API call to the backend.

The second interface that SlogoControl will implement is a backend interface, SlogoBackend. At first thought, this may seem unnecessary. However, our goal is for the backend to not have direct access to the front-end. Thus, this interface will define the method calls the backend can make to the control. A

good example is the method drawDrawableObjects(Queue<DrawableObject>). After the backend has parsed a String into a sequence of DrawableObjects (we will talk about these later in the backend-section), the backend wants to deliver these objects to the front-end to be displayed as shapes to the user so it will call this method.

Now, why two interfaces? The reasoning is simple. The front-end should not be able to make the method calls defined in the Backend interface, while the back-end should not be able to make the calls defined in the GUI interface. This makes sense - there is no reason that the backend should have access to the setBackgroundColor() method. Likewise, the front-end should not be able to have access to the drawDrawableObjects(Queue<DrawableObject>) method. The way this is done is that the GUI's reference to the SlogoControl class is casted to a SlogoGraphics type (the GUI interface) while the model's reference to the SlogoControl class is casted to SlogoBackend (the backend interface).

Now that we have a firm understanding of the purpose as well as the primary classes and methods of the control, we will dive into the Graphical User Interface.

GUI

The Graphical User Interface is tasked with displaying information to the user. In the case of SLOGO, the user will be able to see a turtle on a grid. The user will be able to input commands and write small programs into a command box to make this turtle act. The movements and actions of the turtle will be displayed on the graph as lines. Besides this grid, the user will be able to see a list of the previously used commands, the state of the workspace (variable information), saved commands and functions, and several other buttons allowing the user to interact with the grid and the workspace. To understand the primary classes and methods of the front-end, a class-diagram is shown in Figure 1 below.

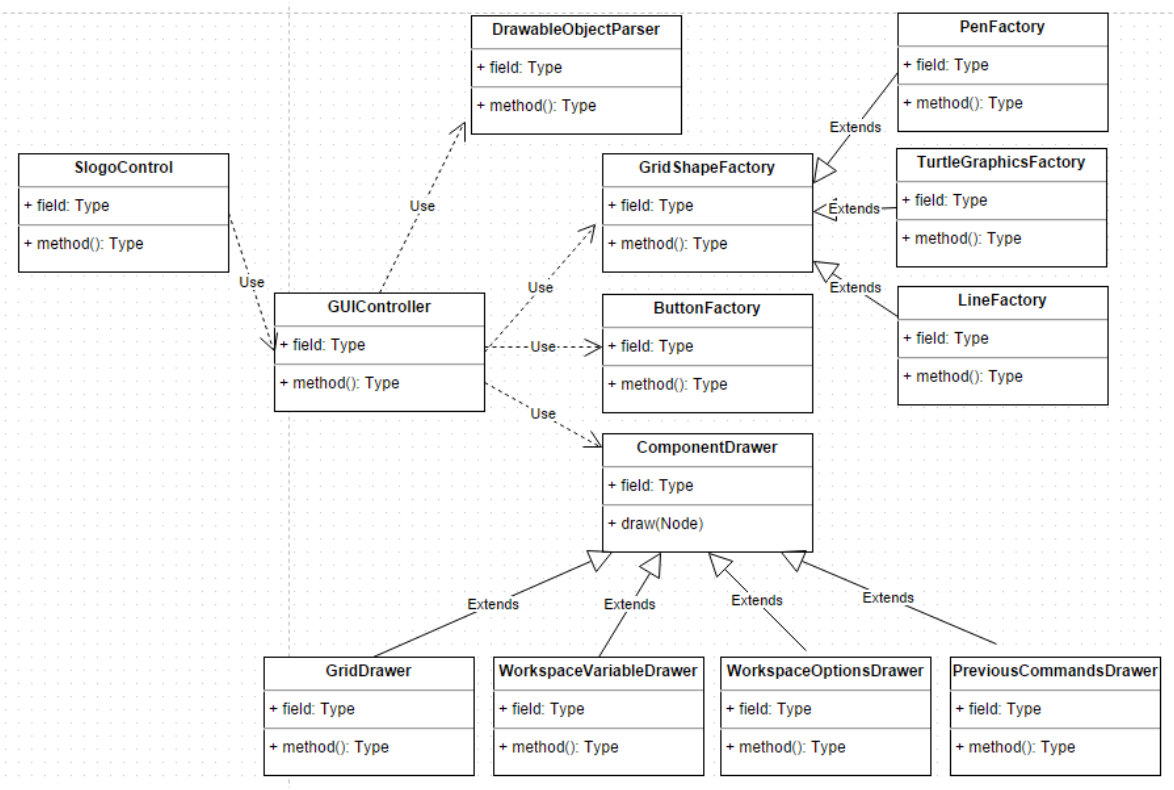

Figure 1: Class Diagram of the Front-End

Top Left is SlogoControl, the main unit of control discussed in the previous section. The GUIController sits at the top of the GUI and will have a SlogoGraphics casted SlogoControl reference. Therefore, it will be able to make the SlogoControl graphics-related method calls. Perhaps the most important method of the GUIController will be its drawDrawableObjects(Queue<DrawableObjects>) method. The queue parameter of this method contains DrawableObjects (we will discuss these in detail in the backend-section, but they are basically information on how to draw something). The GUIController has access to a DrawableObjectParser which can determine which type of shape should be drawn to represent each DrawableObject instance as well as where this shape should be drawn (which component).

To the right of the GUIController is ShapeFactory, This is an abstract class with an abstract method generateShape that's purpose is to create a shape given a parameters map. There will be several different ShapeFactories, all of which will be responsible for creating some type of shape or image to be ultimately displayed on screen. This generateShape method will take a map of parameters that it will use to determine the various properties of the shape produced such as its color, x location, y location, etc.

Once the GUIController finds out which shape to make (using the methods of the DrawableObjectParser), it will call generateShape on that particular factory along with a list of parsed parameters provided by the DrawableObjectParser that the ShapeFactory will need.

The final step will be determining what part of the Graphical User Interface is being updated. A general sketch of what the GUI will look like is presented below in figure 2:

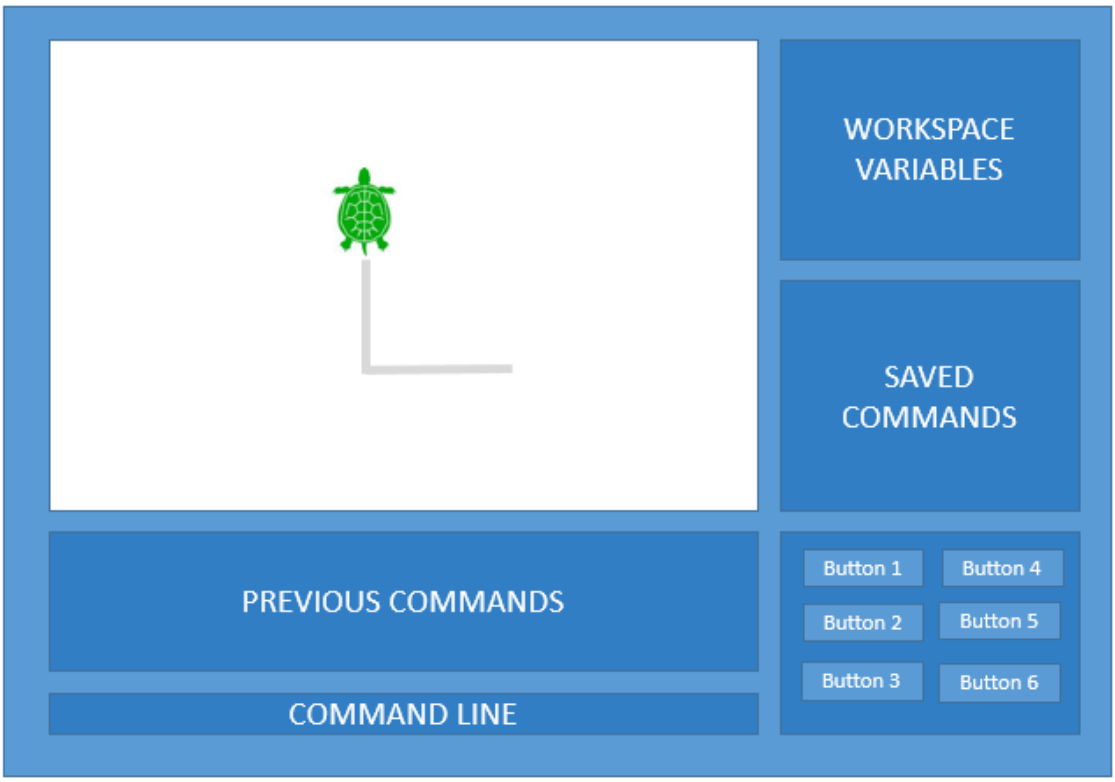

Figure 2: Initial GUI Sketch

Instead of having one GraphicsDrawer class that holds one large JavaFX group and is responsible for drawing and maintaining all of the shapes on the screen, there will be a ComponentDrawer hierarchy. There are at least five or six different components to the GUI including the grid itself, the previous commands, the saved commands, the workspace variables, and the options component (or button holder). Drawer classes will be created for each of the components. These classes will extend the ComponentDrawer class as they will share some basic functionality. At startup, an instance of each of these component classes will be created. Each of these classes will have a JavaFX group instance variable of the nodes that are displayed in their respective components. They will likely each implement an addShape method and removeShape method to simply add or remove shapes from their groups.

The DrawableObjectParser will not only be able to decipher from a DrawableObject instance which type of shape should be created, but also will be able to determine which component the shape will added to. Thus, the GUIController will use the appropriate shape factory to generate the shape and then pass this shape to the appropriate componentDrawer so that it can be updated on the screen.

As is evident from the intitial GUI sketch, our front-end will have to be able to handle a lot of different types of user interactions. In order to do this, buttons will need to be created in the GUIController. This process will be facilitated by the ButtonFactory class. These buttons will have event listeners that when

fired will use the SlogoControl reference to make control API calls. For instance, when the toggle grid button is pressed by the user, its event that will fire will be control.toggleGridButton(). Alternatively, the command line node will have to have an “enter-hotkey” event so that when the user types a command and hits enter control.parseCommandFromString(String command) will be called.

This concludes the primary class and methods section of the front-end. We will now take a look at the parsing capabilities of the backend.

Backend (Model)

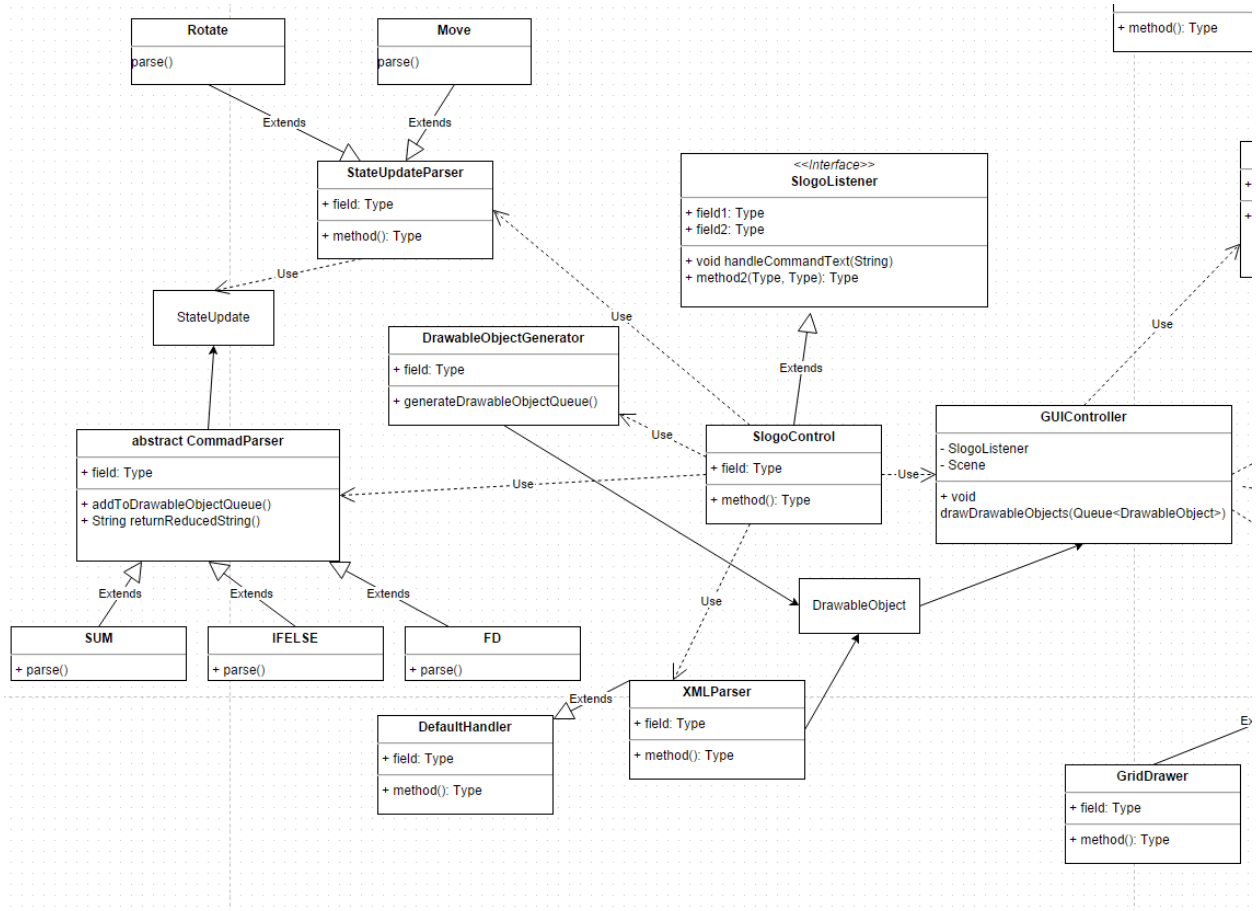

Figure 3: Class Diagram of the Front-End

The back-end of this project is divided into two major sections: the controller and the model. The general work-flow for this section of the project is as such: the GUI/view takes input from the user and relays it to the controller. For some user input, the controller is able to handle everything alone and will do so. For other input, the controller needs more information and asks the backend to parse the input and return a queue of stateUpdates to the state of the program. As the controller parses this queue of state updates, it generates drawableObjects that it will pass to the GUI by determining which of the

stateUpdates result in visible changes and calculating the absolute coordinates, color, etc. of these visible updates and packaging that into drawableObjects.

The model will mainly consist of a large hierarchy of classes implementing a CommandParser strategy - there will exist classes like fd, rt, lt, etc. that each implement their specific rules for parsing a string of commands in order to create stateUpdate objects in response to user input. The parsing of a string will be done by iterating over the command string input using a while loop inside of which reflection will be used to generate the correct parsing strategy; recursion will be used to handle nested commands. For loops and the like will be represented with a set of classes that hold stateUpdate objects The result of parsing a command string will be a queue of stateUpdate objects that will be passed to the SLOGOController.

The controller will mainly consist of a smaller but similar hierarchy of classes implementing a StateUpdateParser strategy - there will exist classes like ForLoop, Rotate, Move, etc. that each implement specific rules for updating the state of the controller using the input queue of stateUpdate objects. Parsing the stateUpdate objects will allow the controller to update the state it holds and also the appropriate nodes in the GUI; when new objects are needed to be generated as a result of state update (e.g. a line being drawn), a DrawableObject will be created and placed in the queue en route to the GUI.

API CONTRACTS

Internal GUI API
GUIController Class
public void drawDrawableObjects(Queue<DrawableObject>)

This method takes as input a queue of DrawableObjects, parses the DrawableObjects (particularly the information needed to draw the objects), and represents them in the View.

public void setListeners(Object listener)

This method takes as input an Object which requests to be added to the list of Listeners in the GUI that are notified of changes as they occur.

External GUI API
abstract GridShapeFactory Class
public abstract void generateShape(ShapeParameters)

Extenders of this class should be able to generate a shape to be displayed based on the ShapeParameters object.

abstract ComponentDrawer Class
public abstract void draw(Node)

Extenders of this class should be able to add the passed Node to their Scene root handles.

Internal SLOGOController API
SLogoGraphics Interface
The control class will implement a SLogoGraphics listener interface.

This interface will define the methods that the GUI requires to be implemented in order to function:

public Queue<DrawableObject> parseCommandString(String commands)

This method is used when the user inputs a string of commands to be executed. When the user types a command, and presses enter, a listener will be fired that makes this method call to the backend with the inputted String. The controller expects the backend to parse this string and return a queue of DrawableObject instances that the GUI can work with.

public DrawableObject editVariable(javafx.node node)

The controller will make this method call to update a variable in the Model.

public void saveCommandsToFunction(String commands)

The control passes a String of commands to be saved for future use to the Model which will create a Queue<DrawableObject> inside of the SLOGOController for storage.

public Queue<DrawableObject> runCommand(String commandName)

Returns the commands represented by the String commandName.

Internal Model API

SLogoBackEnd Interface

public Queue<DrawableObject> parseCommandString(String commands)

This method takes as input a String representing commands delineated with spaces. This method parses the string into DrawableObjects, which hold information like: String parentNode, String objectType, Map<String,String> parameters. The DrawableObjects will be placed in a queue in the order of their actions. This queue will be returned to the View.

public DrawableObject editVariable(javafx.node node)

Updates internal variable representation in the Model. Returns DrawableObject reflecting the new variable information to the View.

public Queue<DrawableObject> saveCommandsToFunction(String commands)

This method simply calls parseCommandString(String commands) to return a queue holding the DrawableObject

External Model API

abstract CommandParser Class

Extenders of this class will be instantiated by reflection to handle a particular command and once done, will call the addToDrawableObjectQueue() method, passing an appropriate DrawableObject to be eventually represented on the GUI.

addToDrawableObjectQueue(DrawableObject)

Example Code

Use Case:

The user types 'fd 50' in the command window, and sees the turtle move in the display window leaving a trail.

When the user enters the command and presses enter, the *parseCommandFromString(String command)* method in SLogoControl is called with the command as the argument. *generateDrawableObjectQueue()* is then called in DrawableObjectGenerator, which returns a DrawableObject queue generated using *CommandParser* subclasses (*FD* class in particular). This drawable objects queue is then passed to the GUIController *drawDrawableObjects()* method, which selects the appropriate GridShapeFactory subclass (LineFactory in this case) that finally animates the turtle forward 50 pixels.

JUnit Tests:

```
@Test
checkCommandParsing() {
SLogoControl  control = new SLogoControl();
String command = “fd 50 rt 20 fd 30”;
Queue<DrawableObject> createdQueue = control.parseCommandFromString(command);
assertEquals(3,createdQueue.size());
}

@Test
checkSetBackGroundColor() {
SLogoControl  control = new SLogoControl();
GUIController gui = new GUIController();
control.setGUI(gui);
control.setBackGroundColor(Color.BLUE);
assertEquals(gui.getScene().getFill(),Color.BLUE);
}
```

Alternate Design

For reasons mentioned already, we decided to go with the MVC pattern. We never really considered any other kind of broad design pattern, so instead this section will simply describe the variations within MVC that we decided to consider. Our initial conception of things had the Controller being essentially the only active part, while both the Model and View were simply passive modules that were manipulated by the Controller. We decided against this approach, as we realized that this would both lead to a bloated Controller module while also reducing the extendability and encapsulation of the code. Now, the Model and View both contain active parts, and the Controller's role has been reduced to a reasonable size.

Another design decision that we made relatively early on was the location of storage of variables. Initially, we thought that the variables would have to be stored in the model because that's where all the calculations were taking place. We then realized that it wasn't necessary - we could simply pass along the information through the controller. again achieves the similar goal of keeping the Model and View separate and encapsulated.

Roles

Austin Kyker and Allan Kiplagat will work together on the graphical user interface. They will be responsible for implementing the ComponentDrawer and ShapeFactory inheritance hierarchies as well as the GUIController class that utilizes these classes. They will also be responsible for implementing the DrawableObjectParser class that parses a queue of DrawableObject and provides the necessary information to the GUIController to call the right factory to generate the appropriate shape and then deliver this shape to the appropriate ComponentDrawer.

Steven Kuznetsov and Stanley Yuan - Will develop an XML parser to create DrawableObjects from an XML defining default state of front end. Will develop CommandParser strategy to parse commands. Will support multiple languages using this strategy. Will develop SLOGOController class to mediate between GUI and model and turn relative output of Model into absolute input to GUI.