SLogo: Team 8

Leeviana Gray (lpg6@duke.edu)
Jiaran Hao (jh370@duke.edu)
Graham Miller (gcm9@duke.edu)
Paul Wright (ptw5@duke.edu)

Design Goals:

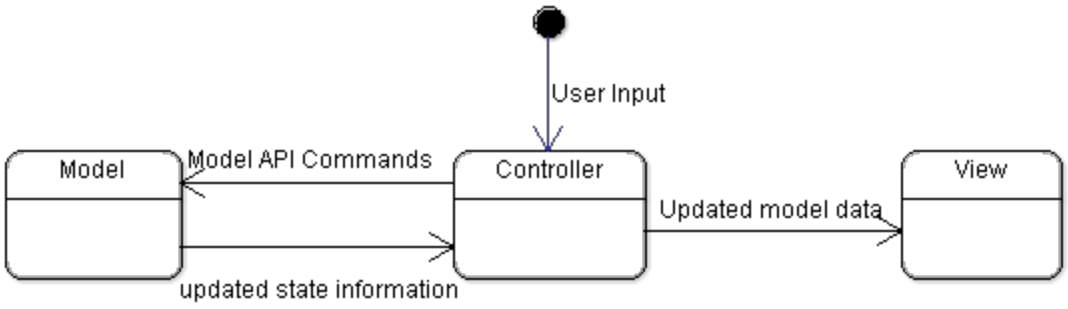
Our goal is to successfully separate the GUI from the actual program processing by implementing a version of the Model-View-Controller model.

Each of these modules will have specific responsibilities. Our design is structured such that the controller will take in the user's input and send it to the model. The model will then process the user's input and update its state. The controller will then ask the model for it's updated state. Finally, the controller passes this updated state to the view which updates the GUI representation of the model.

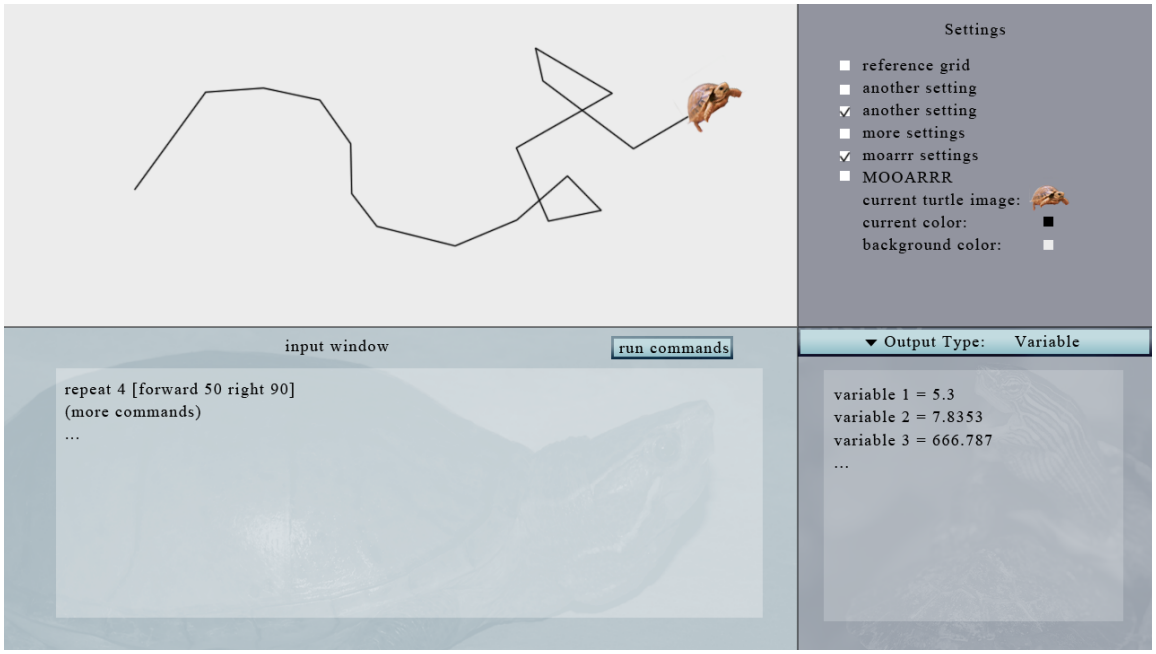
This design, as detailed below, allows us to make both the model and the view very flexible as long as they retain the same set of interactions with the controller. These interactions will have to remain the same, but the way in which the model and the view use and produce the data passed using these interactions can change to meet any given set of specifications.

Primary Classes and Methods

UML Diagram:



Proposed user interface:



Model API (how the controller "talks" to the model)

runCode(String code);

returns Collection of Trails

returns Feedback object

getCommands();

returns Map (?). Keys are commands, Values are number of arguments

get various environment variables and turtle attributes (X, Y, angle)

returns the appropriate variables

set environment variable("name", double NewVariable)

sets the appropriate environment variable

Trail Object:

Has initial and final position objects

Position Object

X and Y

View Program Design:

Base View Class:

The GUI has four parts:

Display Window. Draw turtle and its trails.

Input Code Window let clients input the code they want to run.
Output Window is a kind of CardLayout component to let clients choose which information they want to display. Currently, there's variables, user-define command, commands history.
Settings. Checkbox and button allows user to change the settings of the GUI. Eg, toggle reference grid.

Class MainFrame:

MainFrame is the container where we holds all of our GUI objects. We provide maximum flexibility by letting the user to shrink and expand each part of our screen. It divides our screen into 4 parts each of which holds GUISections.

Class GUISection(name can be changed at last):

GuiSection is the type of the 4 windows we mention about. It is also a container(the subclass of it defines what it contains). All GuiSection subclass should have some similar style. GuiSection class handles that as a base class.
SubClasses of GUISection: Setting,Display,RunCode, Output

These subclasses implement specific function. How they draw pictures or strings or show numbers is based on the data provided by the controller(from the Model). And how to interpolate the data received from controller class is defined by the subclass.

Factories: Except Display and Runcode is relatively fixed. We want to offer Settings and Output extensibility by allowing the user to specify what to put into our GUISections. (Probably we may allow them to read from a file). A factories design pattern will make the code flow open and close principle so adding a new class just mean modifying the factories(adding new subclass maybe) instead of other parts.

Controller Program Design:
Notice: In Swing, the difference between controller and view is vague. Here we call this Controller program because it is responsible for trading message between Model and View.

Controller Function:

Uses Model's API to change state, and pass in data

Sends Data objects to View (different types of data objects extend base Data class, ex. PositionData)

Has the listener for buttons and such

Model Program Design:
Changes state and sends data to controller

Class Hierarchy:

Parser - Parses code (stack processing), deals with []'s and such

Command // have a .do() method and a # of arguments instance variable

- TurtleCommand
  - MoveTurtle
  - ChangeTurtle
    - HOME, SHOWTURTLE etc.
  - StateChange
    - CLEARSCREEN
- Queries
  - XCOR, HEADING, etc.
- Math
  - SUM, DIFFERENCE, SIN etc.
- Boolean
  - AND, OR, GREATER etc.
- Set
  - MAKE, SET, TO (new command)
- Loops
  - REPEAT, DOTIMES, FOR
- Control Structures
  - IF, IFELSE
- User-defined commands
  - Commands that the user defines in TO

API // all public classes that controller calls upon

- Queries

getFeedback, getCommands, getTurtleX etc.

Variable - holds all environment variables

Feedback - Stores any programmatic feedback (i.e. error strings), instance variable
boolean hasError;

Example API Code (model methods the controller can utilize):

The user just typed 'fd 50' and sees the turtle move in the display window leaving a trail

```
Controller:   Model.runCode('fd 50') // Model's parser, parses the string and runs
                                        the command stack
              Model.getFeedback(); // gets Feedback object
              Model.getTrails();    // get Collection of Trails
              Model.getPosition(); // gets current Position Object
              packageAndSendDataToView(); // sends Data Objects to View
```

View: displays the trails and current turtle position using Data object from Controller

Design Alternatives

1. What if the model fed information DIRECTLY to the view, bypassing the controller. In many model-view-controller models the model directly updates the view by passing it's instance variables to the view. In our case, however, we decided that it would be distinctly easier to have the model's updated states pass through the controller to the model. This allows for the model to ignore the existence of the view and the view to ignore the existence of the model because both the view and the model to exist within the controller. As long as it retains it's functionality with the controller, we can change the model in many ways without needing to make any changes to the view. Likewise, we can change the view in many ways without needing to make any changes to the model. This provides us greater flexibility in terms of changing the GUI as well as the backend of the application. Also, the java.swing we are using somewhat combines the controller and the view (from Oracle Website). For example, the button has both view and controller parts. So in order to fit in the swing pattern, we do not choose this alternative.

2. What if we retained the model-view-controller model but used the controller module to parse the user's commands rather than the model module? This is also disadvantageous. This would mean that several essential API components would be contained within the controller, making the API less flexible. By creating a standardized set of interactions between the model and the controller and allowing the model to parse the user's commands we allow changes to be made to the overall set of usable commands far more easily. Consider adding additional user commands: In the design alternative we would have to change code in both model and controller but in our design we would only have to change code in the model. It is far better to implement the

extensible elements of our program such that code only has to be changed in one module.

Project Division

View and Controller:

Jiaran and Paul

Model:

Graham - Focusing on creating command classes

Leeviana - Focusing on parser first

Both - Write tests first.