Design goals

Frontend/UI (Yoon and Eirika)

The frontend of the program will consist of the Movement module, Image Updating module, Main JavaFX module, and the HTML Help module. The Movement module will handle the movement of the Turtle. It will receive information about where the turtle should go from the Turtle Command module in the backend and move the turtle accordingly. Moreover, it will communicate with the Image Updating module to show where to move the Turtle and where to draw the lines. In addition to showing the image for the turtle, the Image Updating module will also be in charge of setting the background color for the turtle's display area and setting the pen color for the turtle's lines. The Main JavaFX module will be in charge of setting up the Stage and Scene. It will also set up the JavaFX TextField where the user can type in commands. If the user input is valid (which means that it can be parsed by the XMLParser), the command will be displayed in a list of previous commands. The Main module will also set up a button that the user can click to launch the HTML formatted help page. Lastly, the HTML Help Page module will handle setting up the HTML formatted help page.

- Movement Module
    - Talks to the Image Updating Module to show where to move the Turtle and draw lines
- Image Updating Module
    - Displays the background
    - Displays
- Main JavaFX class
    - Sets up the stage and scene, plus an area to type in commands (or however the user would control the Turtle)
- HTML Help Module

Backend (Keng and Davis)

The backend of the program will consist of the Parser Module, the Turtle Math Module, and the Turtle Command Module. The Parser Module will parse commands which are received from the frontend as either a string or as a user-selected file. Once the commands are parsed, they are called via the Turtle Command Module. The Turtle Command Module can be thought of as a factory that creates different command objects. These command objects would be added to a queue where priority would be determined, as discussed below. Our design recognizes the difference between turtle commands and math operation, which is why we will have a Turtle Math Module which acts as a factory and similarly creates math operation objects based on individual math commands.

- Parser Module
    - Talks to Turtle Math and Turtle Command (maybe just Command)
- Turtle Math Module

    - Maybe only the Command Module should see and access this?
- Turtle Command Module (Factory)
    - Talks to the Movement Module to update turtle
    - Accepts commands from Parser
- Turtle Command Class
    - Subclassed into different commands (Forward, Back, Left, Right)

Primary Classes and Methods

Given the basic read-eval-print loop (REPL) that we would be using for our code, we aim to have a more interactive display of how code is being interpreted by the program. For example, one way we could is to highlight or color the background of the current set of code (or sentence) that is currently being processed, and then we would progress to the next chunk of frame. Overall, we would aim for more compact classes that might be numerous rather than a few clunky, verbose methods. As we would have both inward and outward facing application programming interfaces (APIs), we would strive to develop lean code that is able to manipulated in a variety of different scenarios.

The first class would be packaged under World, here it would contain all the classes that deal with environment. It would take in parameters such as the dimensions of the stage, how precise the movement commands would be (by determining the size of each individual grid in the environment, the color of the background.

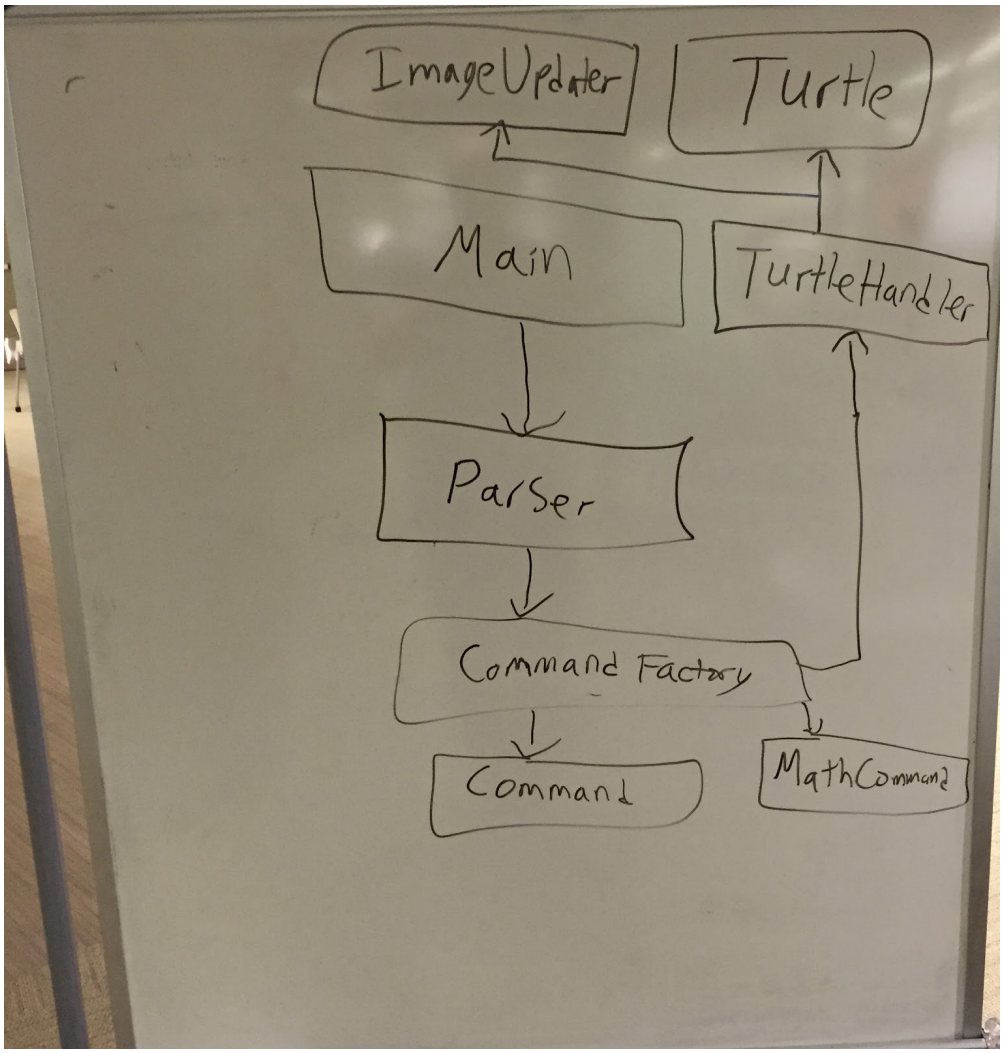
The next package would be unit, or the turtle itself. This package would interface with the parser which we would discuss later. This class can take in a specific number of parameters. It would include directional commands, we would keep it basic at the beginning by including only the four directional arrows as a starting point. It would also include any animations that we might wish to include, such as the legs of the turtle moving for example.

The next package would be the parser. The parser's job is to take an indefinite number of arguments and translate them to reasonable commands that can be acted upon by the unit, or the turtle in this case. It should have a relatively free-form syntax, e.g., it might not be important that a movement command is followed by a distance unit such as 'forward 50' but only that the two commands are adjacent. So, that means '50 forward' should be interpreted the same way. In the case where commands might be misinterpreted, i.e., when there might be overlapping or clashing commands, we must develop a system for taking in priority and order. So if we use the simple case of a chain, where it's first-in-first-out (FIFO) we can consider the preceding commands to take precedence over commands that follow it.

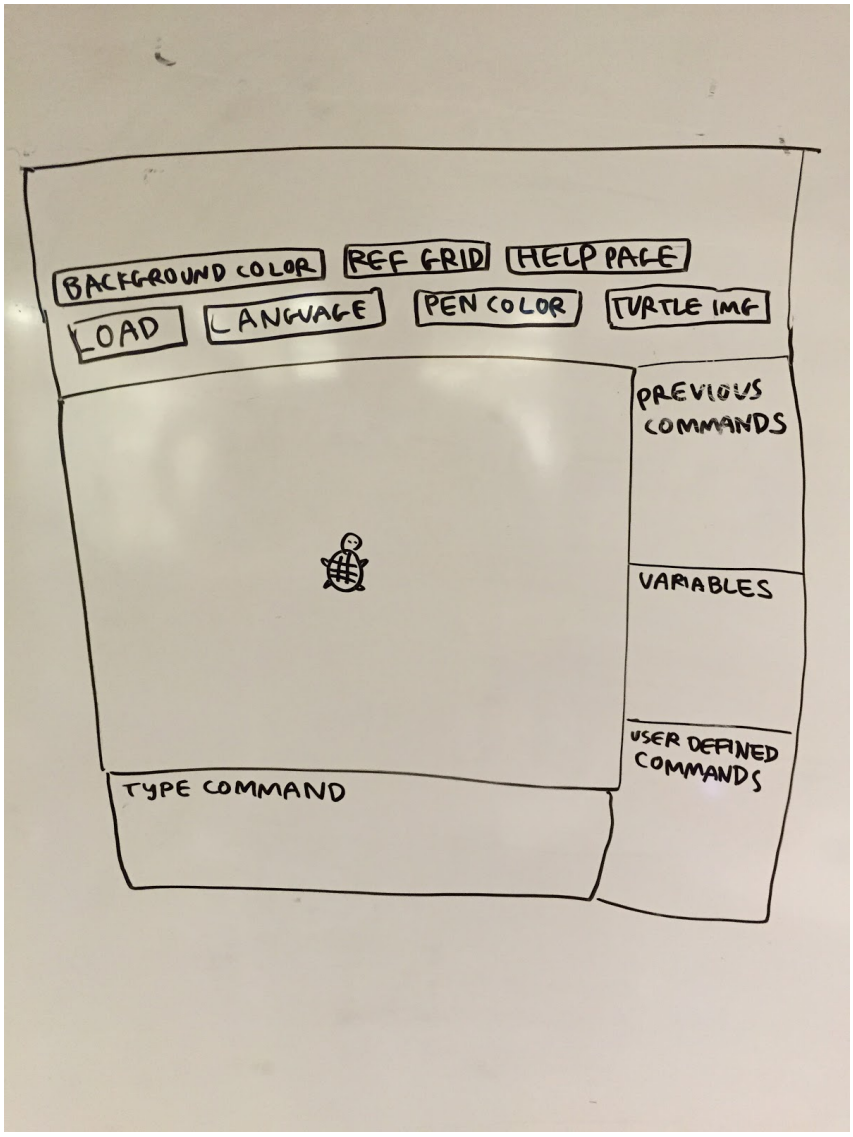
This would be a suitable opportunity for us to introduce a testing unit. With a basic corpus of commands in place, we can write unit tests that would simulate reasonable commands that our integrated development editor (IDE) would accept. We would then test again these basic level commands and see if they return the desired effect. As we progress, our testing class would encompass more robust evaluation such as multi-line commands that might control more than a single unit.

Underlying all of this is a common foundation that links the model and view. In this case, we advocate a Model View Controller (MVC) principle. The model would hold the basic, most essential of the environment and its various participants. The view is a visual representation to the end user that shows boundaries, movements and other elements that we wish to include. Lastly, the controller would represent the bulk of our code that would govern how commands are interpreted and when they evaluate.

UML Sketch



UI Sketch



Example Code

fd 50 example

First, the user types fd 50 into the TextField built in Main. Then, after pressing enter/clicking a button, Main's sendUserInput method will give the Parser the String to parse. The parser accepts the string via the parseCommand method and sends the command to the CommandFactory class where the command is added to the queue as a command object. Once it is determined that this is the only command to execute, the processQueuedCommands

method is called by the parser and, in turn, the executeCommand method is called on the command object. The result of the command gives a new turtle location, which is sent to the TurtleHandler via updateTurtleLocation at the new Point2D 50 pixels in front of the Turtle. After checking whether or not this is valid, the TurtleHandler will tell the Turtle to change its position to the new Point2D, then it will tell the ImageUpdater to updateTurtleImage at that new point and pass in an instance of the Turtle's Image to display. Also, the ImageUpdater will draw a line from the Turtle's original point to the end point.

Test Code

```
import static org.junit.Assert.*;
import javafx.geometry.Point2D;
import org.junit.Test;

public class TurtleTest {

	@Test
	public void testUpdatingTurtleLocation() {
		TurtleHandler turtleHandler = new TurtleHandler();
		ImageUpdater imageUpdater = new ImageUpdater();
		turtleHandler.setSceneSizes(250, 250);
		try {
			turtleHandler.updateTurtleLocation(new Point2D(100, 100));
		} catch (OutOfSceneException e1) {
			fail("Shouldn't throw an exception because 100, 100 is within 250 x
250");
		}

		try {
			turtleHandler.updateTurtleLocation(new Point2D(300, 300));
			fail("Should have thrown OutOfSceneException because the size is
only 250 x 250.");
		} catch (OutOfSceneException e) {
		}
		fail("Not yet implemented");
	}

	@Test
	public void testChangingTurtleImage() {
		Turtle testTurtle = new Turtle();
		try {
			testTurtle.updateImage("notARealFileLocation.gif");
			fail("Should have thrown an ImageNotFoundException because that file
location isn't real.");
		} catch (ImageNotFoundException e) {
		}
	}
}
```

```java
public class MathTests {

    @Test
    /**
     * TODO: extend this to test each math command subclass
     * doesn't work with the abstract class
     */
    public void testMathCommand () {
        double[] arguments = new double[0];
        MathCommand mathCommand = new MathCommand(arguments);
        assert(mathCommand.getResult() == 0.0);
    }

}
```

```java
public class CommandFactoryTest {

    @Test
    public void testForward () {
        CommandFactory commandFactory = new CommandFactory();
        commandFactory.addCommand("FD 50");
        commandFactory.addCommand("FD SUM 50 20");
        //assert that turtle position is forward 120 from start
    }

    @Test
    public void testBack () {
        CommandFactory commandFactory = new CommandFactory();
        commandFactory.addCommand("BK 40");
        commandFactory.addCommand("BK 10");
        //assert that turtle position is back 50 from start
    }

    @Test
    public void testChainedCommand () {
        CommandFactory commandFactory = new CommandFactory();
        commandFactory.addCommand("fd sum 10 sum 10 sum 10 sum 20 20");
        //assert that turtle position is correct
    }

}
```

Alternate Design

Our initial design had the Parser responsible for creating the command objects. Based on the team's experience with the last project, this was decided against because of how much the parser would be responsible for. Instead, the team decided to use a factory class which leverages reflection to create instances of command objects. This allows the parser to focus on other various components, such as variables in the input file.

Roles

Frontend: Yoon & Eirika
Backend: Davis & Keng

Class/Module	**Responsible Team Member**
Main	Primary: Yoon Secondary: Eirika
Help Page	Primary: Yoon Secondary: Eirika
ImageUpdater	Primary: Eirika Secondary: Yoon
TurtleHandler	Primary: Eirika Secondary: Yoon
Turtle	Primary: Eirika Secondary: Yoon
Parser	Primary: Keng Secondary: Davis
CommandFactory	Primary: Davis Secondary: Keng
Command	Primary: Davis Secondary: Keng
MathCommand	Primary: Keng Secondary: Davis

As shown in the table above, each team member will take 2-3 primary roles and 2-3 secondary roles in coding the 9 classes of the program. The primary is responsible for leading

and completing the codes, and the secondary is responsible for collaborating with the primary and providing help if needed. Our team will work together in group meetings as well as primary-secondary meetings throughout the next 3-4 weeks.