PROJECT ARTIFACT

**Introduction**

*This section describes the problem your team is trying to solve by writing this program (including extra credit variants you are interested in working on), the primary design goals of the project, and the primary architecture of the design. It should be approximately two or three paragraphs and discuss the program at a high-level (i.e., without referencing specific classes, data structures, or code).*

SLogo is an interpreter for a simplified version of the Logo programming language. In order to accommodate a flexible command base, our system utilizes a class hierarchy for commands such that new commands merely need to extend the super instruction skeleton. The view operates the same way in that there exists base requirements for what a view must have, but otherwise most major decisions can be implemented at will by any future developers. Extensibility was the main design goal for the view and the model in that we wanted it to be very easy to add more commands or to use a completely different looking view with the same foundation.

To execute a command, the user first enters the desired instruction into the view. The view then passes this along to the controller, which parses the raw text into an executable instruction. This instruction is then executed on the model, where it updates lines and images on the screen and returns an integer value to the view to display in the command history. Of course, if the raw text entered by the user does not conform to the correct syntax for a known instruction, no instruction will be executed as the parser will throw an exception that is caught in the controller and an error message will be displayed in the command history.

**Overview**

*This section serves as a map of your design for other programmers to gain a general understanding of how and why the program was divided up, and how the individual parts work together to provide the desired functionality. As such, it should describe specific modules in your project, their purpose with regards to the program's functionality, and how they collaborate with each other. A module may represent a single class or a group of classes related in some standard way, such as set of subclasses. It should also include a picture of how the modules are related (these pictures can be hand drawn and scanned in, created with a standard drawing program, or screen shots from a UML design program). This section should be approximately two pages and discuss specific classes, methods, and data structures, but not individual lines of code*.

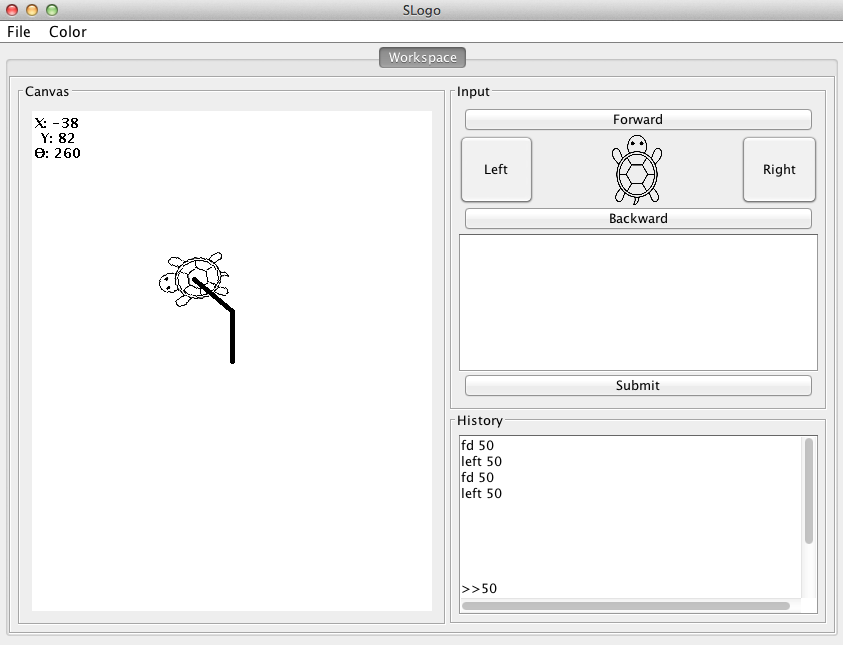
The view and the underlying simulation are divided such that our framework can implement any view a end user might desire (as long as it follows the requirements outlined in the view superclass). The SLogoview class is our specific implementation.

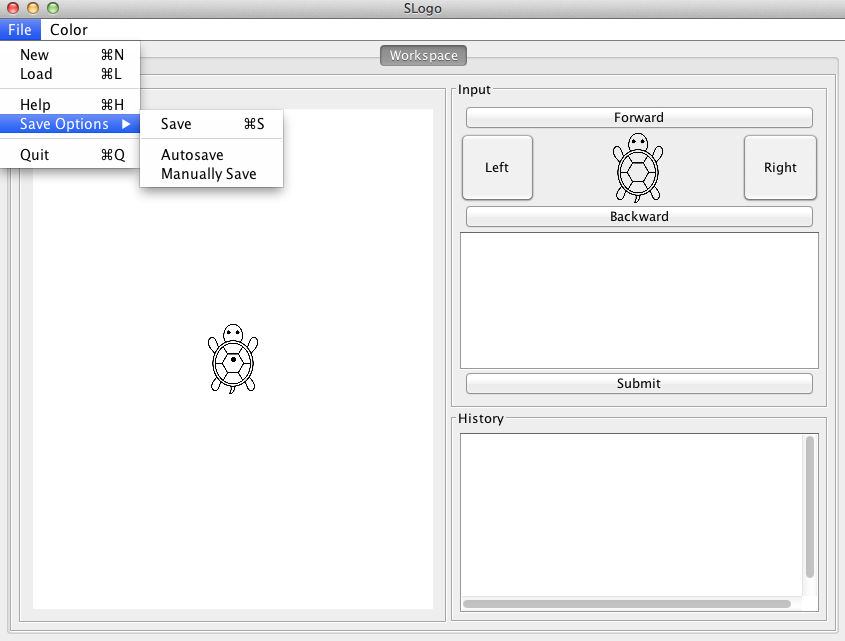
The model side of the program is split into two components: Control and Model. The responsibility of Control is to receive communication from the View (e.g. do an instruction, save or load state) and do processing of that action, and then manipulate the Model accordingly. There are several classes in the Control that help to complete that function. One is the Parser which takes raw text and generates an executable Instruction by calling the method generateInstruction(). There is also an Environment class which contains InstructionMaps of all the instructions and variables that are available to be used by the Parser, and which also contains a Palette is just a container class for Maps of different graphical options such as turtle images, background images, pen colors, and line styles.

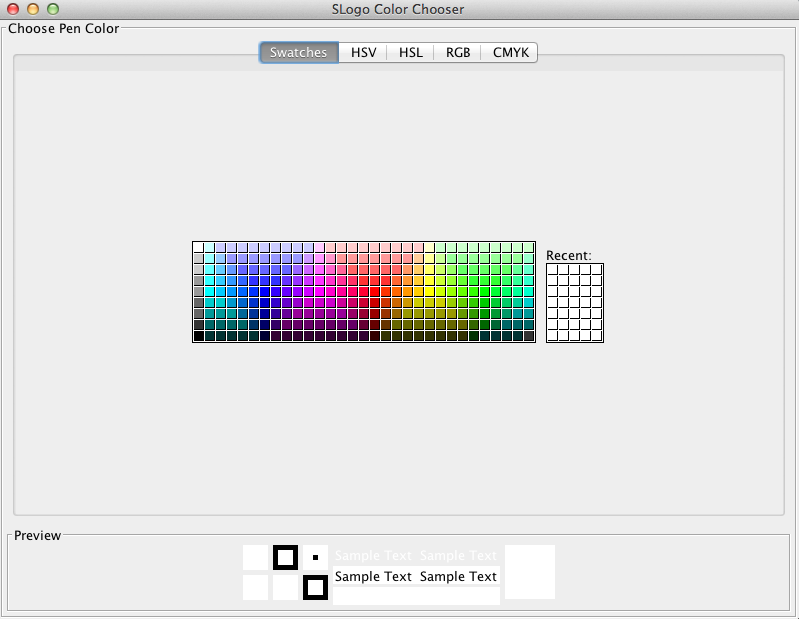
The Model component has the responsibility of holding state of the simulation. This includes a Turtle which has the ability to be moved around and draw lines with a Pen object that it has. There is also storage of all the lines on the screen which are a collection of individual Points, and storage of the background images on the Canvas.

See the provided UML diagrams to see the interaction of these components.

**User Interface Design (as needed)**







To enter commands for the SLogo simulation, the user merely needs to enter desired instructions in the textbox located under the “Input” region above the “submit” button. Multiple lines can be input at once. Once the commands are written, the user merely needs to press the “submit” button. The commands entered, and the last return value will be displayed in the history console. To repeat a command, or set of commands, click on their presence in the history console. To copy paste from the history, groups of continuous commands, only, can be highlighted and copied at once. To interact directly with the simulation the user can press either the forward, back, left, or right, buttons to alter the turtle according to default values.

If the user wishes to alter the pen color, an easy to use color chooser is included. Simply navigate to Color->Change Pen Color to open the color selector. Select your color and the color will change on selection. Minimize or close the window as desired. If you have any questions about what SLogo commands can be entered, navigate to File->help and an html document with implemented instructions will be displayed in your default web browser.

Any errors encountered will be output in the History console.

**Design Details**

*This section describes each module introduced in the Overview in detail (as well as any other sub-modules that may be needed but are not significant to include in a high-level description of the program). It should describe how each module handles specific requirements given in the assignment handout, what resources it might use, how it collaborates with other modules, and how each could be extended to include additional requirements (from the assignment handout or discussed by your team). Finally, justify the decision to create each module with respect to the design's key goals, principles, and abstractions. This section may be as long as it needs to be*

*and go into as much detail as necessary to cover all your team wants to say.*

View:

* Canvas.java is the graphical display space for the simulation, as in the environment where the turtle operates and that environment only.
* View.java is the superclass which all view implementations for our implementation must extend. It has public methods accessible for the controller to accommodate the functionality of our simulation (detailed later)
* ViewFileMenu.java is responsible for building the file menu for the view to implement.
* SLogoView.java is our implementation of how we felt the graphical interface should look. It is responsible for all user input pertaining to the underlying framework
* ColorChooser.java is responsible for the graphical element for the user to easily change pen colors. We thought this would be a worthwhile addition

Model:

* Control
  + The control gets passed an input from view and then uses the parser to generate an instruction that can be executed on the model.
  + The controller has access to the resource bundle, the parser, the environment, the model, and the view.
  + The controller needs the model so that it can execute instructions on the model and print instructions through the model
  + The controller needs a view so that the instruction that was inputted by the user could be displayed on the view.
    - This design was chosen so that the underlying simulation is independent of the view.
  + The controller has an environment so that it knows about all of the instructions, the scope, and the palette
  + Parser
    - The parser contains an environment so that it knows about all of the instructions
    - The parser creates an instruction to run based on what the user inputted.
    - If the user’s inputted instructions do not have the correct syntax, an error will be thrown in the parser which is caught in the controller.
    - The parser has the ability to unpack a list of instructions
      * This is done to parse instructions that contain lists (For example a for loop or an instruction that was written with brackets to make it clearer, sum [ sum [ 1 2 ] 3 ])
    - The parser calls the preparser, which has the most complex methods that are used to make parsing possible, at the beginning of creating an instruction.
      * The presparser has several methods that are used to add brackets to instructions in the correct locations. For example, the instruction sum sum 1 2 3 would be turned into sum [ sum 1 2 ] 3 ].
        + This was done so that all instructions (both those that were entered with brackets and those that were not) could be treated the same way by the parser.
      * The complex methods that are in this class are all in one place that no class except the parser ever has to deal with this class
  + Environment
    - The environment stores the scope, the map of instructions, and the palette.
    - The environment changes the scope, adds and removes functions, and is used to save and load workspaces.
    - InstructionMap
      * This class contains all the local instructions, global instructions, and variables
        + This allows all these data to be packaged together into one object for a given scope. That is, when looking up a variable or instruction at a given scope, the Environment checks every InstructionMap at the current scope and more global scopes.
* Factories
  + MapFactory<K,V>
    - Abstract generic class. This is used for creating Maps of values or keys that are read in from file. This class focuses on consolidating file reading (along with corresponding error messages and function), such that all subclasses only need to define how to parse text to create a new mapping.
  + PrototypeMapFactory<V>
    - Extends MapFactory. Generic factory class that uses reflection of file names to create a mapping of keywords to object instances. This reads data from a text file. Since it is generic, it can be used to load any type of object. In SLogo, it is used for mapping Instructions and LineBuilders from strings or indices to object instances.
  + IndexMapFactory<V>
    - Extends MapFactory. Abstract generic factory for creating a mapping of indices to objects. This reads in data from file to create objects and to assign indices. This allows for various specialized Factory subclasses to be defined from reading in colors, pictures, backgrounds, etc.
  + PaletteFactory
    - Uses various subclasses of MapFactory to create mappings necessary for the palette: images, backgrounds, colors etc. Consolidates many factories into a single factory that the environment can use to a palette.
* Simulation
  + Model
    - The model stores the state (and instance of the State class), the resource bundle, and a view.
    - State
      * This class is an inner class in the model that stores the following variables:
        + Map<Integer, Turtle> turtles;
        + Turtle activeTurtle;
        + int turtleID;
        + Collection<Point> lines;
        + Collection<StampSprite> stamps;
        + Environment environment;
        + Background background;
      * By creating this class, we were able to wrap all the state that the model stores into one class which would help for undo and redo. We wrote the code to make undo and redo work using stacks but it is commented out because we would have needed a copy method for each part of the state.
      * Turtle
        + The turtle class was limited to just have functions that impacted the turtle in order to make this module more flexible.
        + The turtle has the ability to update and paint itself as well as paint its status (x position, y position, and angle) to the top left corner of the canvas
        + The turtle also has the ability to change and get its image, to get its location, to move, and to get the pen that is being used to draw the line behind it
        + We felt that with these methods it would be easy to extend the turtle to create other sprites that could take the turtles place because they would be able to use most of the functionality of the turtle class
      * Point
        + This class allows us to easily change the thickness and color of points which are used to draw lines
      * StampSprite
        + This class is only used to make a permanent stamp (until the screen is cleared) of the turtle on the canvas
        + This class was created so that the Sprite that it represents is fixed on the screen and cannot be translated around on the screen. This is achieved by the resetting to the original location data that the superclass Sprite has knowledge of.
      * Background
        + This class has a list of PriorityPixmaps, a Palette, and an index of the current background image.
        + This class has the ability to paint the background image onto the canvas, change the background image, and turn the grid on and off.

This class essentially has everything that could be displayed as the background for the canvas and is able to make the background be whatever it is supposed to be

PriorityPixmap

This class stores the priority of a pixmap

This class is used so that we can sort the background images so that they can be painted in the correct order in the background class. For example, we want any background image (like a green background to be painted before a grid is painted so that the grid can be seen over the green background

* + - The model updates the turtles and paints the turtle, background, and lines

**Design Considerations**

One of the biggest considerations in design was factoring in how to split our team into two mostly independent subteams. This requires being very firm about which methods that the View team knows about that are present on the Model side, and which methods the Model team knows about which are present in the View side. This was the primary motivation for the Controller class that we created. The Controller serves as the sole communicator between the View side and the Model side, and has every public method that the View should ever need to call. These are: createRunInstruction(String command), saveState (InputStream is), loadState (OutputStream os), clear(), and setSaveOption(SaveOption option). In fact, the Controller class itself does not have very much code itself other than passing inputs from the View side to the appropriate location in the Model side. By creating this class, we had a very clean interface that the View team can use to access any appropriate actions. This is why it was the first class we designed.

**Remaining Issues**

* The Operating System level close button (ie red “x” in Windows or red circle in Mac) does not kill the process running our simulation- it only closes the view. We were presented with the choice of this error or that all running workspaces would be terminated upon any of them being closed out. Our quit button, however, successfully terminates the underlying process. Additionally, the user can enter a “quit” instruction into the console to properly close the entire process.
* If a command is entered that requires the turtle to move more than 6990 spaces, we get a stack overflow error. This occurs because the method we use to draw lines recursively draws the lines one point at a time. For example, if fd 7000 is entered the method will have to make 7000 recursive calls since it has to make a call for each pixel that needs to be drawn.
* User defined functions do not always get properly saved to a file. However, this behavior is not consistently reproducible. The current mechanism for saving uses Java’s Serializable interface to write Objects, and the error might have something to do with how that works.

**Team Responsibilities**

We had a group of five people and we were split up into two teams: The model team (and control) and the view team. Ellango, Ryan, and Scott were on the model team while Sean and Yoshi were on the view team.