Design Goals (high-level description of the project's "vision")

Our design has three main modules: the visual, the backend logic, and the controller to facilitate communication between the front and backend. The goal of the visual module is to be able to pass the commands from the user to the controller for parsing and calculation by the backend. Afterwards, the backend will tell the controller what needs to be done by the visual. The controller will then send this information to the visual in order for the user's command to be seen visually. The subclasses of the frontend will include a Button hierarchy of classes, a layout class, and a menu class. The goal of the backend module is to take in parsed commands and do the calculations necessary in order to move a Turtle object. Thus, the backend will have a Turtle object class as well as a Model class to calculate the movement of the Turtle. The controller's role is to communicate between the backend and frontend and to parse the commands of the user in a way that each end can understand through their methods. Thus. the controller will have a parser subclass.

View API's

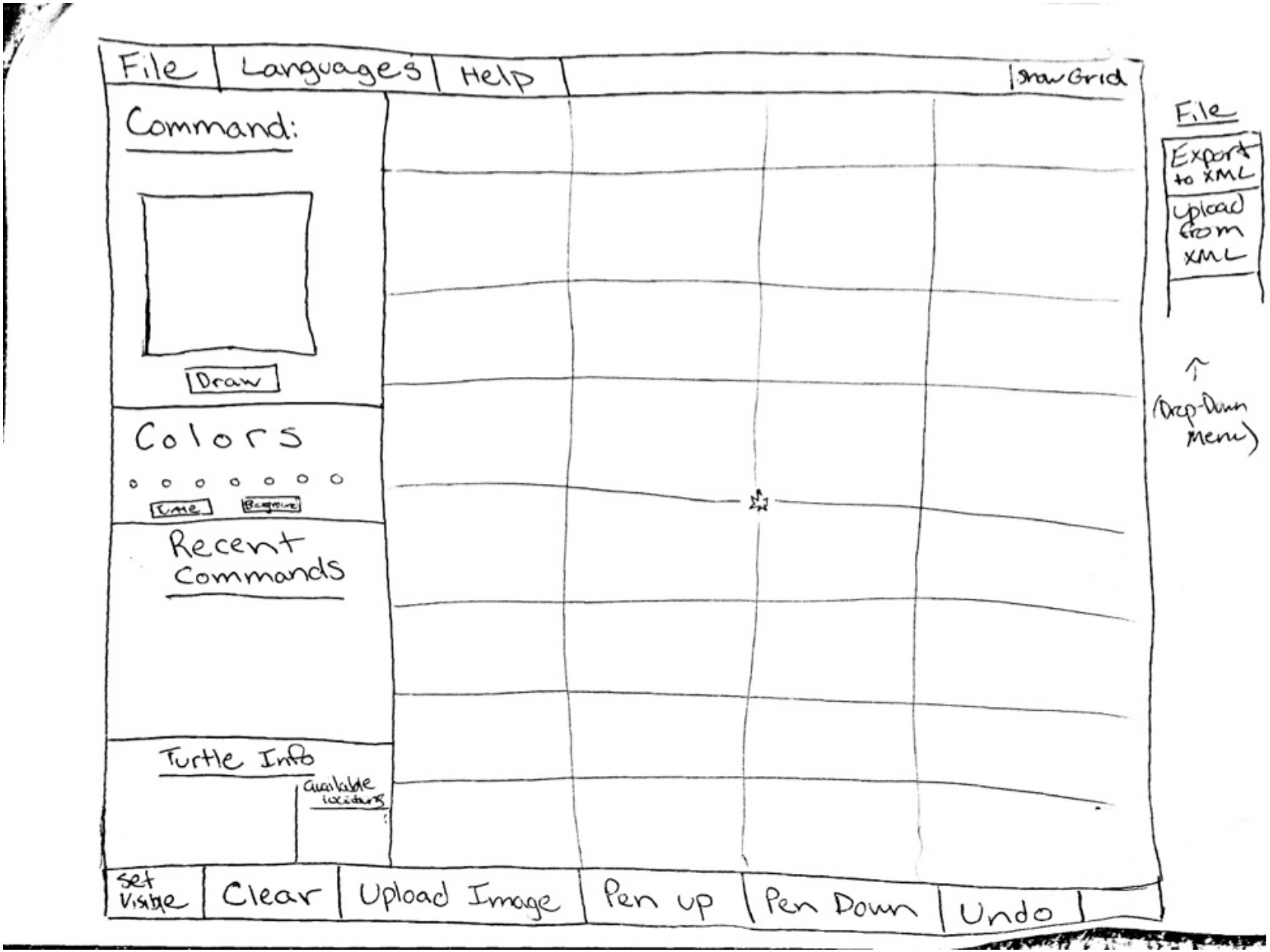
Inward Facing:

- draw Line(int x, int y): draws a Line from the lines' current location to the desired location (x,y)
- moveTurtle(int x, int y): moves the ImageView of the Turtle from its current location to the desired location (x,y)
- clear(); Clears everything drawn on the mainPanel
- initialize(): builds the beginning layout of the GUI
- build(): returns a KeyFrame for the Timeline of the animation to use.
- setTurtleVisible(boolean b): Determines whether or not the Turtle is visible
- showError(String message);displays the error on the screen
- private Button makeButton(String s, EventHandler<ActionEvent> handler)makes a button
- move(int x, int y):moves the Turtle and draws the Pen
- speed(int x): sets the speed of the Turtle
- update();//takes in the coordinates from the controller and pops the last coordinate from the stack to call move
- setTurtleImage(Image img);
- setBackgroundImage(Image img)
- undo();//returns the line and the turtle to the last coordinate
- setBackgroundColor(Color c);
- setTurtleColor(Color c);
- getLanguage(): //gets the language the user prefers
- showHelp(): //displays the HTML help page
- displayAndMakeRecentComands()//makes buttons for the top 5 recent commands
- showGrid(boolean b)://determines whether or not GridLines should be displayed
- exportToXML(): export all the commands to the XML file

- uploadFromXML(): uploads commands from an XML file
- displayTurtleInfo(): displays the turtle's coordinates and other pertinent information.

Outward Facing:

sendCommands(String commands): Pass the commands from the textbox on the GUI to the controller for parsing.
updateTurtle(): Receive x,y coordinates from Controller()
handleCommand() Receive front end specific commands from the controller


File
Languages
Help
DrawGrid
Command:
Draw
Colors
Recent Commands
Turtle Info
Set Visible
Clear
Upload Image
Pen up
Pen Down
Undo
File
Export to XML
Upload from XML
(Drop-Down Menu)

Model API's

Inward Facing:

```
import java.util.List;

/**
 * Backend Turtle class which holds the most basic unit of movement (up/down/left/right,
rotations).
 * All other feature that need to be added/extended will take the turtle as an argument.  For
example,
 * the Turtle's "move" command takes in as its sole parameter, how much to move. If one
wanted to
 * specify a command that moved the turtle to a new location, they could do so by using the turtle
 * passed in as a parameter.
 *
 */
public class Turtle {
	private double myXPos;
	private double myYPos;
	private double myRotate;
	/**
	 * List that holds the set of X Coordinates of the Turtle
	 */
	private List<Double> myXPosList;
	/**
	 * List that holds the set of Y Coordinates of the Turtle
	 */
	private List<Double> myYPosList;

	/**
	 * List that holds the set of Orientations of the Turtle
	 */
	private List<Double> myRotateList;
	/**
	 * Turtle constructor that takes in an initial X and Y position with the orientation defaulted
to 0.
	 */
	public Turtle(double xPos, double yPos) {
		myXPos = xPos;
		myXPosList.add(myXPos);
		myYPos = yPos;
		myYPosList.add(myYPos);
	}
```

```
	/**
	 *	Alternate turtle constructor that takes in a starting position(X,Y) and orientation
(degrees).
	 */
	public Turtle(double xPos, double yPos, double rotatePos) {
		this(xPos, yPos);
		myRotate = rotatePos;
		myRotateList.add(myRotate);
	}

	/**
	 * Moves the turtle horizontally by the specified amount passed in.
	 * @param xIncrement
	 */
	public void moveHorizontal(double xIncrement) {
		myXPos+= xIncrement;
		myXPosList.add(myXPos);

	}

	/**
	 * Moves the turtle vertically by the specified amount passed in.
	 * @param yIncrement
	 */
	public void moveVertical(double yIncrement) {
		myYPos+= yIncrement;
		myYPosList.add(myYPos);
	}


	/**
	 * Move the turtle a specified amount in the horizontal and vertical
	 * directions.
	 * @param xIncrement
	 * @param yIncrement
	 */
	public void move(double xIncrement,  double yIncrement) {
		moveHorizontal(xIncrement);
		moveVertical(yIncrement);
	}

	/**
```

```
 * Rotates the turtle by the increment specified.
 * @param rotateIncrement
 */
public void rotate(double rotateIncrement) {
	myRotate+=rotateIncrement;
	myRotateList.add(rotateIncrement);
}

/**
 * Returns the list of x-coordinate positions of the turtle until now.
 */
public List<Double> getXPosList() {
	return myXPosList;
}

/**
 * Returns the list of y-coordinate positions of the turtle until now.
 */
public List<Double> getYPosList() {
	return myYPosList;
}

/**
 * Returns the list of orientations of the turtle until now.
 */
public List<Double> getRotateList() {
	return myRotateList;
}

/**
 * Returns current orientation rotation of the turtle.
 */
public double getRotate() {
	return myRotate;
}

/**
 * Returns current x-coordinate of the turtle.
 */
public double getXPos() {
	return myXPos;
}
```

```
	/**
	 *
	 * Returns current y-coordinate of the turtle.
	 */
	public double getYPos() {
		return myYPos;
	}
}
```

```
/**
 *
 * Interface for a Turtle command.  The update method will be filled-in by any Commands that
implement this interface.
 * returns the turtle object that can be manipulated by other commands.
 *
 */
public interface ICommand {

	public Turtle updateModel(Turtle turtle);
	public Turtle updateView(Turtle turtle);
}



import java.util.List;
import java.util.Stack;


public class Model {
	/**
	 * Stores list of all commands that program might execute
	 */
	private List<Command> myAllCommands;
	/**
	 * List of recent commands.
	 */
	private Stack<Command> myRecentCommands;
	/**
	 * Controller object with which Model interfaces.
	 */
	private MainController myController;

	/**
	 * Backend turtle object that performs manipulations of turtle movement
	 */
	private Turtle myTurtle;

	/**
	 * Model constructor takes in the Controller with which it needs to interface
	 * @param controller
	 */
```

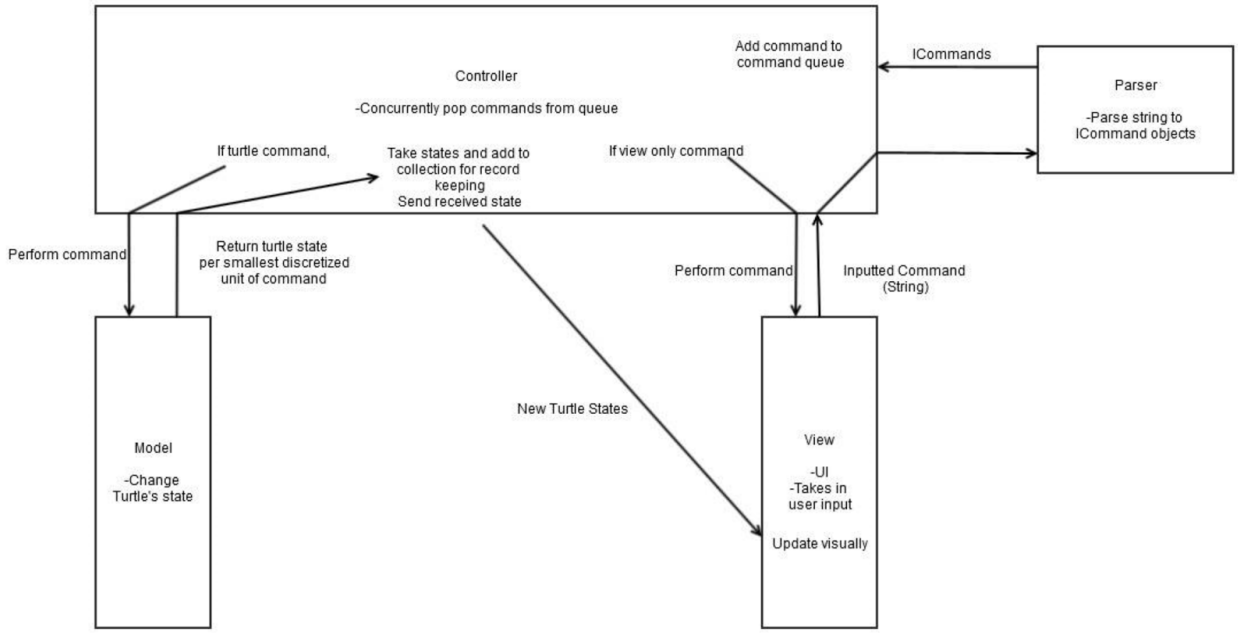
```
	public Model(MainController controller) {
		myController = controller;
		myTurtle = new Turtle(controller.xPos, controller.yPos);
	}

	/**
	 * Adds command to comprehensive list of commands
	 * @param command to be added
	 */
	public void addCommand(Command command) {
		myAllCommands.add(command);
	}

	/**
	 * Executes a given command and returns Turtle object
	 * @param command that needs to be executed
	 */
	public Turtle executeCommand(Command command) {

		for (Command c: myAllCommands)
			if (c.equals(command)) {
				c.update(myTurtle);
				myRecentCommands.add(command);
			}

		return myTurtle;
	}
}
```

Outward Facing:

```
public class MainController {

private CommandParser myParser;
private Node myView;
private Model myModel;

private BaseTurtle myTurtle;

public MainController(Node view){
        myView = view;
}

/**
* Receive commands from from View via string.
* @param enteredText Text input by user
*/
public void receiveCommand(String enteredText){

translateAndAddCommand(enteredText);
}

/**
* Initializes the model.
* @param initialTurtlePosition
* @param initialOrientation
*/
protected void initializeModel(int x, int y, Orientation initialOrientation, /*other params?*/){}

/**
* Send resulting turtle coordinates to view
*/
protected void sendTurtleResultsToView(){}


/**
* Does the next queued command for turtle
*/
protected void doNextTurtleCommand(){}

/**
* Pauses the turtle. Changes what coordinates are sent to front end.
```

```
	* @param stoppedPosition The stopped position on the front end. Takes into account lag.
	*/
	public void pause(int x, int y){}


	/**
	* Hard stops the turtle. Erases all coordinates and commands that it was supposed to do.
	*/
	public void stop(){}

	/**
	* If any errors happen on the back end, it should not crash the program
	* Sends the error back to View so that it can tell the user of the problem
	* @param ex exception that was thrown in back end
	*/
	protected void reportErrorToView(Exception ex /*potentially a separate error object that wraps around exception*/){}

	/**
	* Hard sets the Turtle's position and orientation. It will clear any queued commands and coordinates.
	* @param position New position
	* @param orientation New orientation
	*/
	public void hardSetTurtle(double x, double y, double orientationAngle){
	}

	/**
	* Hard sets just the position. Still clears any queued commands and coordinates
	* @param position New position
	*/
	public void hardSetTurtlePosition(double x, double y){}


	/**
	* Hard sets just the orientation. Still clears any queued commands and coordinates
	* @param orientation New orientation
	*/
	public void hardSetTurtleOrientation(double orientation){}

}
```

- Describe the program's core architecture (focus on behavior not state), including pictures of a UML diagram to describe the Model and "screen shots" of your intended View interface



Show actual "sequence of code" that implements the following use case:
The user types 'fd 50' in the command window, and sees the turtle move in the display window leaving a trail.

- **Example code:**

The user types in ‘fd 50’ then clicks the enter button or presses the enter key

This causes the GUI to call sendCommand() which passes the command to the controller

The controller takes the command and parses the input into an ICommand object. Since, this command requires the backend Model to perform a computation. This will get send to the Model, instead of the View.

The back end runs logic and calculations according to the command.

The controller calls the updateTurtle() method in the Model and receives the turtle object.

The controller calls move(int x, int y, double orientation) on the front end with the given Turtle object’s variables.

The front end draws the specified line based on extracting parameters from the Turtle object.

Write JUnit tests for your primary classes that show how you intend to create each class, call their primary public methods, and what return values you expect (and what their actual values should be).

GUI JUnit Test

testBoundaryControl(){
see if turtle is unable to move outside of boundary
}

testCommandHistory(){
input commands and see if history properly displays them
}

testUndo(){
get location of turtle
move the turtle
call undo
check current location against previous location to see if same
}

Test for MainController:

```
import org.junit.Test;


public class ControllerTest {

   @Test
   public void testThatNextCommandExecutes(){}

   @Test
   public void testPauseOccurs(){}

   @Test
   public void testModelInitializes(){}

   @Test
   public void testThatErrorPropagatesToView(){}

   @Test
   public void testThatHardSetOverridesAll(){}
```

```
    @Test
    public void testThatHardSetClearsHistory(){}

    private MainController initiateController(){
            Node view = new Node();
            return new MainController(view);
    }
}
```

Turtle Backend JUnit Test:

```
import static org.junit.Assert.*;

import org.junit.Test;

public class TestTurtle {

        @Test
        public void testMoveHorizontal() { };
        @Test
        public void testMoveVertical() { };

        @Test
        public void testMoveHorizontalVertical() {      };

        @Test
        public void testRotateValue() { };

}
```

- **Alternate Designs**

Model-View-Controller vs Model-View

In our planning meetings, one design that we entertained, but ultimately decided to the turn down was a Model-View interaction. Instead, we opted for a Model-View-Controller structure. In class, Professor Duvall had mentioned that a Controller was not a necessity for this project. We wanted to ensure that there was some behavior associated with a Controller. Merely having it pass information back-and-forth from the Model to the View, and vice versa, was not satisfactory in our opinions. We decided that the Controller would handle the parsing duties. The Controller would call either the Model or the View, based on whether the command required the View to perform some action (clear screen), or

the Model to perform some calculation (draw circle). It would also handle some basic error checking. In this way, we were able to add another layer of abstraction by taking out the parsing aspect from the Model (which was our original plan) and push it to the Controller. For the reasons stated above, we believe that the M-V-C design decision over the M-V is a better design choice, and would make the code more readable.

Roles:
Model -
Duke Kim: Controller, Resource Files, Resource Parsing
Scotty Shaw: Command Objects, Command Interface
Rahul Harikrishnan: Command Parsing, Command Factory, Backend

View:
Petra Ronald - Buttons and general layout
Jack Baskin - Menu bar, command line
Both - additional GUI functionality