**SLogo API**

Team Members: Sean Wareham, Elder M. Yoshida, Ryan Fishel, Scott Valentine, Ellango Jothimurugesan

**INTRODUCTION:**

SLogo is an implementation of a much simplified version of Logo. "Simple Logo", which retains the features most commonly used by beginning users so that it can be used to provide an introduction to computer programming. Our program will be split into modules of Model and View so that we can split into subteams and work independently. The View team will be handled by Elder and Sean, and the Model team will be handled by Ellango, Ryan, and Scott.

**DESIGN GOALS:**

Model, View, and Control are separate modules. Our design is flexible to the addition of new commands for extensibility. We also have an interface for what the view must contain so that new view could be easily written for our API. We make the assumption that workspaces correspond to separate windows. This makes our design inflexible in that the workspaces cannot communicate with each other.

**SAMPLE WORKFLOW:**

Example: The user enters ‘fd 50’

The view has an ActionListener that waits for the user to click the Submit button for commands that they have entered, and the actionPerformed method runs myController.createRunInstruction(“fd 50”). The Controller then hands off the action to the Parser, by calling myParser.generateInstruction(“fd 50”). The Parser then hands it to a Preparser which does some initial processing. Then the parser operates one token at a time, first looking up “fd” in the environment with myEnvironment.getInstruction(“fd”) which returns a Forward object with no arguments. Then Forward.load() is called with the rest of the line to load in the single argument it needs with parser.nextInstruction(). Because the next token is “50”, the number will get wrapped in a ConstantInstruction class and returned to the Forward object, which after being loaded, gets returned back to the Controller. Then the Controller does Forward.execute(model) which first evaluates how much to go forward by with nextOperand().execute(model) which evaluates its ConstantInstruction and gets the number 50. Then it gets the turtle with model.getTurtle() and tells it to go forward 50 pixels by creating a Vector v = new Vector(turtle.getAngle(), 50), and then calling turtle.translate(v). The Forward instruction then finally returns 50 back to the Controller which prints it out to the user with myModel.informView(50 + “”), which calls myView.displayText() with the string. Finally, when the Canvas.repaint() gets called on every step, it calls model.paint() which calls turtle.paint() which paints the new turtle’s location on the screen.

Example 2: The user enters “fwd 50”, an invalid code

The process is the same as the previous until it gets to looking up “fwd” in the environment with myEnvironment.getInstruction(“fwd”). The Environment will see the keyword is not present in its data, and then throws a new llegalInstructionException(message) which bubbles back all the way up to the Controller which catches it and then calls myModel.informView(exception.toString()) which gets the string representation of the exception and prints it to the history to the user.

**PRIMARY CLASSES/IMPORTANT METHODS**

**Full description available at** [**http://people.duke.edu/~ej48/slogo/**](http://people.duke.edu/~ej48/slogo/)

Main.java

* new SLogoView(String programTitle, String language)
  + View is responsible for instantiating Model and Controller

Model Team:

* Control Package
  + Controller.java
    - Passes instructions to the parser and executes those instructions on the model. Saves and loads the state of the Environment.
      * public void createRunInstruction (String s)
        + This creates an instruction from a given command from the view by having the parser generate an instruction. It then runs the instruction that was created.
      * public void saveState(OutputStream os)
        + Saves the instructions, variables and palette that are available to the user.
      * public void loadState(InputStream is)
        + This loads in the state of instructions, variables, and palette from a source that was saved by saveState().
      * public void setSaveOption(SaveOption option)
        + Set the Controller to save automatically or when the user chooses
      * public void clear()
        + Tells the Model to clear its state.
    - public enum SaveOption { AUTO, MANUAL }
      * A type representing whether the Controller should save the environment automatically or only when the user specifies to
        + public abstract String getResourceName ()

Gives the name of the saveoption

* + Parser.java
    - Converts a user inputed string into an Instruction that can execute on the model.
      * public Instruction generateInstruction(String s)
        + The Parser takes in a String and returns an Instruction object as the output or throws an exception if not a legal instruction
      * public Instruction nextInstruction (Scanner line)
        + Reads through the next single complete Instruction
      * public String unpackList (Scanner line)
        + Removes brackets from provided input
  + Preparser.java
    - Handles initial processing of the raw user input. This creates a form that the Parser class can read.
      * public String preParse (String s)
        + Takes user input, converts to lower case so that case does not matter, removes lines starting with a do not read message, and adds brackets so that the parser knows where arguments start and end.
  + Environment.java
    - Represents the current available commands and variables and palette in the workspace. Any user defined variables or instructions are also added to the environment.
      * public void addInstruction (String keyword, Instruction userInstruction)
        + Adds a new user defined instruction to the environment.
      * public void defineFunction (String keyword, Instruction instruction)
        + Defines a new user function.
      * public void defineVariable (String keyword, Instruction value)
        + Defines a new user variable.
      * public void removeInstruction (String instructionName)
        + Deletes an instruction from the environment.
      * public String customValuesToString()
        + Gives all user defined functions and variables as a string.
      * public BaseInstruction getInstruction()
        + Gives the Instruction associated with the passed keyword.
      * public void inScope ()
        + Change current variable scope to more specific
      * public void outScope ()
        + Change current variable scope to more general
      * public void save (ObjectOutput out)
        + Save the state of the environment to be loaded in later
      * public void load (ObjectInput in)
        + Load in the state of the environment
  + InstructionMap.java
    - represents all instructions (which includes all variables, local and global).
      * public void addInstruction (String keyword, Instruction userInstruction)
        + Adds a new user defined instruction to the environment.
      * public void addUserDefVar (String keyword, Instruction value)
        + Adds a new user defined variable.
      * public void addUserDefFunct (String keyword, Instruction instruction)
        + Adds a new user defined function.
      * public String variablesToString (String header)
        + Makes a string containing info for all user defined variables.
      * public String userDefinedInstructionstoString (String header)
        + Makes a string containing information of user-defined functions.
      * public boolean containsKey (String key)
        + Whether the key is corresponds to an instruction in this.
      * public BaseInstruction get (String key)
        + Returns the instruction corresponding to the passed keyword.
      * public void remove (String key)
        + Deletes an instruction from the environment.
      * public void save (ObjectOutput out)
        + Save the mutable state of the InstructionMap to be loaded in later
      * public void load (ObjectInput in)
        + Load in the mutable state of the InstructionMap. Objects must be loaded in the same order they were saved.
* Instructions package
  + Instruction.java
    - Interface representing every user or otherwise generated instruction that acts on the model.
      * public int execute (Model model)
        + executes this Instruction on the model
      * public String toString ()
        + Gives a string representation of this instruction.
  + BaseInstruction.java
    - Implements Instruction Interface. Abstract class that all instructions in the SLogo instruction set should extend
      * public void load (Scanner line, Parser parser)
        + Loads this instruction from a scanner of user input.
      * public Instruction nextOperand()
        + Gives the next operand in the instruction.
  + CompoundInstruction.java
    - Implements Instruction interface. Represents a list of instructions as one whole instruction.
  + Booleans subpackage
    - BooleanInstruction.java
      * Represents instructions that evaluate as true or false
    - And, Equal, LessThan are examples of subclasses of BooleanInstruction that are present in this package
  + Math subpackage
    - All math instructions, such as Sum, Product
  + Turtle subpackage
    - All instructions that operate on the Turtle, such as Forward, SetPenColor, XCoordinate
  + User defined subpackage
    - Instructions for control flow, such as For, DoTimes, IfElse
    - Instructions for creating user defined instructions and variable instructions
    - Instruction representations of previously defined variables and user defined commands
  + Extra subpackage
    - Other instructions, such as GridOn, SetBackground, and Rainbow
* Factories Package
  + MapFactory.java
    - Abstract Class that acts as a factory for maps that map keys to objects. This class allows for object data to be read in from file.
      * buildIndexMap()
        + Builds a Map of prototypes based on index in text file.
  + PrototypeMapFactory.java
    - Extends MapFactory. Creates a map for all of the keywords to a prototype of the corresponding class. Creating the map requires an index file. This is set in the constructor.
      * public Map<String, V> buildStringMap ()
        + Builds an instruction map that maps keyword strings to their instructions.
  + Palette Factories subpackage
    - IndexMapFactory.java
      * Extends MapFactory. Abstract Class that acts as a factory for maps that map integer indices to objects. This class allows for object data to be read in from file.
        + protected abstract V getObject (String objectData)

Returns the object to put in the map

* + - Other subclasses of IndexMapFactory that generate specific objects for the Palette:
      * ImageMapFactory
      * BackgroundMapFactory
      * ImageMapFactory
* Simulation Package
  + DisplayEditor.java
    - Interface for objects that can have their graphical elements manipulated
      * public void addLine (Point line)
      * public void addStamp (StampSprite st)
      * public Palette getPalette ()
  + Model.java
    - Implements DisplayEditor. Represents the simulation of the drawings on the screen. The model holds the drawer (the turtle) and all of the lines that have been drawn. For the model to be ready for use, both its constructor and initialize() need to be called.
      * public Environment initialize ()
        + Initializes the model with an environment and at least one active turtle.
      * public void update (double elapsedTime, Dimension bounds)
        + Updates the elements of the Model
      * public void paint (Graphics2D pen)
        + Paints all current elements (turtles and lines) of the model, and the background layer.
      * public Turtle getTurtle ()
        + Returns the active turtle
      * public void switchTurtle (int index)
        + Switches the active turtle to turtle with the provided ID.
      * public void clear ()
        + Clears stamps and lines from model.
      * public void clearLines()
      * public void clearStamps ()
      * public void informView (String s)
        + Calls the view method to display the result of the command, or an error message back to the user. to differentiate the result from commands issued by the user.
  + Turtle.java
    - Represents the turtle displayed on the screen. Extends Sprite class.
      * public static Location startingLocation()
        + Calculates the location of the center of the screen (the starting location) based on the size of the canvas
      * public Location getLocationOnCanvas ()
        + Calculates the current position of the turtle as a location with (0,0) at the center of the canvas.
      * public void changeImage (int index)
        + Replaces the current image with the image corresponding to the provided index in the palette.
      * public void paintStatus (Graphics2D pen)
        + Paints the text which represents the turtle's position and heading
* Exceptions package
  + CorruptedEnvironmentException.java
    - A runtime exception that is thrown when the Environment tries to createa new instruction but fails. This typically means a programming bug such as not loading the environment correctly, or not providing an appropriate instantiation method for a instruction in the library.
  + IllegalInstructionException.java
    - Exception for instructions and variables that are not defined in the current environment or for using improper syntax.
  + UndoException.java
    - Exception when there are no more states available to restore.
  + RedoException.java
    - Exception when there are no more states available to restore.
* Drawing package
  + Background.java
    - Represents all possible background layers for the Canvas. The Background is paintable and paints all images in a pre-defined order. Currently supports two layers: one grid, and one color image, but more can be added by adding additional Maps for each type of background, and additional indices
      * public void gridOn ()
      * public void gridOff ()
      * public void switchColorImage (int index)
        + Switches the background color image with the one specified in the Palette at the provided index
  + Palette.java
    - Represents all of the available colors, lines, and shapes in the active environment.
      * public void addColor (int index, int r, int g, int b)
      * public void save (ObjectOutput out)
        + Save the mutable state of the palette to be loaded in later
      * public void load (ObjectInput in)
        + Load in the mutable state of the palette
  + GraphicsMap.java
    - Keeps track of all of the graphical options available (e.g. colors, linestyles). A GraphicsMap is essentially a specialized map that is constrained to SLogo specifications and supports SLogo exception handling.
      * public E get (int index)
      * public void put (int index, E element)
  + PriorityPixmap.java
    - Extends Pixmap. Represents a Pixmap that can also have an order of precedence when painting by calling Collections.sort() on a collection of PriorityPixmaps. A lower myPriority value corresponds to a higher priority.
  + StampSprite.java
    - Represents a stamp of the turtle in the workspace.
  + Pen.java
    - Paints lines
      * public void draw (double mag, Dimension bounds)
        + Draws all necessary lines between the start and end positions.
      * public void penOn ()
        + Puts the pen down: enables the drawing of lines.
      * public void penOff ()
        + Picks the pen up. Disables the drawing of lines.
      * public void changeColor (int index)
      * public void changeLineStyle (int index)
      * public void changeSize (int size)
  + Line subpackage
    - Point.java
      * Represents a 2d line or variable thickness and color.
    - LineBuilder.java
      * Interface for drawing a line in a certain style.
        + public abstract Point buildLine (Location start, Location end, double thickness, Color color)
      * SolidLine, DashedLine are examples of classes that implement LineBuilder
* Util package
  + Location.java
    - Extends Point2D.double. Represents a Location (in pixels) on the screen and adds some utility functions to the Point2D class.
      * public void translate (Vector amount)
        + Move this location by given vector.
      * public boolean tryCorrectingBounds (Dimension bounds)
        + Resets the provided location to lie within the provided bounds
  + Vector.java
    - A mathematical vector with magnitude/direction
  + Sprite.java
    - Represents a shape that moves on its own.
      * public void update (double elapsedTime, Dimension bounds)
        + Describes how to "animate" the shape by changing its state.
      * public void paint (Graphics2D pen)
        + Display this shape on the screen.
  + Pixmap.java
    - This class represents an image on the screen and adds some utility functions to the Image class.
      * public void paint (Graphics2D pen, Point2D center, Dimension size)
        + Describes how to draw the image on the screen.
  + Text.java
    - This class represents text that appears on the screen and adds some utility functions to the Image class.
      * public void paint (Graphics2D pen, Point2D center, Color color)
        + Describes how to draw the shape on the screen.
  + ValueText.java
    - Extends Text. This class represents text that is a labeled numeric value.

View Team:

* View.java
  + Highest level of the view. All visible elements are contained within this.
  + Contains: menu bar, buttons, text fields, and the Canvas
  + Methods:
    - public void displayText(String text)
      * This method will be used to display textual information to the user via the GUI. This data can be exemplified by the Turtle’s current position/ heading as well as Error Messages
      * Also can be used to display commands that are executable by clicking on them
    - public void setController(Controller controller)
      * sets the controller
      * this is needed for the sequence of objects created in Main.java
    - protected JMenuBar createMenuBar()
    - Collection<JButton> createButtons()
    - Collection<JTextArea> createTextFields()
* Canvas.java
  + Is the working environment for turtle.
  + Is where objects in the simulation are painted.
  + Methods:
    - public void paintComponent(Graphics2D pen)
      * This method paints all Graphics objects such as the turtle and the lines
      * is an overridden method inherently called by JComponent methods
    - public step()
      * Tells the model to update the simulation.

**ALTERNATIVES TO DESIGN**

We could have kept drawing elements and instructions separate instead of both living in the Environment, or store this state divided up and localized. This would allow for more modularity and reusability of these components. We did not do it this way because we did have the flexibility since our original design included only storage of instructions, and combining drawing elements and instruction storage allowed for the least changes.

The individual instructions could be more responsible for reading in their arguments instead of offloading that work the Preparser to divide arguments up and the Parser to read them in. This would reduce coupling between Instructions and the Parsers, but we did not take this approach due to a lack of foresight in the complications of parsing.

The drawing of lines onto the screen and all associated tasks (change of color, style of lines) could take place on the View side of the architecture instead of the Model. That is, the Model side just tells the View when, where, and how to draw the lines, but the View does the actual visual editing on the screen. We opted for the current design of the Model doing it so that the Canvas class could be reused from previous projects which requires each element to know how to draw itself.

## UML



