APIs

Front-End to Back-End (public things in "frontEnd" package)

```
public class View {
        public View(Model model, String language);

        public void display(...);
}
```

Back-End to Front-End (public things in "backEnd" package)

```
public class Model {
        public Model(View view);
        public List<ScreenUpdate> run(String input);
}

public class ScreenUpdate {
        // Returns a Map with a Key of type String representing the kind of thing you're updating
the screen with, and a Value...
        public Map<String, T> giveUpdate();
}
```

Back-End to Back-End (within "backEnd" package)

```
class Parser {
        Parser
        List<Command> checkAndReadInput(String input);
}

class ScriptManager {
}

class Turtle {

}
```

Front-End to Front-End (within "frontEnd" package)

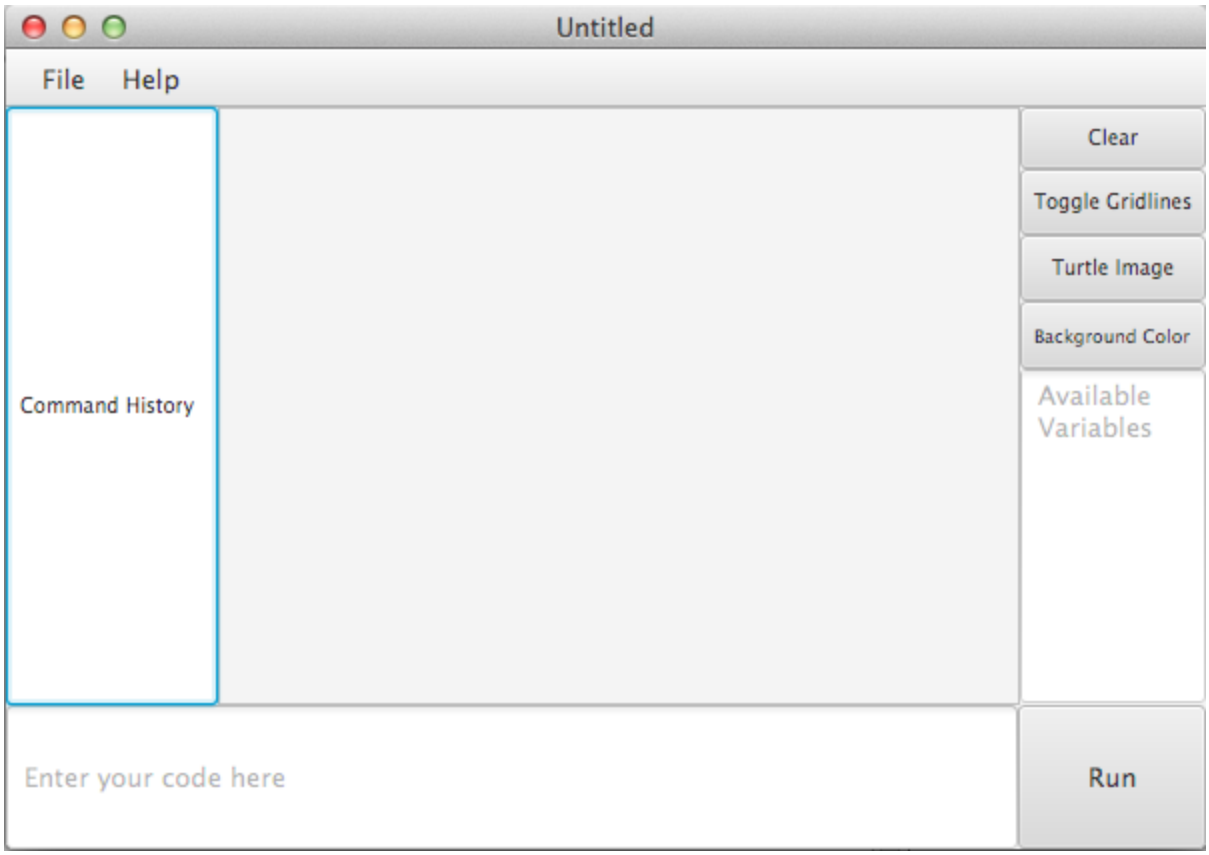
```
Public class View(){
public void run(String input);

}

public class Model(){
public void display(List<LinePoint>);
```

}

Interface Command(){

}

public class LineDrawer{
public LineDrawer();
draw(Collection<LinePoint>);
}



Design Goals
The goal of our design was to minimize the interactions between the front end and back end by creating View and Model classes, which serve as the sole points of interaction between the frontend and backend.

The only information the back end needs from the front end is the input entered by the user. Our view will contain an instance of the model and call myModel.run(String input) to pass off this

information. The view only allows for the user to pass off the information without processing any of the input.

However, the frontend also needs information from the backend regarding what needs to be drawn as a result of the commands. This each of these updates will be contained in a ScreenUpdate object created by the Turtle in the back end. This will contain attributes such as the turtle's old x and y location, the new x and y location, the color of the line that needs to be drawn, any specifications as to text that needs to be drawn, and an orientation. The Drawer in the front end will translate this Update object into information the NodeFactory needs in order to create a node. The NodeFactory will take in the shape or text as a parameter to the constructor and create a Node object, such as LineNode or TextNode. These Nodes will be passed back up to the view, where they will be added to a group to be displayed.

In the back end, we will have a multi stage process through which input from the front end will be turned into executable commands for the front end to display. The input received in the model will first be passed to the Parser which will check if the text is a valid command and will return a list of commands if it is. These commands will then be passed to the ScriptManager which will simplify and evaluate numeric or Boolean commands and assemble any turtle commands into a list of executions. This will be done by creating and calling a CommandFactory object which will instantiate commands that will either compute the numeric or Boolean values or break down commands into TurtleCommands. These TurtleCommands, will be passed to the Turtle, which will break down the executions (steps to be taken and displayed) into a collection of ScreenUpdate objects, which contain x and y coordinates, color, text/shape, and orientation that needs to be translated into actions by the graphical turtle.

Further, we wanted to enhance the extendability of our code in terms of adding new commands. This is done by our command hierarchy in the back end. We will have a Command interface, which will be implemented by abstract classes for each of the different types of commands such as TurtleCommands, TurtleQueries, MathOperations, BooleanOperations, and Variables, Control Structures, and User-Defined Commands. These abstract classes will be extended for each specific command that needs to be implemented. All command objects will be created by the CommandFactory and TurtleCommands will be passed on to the Turtle object, while all other commands are dealt with in the ScriptManager.

The ScriptManager only passes Turtle commands to the turtle. All other types of commands are dealt with after they are created by the Command Factory. The Parser only needs to know how to deal with Strings and produce commands, the Trutle only needs to take in TurtleCommands and turn them into ScreenUpdates for the front end to update the display.

As the Model and View control all other classes on the backend and frontend respectively, there is little dependency between classes and each class specializes in a single functionality.

Alternate Designs

An initial design of ours was split into two main modules: the model and view. The View contained the programs main, loop, and window/canvas, while the model contained a handler, which would parse through commands, and send them to the appropriate class to fulfill operations. The model also had a command interface that was implemented by internal and external commands. Internal commands consisted of operations such as performing a sum, or dealing with flow control (if statements or repeats). External commands were ones that directly interacted with the view, such as moving the turtle, rotating it, or raising/dropping its pen. External events were able to directly interact with the canvas, while internal operations were sent back to the handler, put into a collection, and given to the canvas class. The canvas class would then iterate through the collection and perform the stored commands. Lastly, the handler was also capable of dealing with throwing errors, and clearing the canvas if a reset button was pressed.

Another design decision we discusses was whether or not to create a Turtle object. We were initially afraid that the Turtle would be a passive data object that would only contain its current position and orientation. As a result we had a different method called Executor which would take executable commands and turn them into commands to update the view. In the end, we opted to create a Turtle object, but rather than leaving it as a passive data object, we added an Update method to convert TurtleCommands into ScreenUpdates. This Update method essentially performed the same action that the Executor function did.

We chose our final design because we wanted to minimize interactions between the frontend and backend. The view and model classes are the only two classes are the only two classes from the frontend and backend which can interact with each other; all other classes must act through one of these classes to interact with the other layer. The back end deals with parsing the commands, simplifying the commands into turtle commands, and returning a series of view updates to the model, which then sends the updates to the view. The view then uses these points to create nodes, update the turtle, and create lines. By doing so, we ensure that any logic is kept in the backend and the frontend only deals with creating lines on the canvas and updating the turtle's image's position on the grid. We removed the ability for commands to interact directly with frontend from our initial design because we felt that this didn't provide adequate separation between the frontend and backend.

Example Code

```
View.run(fd 50)

in the Model
myInput = “fd 50”

//create a parser
Parser myParser = new Parser(myInput)
```

String myCommand = myParser.checkAndReadInput(myInput)

ScriptManager myScriptManager = new ScriptManager(“forward 50”);

//in ScriptManager
ArrayList<Command> commandList = new ArrayList<Command>;

public ArrayList<Command> assemble(){
CommandFactory factory = new CommandFactory();
Command c = factory.buildCommand(“forward 50”);
//the command created here will be a ForwardCommand object
c.update();
commandList.add(c);
}

back in Model
myCommands = myScriptManager.assemble();
Turtle myTurtle = new Turtle()
List<ScreenUpdate> screenUpdates = Turtle.ProcessCommands(commandList)

myView.display(screenUpdates)

in the View
Drawer myDrawer = new Drawer()
//create a group to contain the lines to be displayed
Group myGroup = new Group();
myGroup.add(myDrawer.draw(screenUpdates))
//myDrawer.draw returns a new Line taking in the start and end x,y coordinates

display the group in the View and the line is drawn.

Test
package tests;

import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;

public class junit_tests {

    @Test
    public void testInputReceived(){
    Parser myParser = new Parser();

```
    String myCommand = checkmyParser.checkAndReadInput("fd 50");
    assertEquals(myCommand,"forward 50")
    }

    @Test
    public void testScriptCompiled(){
       ScriptManager myScriptManager = new ScriptManager("forward 5+45");
       ArrayList<Commands> myCommands = new ArrayList<Commands>;
       myCommands = myScriptManager.assemble();

       for(int i =0; myCommands.size(); i++){
          assertEquals(myCommands.get(i),"forward 50");
       }
    }

    @Test
    public void testCommandsProcessed{
       ArrayList<Commands> myCommands = new ArrayList<Commands>;
       myCommands.add("forward 50");
       Turtle myTurtle = new Turtle();
       List<ScreenUpdate> screenUpdates = Turtle.ProcessCommands(myCommands);
       assertEquals(screenUpdates.get(1).yCor,50 );
    }

}
```

Roles

Anna: Parser
Brian: Front-end
Eli: ScriptManager
Ethan: Commands