SLogo Basic Implementation Document

Abhishek Balakrishnan, Will Chang, Tanaka Jimha, Jonathan Tseng

Design Goals: (high-level description of the project's "vision")
The primary goal of our SLogo design is to create an intuitive and extendable Logo API through implementing the Model-View-Controller (MVC) paradigm in unison with the Observer pattern. The idea of Logo is to provide an interface from which users may create their own programs to draw lines, shapes, and designs. To allow this functionality, we separate our program into three modules–the Model, ViewController, and the Interpreter–which together execute a *read-eval-print* loop. We combine the traditionally separate View and Controller into our ViewController user interface module to represent the *read* aspect of the loop. The Interpreter module then serves as the *eval* step, applying any logic necessary, represented by our SLogoExpressions hierarchy, to transform user input into usable states. The Model completes the loop, saving the state and updating the ViewController, performing the *print* step and allowing future user inputs. Furthermore, the Model serves as the Observable module, saving states and updating our list of Observers in the ViewController. In designing it thus, we will close to modification the order of method call within this loop and the message types passed between the modules, however leave open to modification the types of SLogoExpressions users might want to add to SLogo. Further, as an Observable, the Model will be extendable to additional graphical features. We assume that users of our API desire the use of our Interpreter as a Black Box to parse input (from our ViewController) into user defined state changes. Further, we assume that users may want to customize our ViewController, as well as our *read-eval-print* loop for their own applications.

Modules:
- View/Controller/UI
  - Scene
  - Inputs
  - Observer
- Interpreter
  - Expressions
  - Logic
- Model
  - SLogo States
  - Observable

Primary Classes and Methods: (more detail with actual complete method signatures)

The full list of method stubs can be found in the api branch of our GitHub repository.

The program will be divided into 3 main modules (the Main Module only serves to initialize the modules and start the program, and will probably be less than 50 lines of code): the ViewController module, the Model module, and the Interpreter module. The ViewController will handle updating the view and giving functionality to the buttons and input fields of the UI. It will be an observer of the Model. The Model will hold all of the state of the program. This will include things like the list of transitions that have occurred to the turtle from the SLogo function calls and the position and properties of the turtle (position, rotation, visibility, pen down/up, etc.). Finally the Interpreter will work to understand and evaluate the user-inputted SLogo commands and pass this info onto the model. Overall, the flow of code will be thus; the user will see the View supplied by the ViewController and can input SLogo commands. The input string will then be passed to the Model which will pass it to the Interpreter for evaluation. This will update the Model, and because the ViewController is an observer of the Model, the View will in turn be updated as well.

Example Code: (this is especially important in helping others understand how to use your API)

Example JUnit tests can be found in the test package of the project on the api branch of our GitHub repository.

Sequence of code:
If a user types "fd 50" in the command window, the string "fd 50" will be passed by the MainViewController to the MainModel. The model will interpret this string by passing the command to its instance of the Interpreter. The interpreter will evaluate the expression and create a SLogoExpressionResult instance which is a wrapper for the result of the expression (i.e., whether the expression compiled, what error message there is, how to update the view, and what value is returned by the expression). This result will be passed back to the model, which will then update the View (through the Observer pattern). This is a single cycle of user input--model update--view update.

CommandWindow:

```
final TextField textBox = new TextField();
textBox.setPromptText(">");
textBox.setOnKeyPressed(new EventHandler<KeyEvent>() {
   public void handle(KeyEvent key) {
       sendCommandToModel(key.getText());
   }
});
```

Model:

```
myInterpreter.parseSLogoExpression(string commandText)
myViewController.update();
```

Interpreter:

```
mySLogoExpression = mySLogoExpressionFactory.createSLogoExpression();
mySLogoExpressionResult = mySLogoExpression.evaluate();
sendResultToViewController(mySLogoExpressionResult);
```

Alternate Designs:

In our chosen design, we decided to combine the traditionally separate View and Controller into our 'ViewController' user interface module to represent the *read* aspect of the read-eval-print loop. The alternate design would have bee the usual MVA approach of having the view and controller separate and treating them as separate entities, instead of having them encapsulated as one component. However, we instead decided to have 3 distinct modules - View Controller, Model and Interpreter, we chose this in order to keep the reading, evaluating and printing parts separate. The alternate design would have had reading the input as 2 parts, and would have made our separation of modules less uniform so and blurred the division lines between modules The alternate design would have consisted of having both inputting commands, and updating turtle as different components such that both type of functionality would be under different parts of the program. This would have included the command line, input buttons and options, the view of the turtle, and the view of previous commands. However, as the functionality would got lengthier and more complex, this may have resulted in a scattered design. So in order to prevent this and keep our design clear, simple and distinct, we joined the view and controller and utilised the shared properties and similar functionality between them.

Roles:

- View-Controller
  - Controller components - Abhishek and Jonathan
    - This involves setting up the menu bar and buttons in the view.
  - Command Window, History Window, User-Defined Command Window - Abhishek
    - This involves setting up the
  - Grid/Pane & Turtle, Status Window - Jonathan
    - This will involve displaying the turtle on the screen, dealing with the styling requirements, and extending the view to other desired functions.
- Model - Tanaka
  - This will involve creating the transition states and communicating updates back to the ViewController.
- Interpreter - Will Chang will lead this, all of us will contribute
  - This involves setting up the parser, expression factory, and expression interface.

API: What is open to the BackEnd from the FrontEnd

```
public class MainViewController implements Observable{
    public void update(MainModel);
}
```

API: What is open to the FrontEnd from the BackEnd

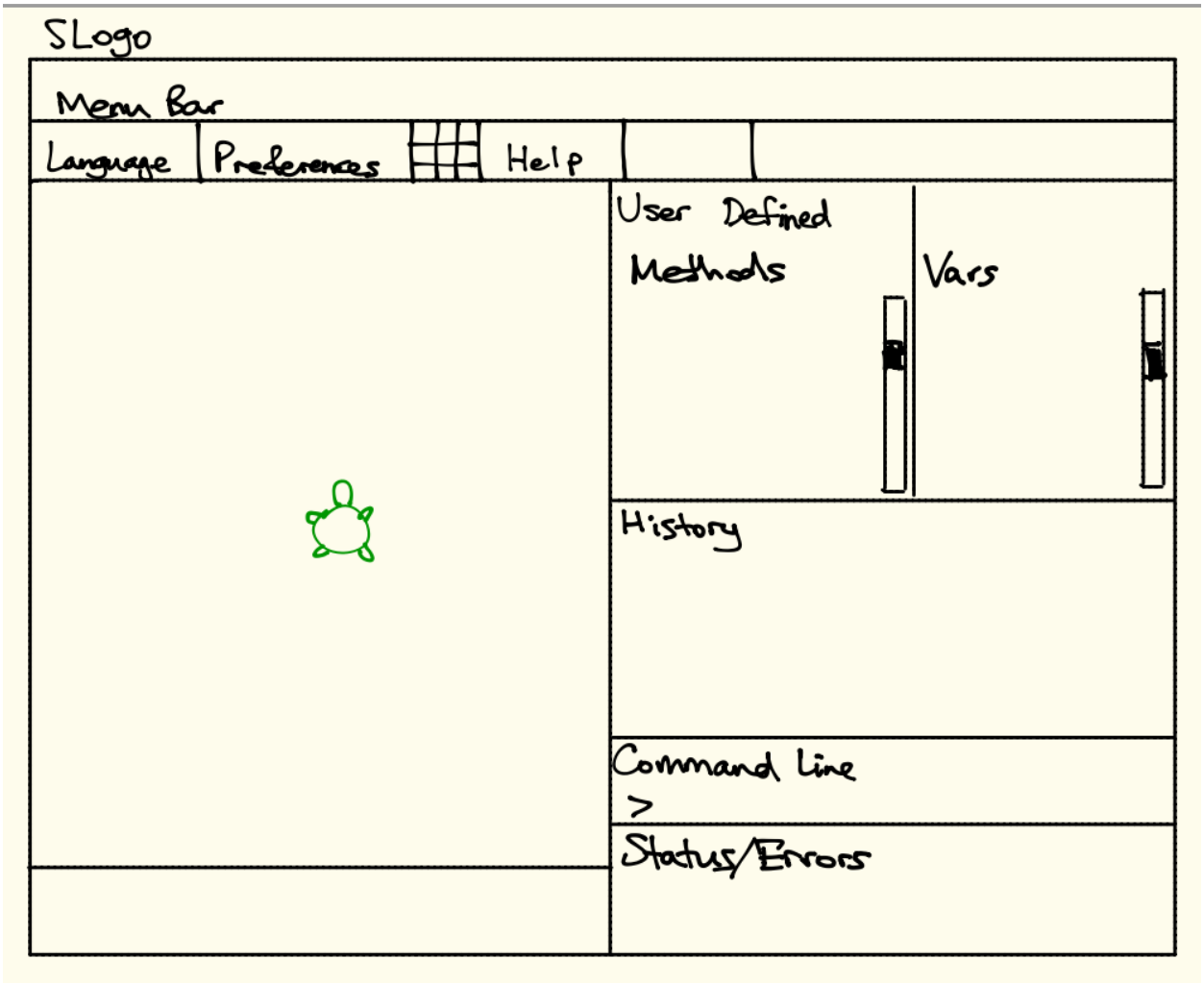
```
public class MainModel {
    public void interpretSLogo(String sLogo);
    public void setLanguage(String language);
    public void updateTurtle(TurtleUpdate properties);
    public void attachObserver(Observer observer);
    public void updateObservers();
}
```

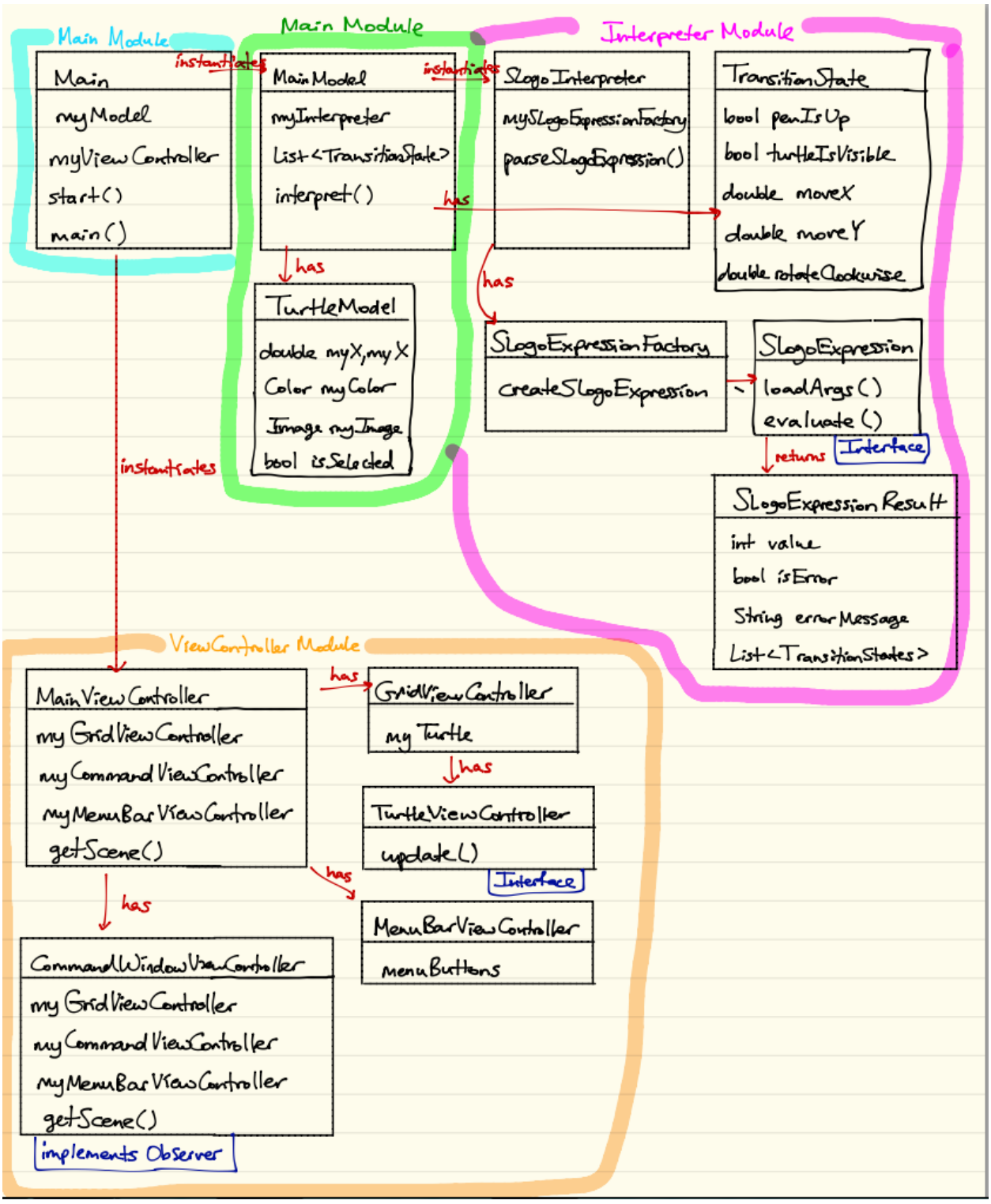
API: What is open to the FrontEnd from the FrontEnd

```
public interface ViewController {
    public Node getNode();
}
public class MainViewController implements ViewController;
public class GridViewController implements Observer implements ViewController;
public class TurtleViewController implements Observer implements ViewController;
public class CommandWindowViewController implements Observer implements ViewController;
public class MenuBarViewController implements ViewController;
```

API: What is open to the BackEnd from the BackEnd

```
public class MainModel implements Observable;
public class Turtle implements Observable {
    updateTurtleModel()
}
public class Interpreter {
    public SLogoExpressionResult parseSLogoExpression(String sLogo);
}
public abstract interface SLogoExpression {
    public abstract SLogoExpressionResult evaluate (List<String> arguments)throws invalidExpressionException
    (SLogoExpression e1, SLogoExpression e2);
}
public class SLogoExpressionFactory {
    public SLogoExpression createSLogoExpression(String sLogo);
}
public class SLogoExpressionResult {
    TransitionState;
    String getErrorMessage();
    boolean hasError();
    int getValue();
}
```


SLogo
Menu Bar
Language
Preferences
Help
User Defined
Methods
Vars
History
Command Line
>
Status/Errors


Main Module
Main
myModel
myViewController
start()
main()
instantiates
Main Module
MainModel
myInterpreter
List<TransitionState>
interpret()
instantiates
has
TurtleModel
double myX, myY
Color myColor
Image myImage
bool isSelected
instantiates
Interpreter Module
SlogoInterpreter
mySLogoExpressionFactory
parseSLogoExpression()
has
TransitionState
bool penIsUp
bool turtleIsVisible
double moveX
double moveY
double rotateClockwise
has
SLogoExpressionFactory
createSLogoExpression
SLogoExpression
loadArgs()
evaluate()
returns
Interface
SLogoExpressionResult
int value
bool isError
String errorMessage
List<TransitionStates>
ViewController Module
MainViewController
myGridViewController
myCommandViewController
myMenuBarViewController
getScene()
has
GridViewController
myTurtle
has
TurtleViewController
update()
Interface
has
MenuBarViewController
menuButtons
has
CommandWindowViewController
myGridViewController
myCommandViewController
myMenuBarViewController
getScene()
implements Observer