SLogo Design Document

Chinmay Patwardhan, Lalita Maraj, Ken McAndrews, Susan Zhang

Oct. 3, 2013

# Design Goals

The goal of our project design is to break the project into two discrete modules: the model (Interpreter) and the view (GUI). These modules will only be able to communicate with each other as necessary.

The model will be in charge of parsing input received from the GUI, pending instructions in the queue, reporting errors, and tracking all stored variables, commands, as well as the Turtle’s current location and direction. Naturally, this module will be composed of separate submodules to handle each of these tasks.

* Interpreter: Will take in and process the input from the GUI. Will process desired commands and add necessary Instructions to the queue of pending Instructions
* Pending Instructions: Will keep a queue of the Instructions that have yet to be reflected in the view module. The view will be able to request that the next Instruction be processed if there are pending Instructions when the previous Instruction has finished animating (if it needs to animate)
* Stored variables: Will store names and values of available variables in a structure that is viewable and editable by the view module
* Stored commands: Will store names and instruction sets of available commands in a structure that is viewable by the view module
* Turtle: Will store the x and y location of the turtle, the turtle’s heading, if the pen is down, if the turtle is visible, and the turtle’s image
* Errors: A hierarchy to describe the various errors that may occur during parsing of input

The view will be in charge of sending user input to the Model to be parsed and retrieving the updated turtle data, currently available commands and variables, and a log of the user input. The View uses the updated turtle data to display the turtle actions.

* Display Window : will use the updated turtle data such as angle, x/y coordinates and pen status to appropriately display the turtle’s actions.
* Text Box : will receive user input via the keyboard or clickable commands from the command log and/or available commands/variables from the sidebar.
* Run Button : will gather the user input from the textbox and relay the String to the model.

# Primary Classes & Methods

* Model class
  + gives public access to all main submodules used by the backend for the frontend to query
* Interpreter class
  + Protected constructor
  + Public parseInput method
* InstructionQueue class
  + Protected constructor
  + processNextInstruction method to return back an instruction of type DataType
* Abstract Instruction class
  + inherited by various instruction types to create various instruction objects such as movement, rotation, math functions, setting variables, etc.
* DataType<ClassName> class
  + allows us to encode strings, integers, and doubles in a single class
* DataType factory
  + returns a DataType of the appropriate class for a given string representation of the data
* VariableCache class
  + temporarily stores all variable names and values used until user wishes to clear
* CommandCache class
  + temporarily stores all command names and bodies used until user wishes to clear
* Turtle class
  + stores information about the turtle (e.g. location, heading, is drawing, etc.)
  + this information is publicly available to the frontend
* CommandHistory class
  + stores the history of commands inputted by the user for reference/use later
* View Class
  + orchestrates the different stages to update display window as well as display current variables and commands and a command log
* Textbox Class
  + is able to receive input from the user via the keyboard or from a clickable sidebar module
* Sidebar Class
  + Consists of distinct modules that represent available variables, available commands, and a log of user input
* Module Class
  + has 3 subclasses: CommandModule, HistoryModule, VariableModule
  + each subclass allows us to represent its respective data as separate clickable buttons each with slightly different update and click functionality

# Example Code

**Executing User Input from textbox:**

1. Text is entered by user
2. When run button is pressed, text is sent to Model to be parsed and added to instruction queue

Model.parseInput(inputString)

1. View gets next instruction (and processes) from queue until queue is empty:

while (myModel.hasNextInstruction()){ myModel.processNextInstruction(); myViewUpdater.displayOutput();

mySidebar.update();}

1. For each instruction the view is updated by adjusting the turtle according to the information that is sent back by the Model, and the sidebar is then updated with new variables, available commands, and history that is stored within the Model:

protected void executeInput(){

while (myModel.hasNextInstruction()){ myModel.processNextInstruction();

updateDisplay();

updateSidebar(); }

}

**Executing Clicked Sidebar Module Command:**

1. User clicks on sidebar command module and click() method is called to retrieve text associated with the clicked module
2. Text associated with clicked module is populated in textbox for user to modify

Textbox.addInput(clickedComandString)

1. When run button is pressed, steps 2-6 from above are executed

**Executing Variable Editing via clicking Variable module:**

1. User clicks on sidebar variable module and inputs updated variable value :

click() method is called to gather variable name and value

1. Variable name and user input is used to update BackEnd VariableCache:

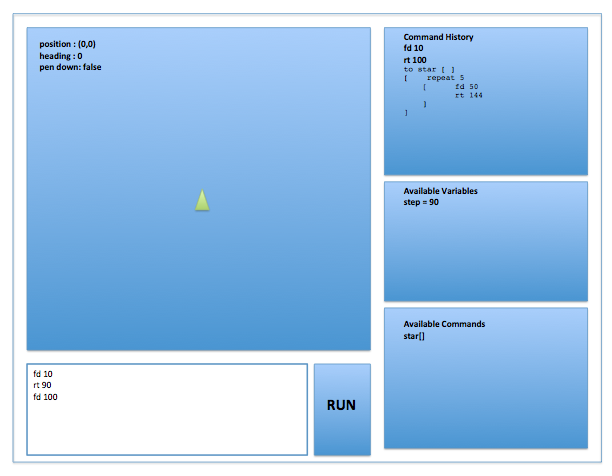
Model.putVariable(variableName,userInputValue)

1. Sidebar is updated to reflect updated variable value display

sideBar.update()

1. When run button is pressed, steps 2-6 from above are executed

# Frontend Mockup



# Design Alternatives

We decided to have a pending instruction queue that the GUI can process at a desired rate. This helps the GUI properly animate the turtle’s movements at a speed that is visible to human eyes. An alternative would have been to immediately process each of the instructions as opposed to inserting them in a queue.

For updating the modules, we decided to have each of the modules re-initialize all their data each time a command is run. Another way of doing this would have been adding detection of new additions to the classes that hold the data, but this would have over complicated the communication between the model and the view. The view would have needed to query for newly added commands

# Team Member Roles

Lalita and Susan will be working on the front end (View), while Chinmay and Ken will be working on the back end (Model).