Steven Pierre

Time Review

- We spent over 60 hours working on the project. We started working on the project on September 29th 2014 and we ended on October 24th. We had three meetings as a large group and several other meetings as two separate groups.
- In our group we divided the front end and the back end. Half of us worked on the front end, the other half worked on the back end. The time spent coding could be broken down as follows.

  - 10% Coding new features
  - 20% Refactoring
  - 20% Testing
  - 5% Reading
  - 20% Designing
  - 20% Debugging
  - 5% Documenting
  - 100% Meeting with Team Members (the one with whom I worked on the Front end)
  - 10% Meeting as a full team.

- I personally wrote my code by first writing the name of the classes that we knew from the beginning that our design would utilize. Then we added methods solely by name of the commands we thought we needed to have to have my turtle moving in the scream. Then we added the code that performed the specific the methods said they would. After each method proved to work, I pushed the code online. I pushed my code every time I added new code that did not break the rest of the code.
- The tasks that were the easiest were to add buttons/functionalities to the GUI. It was easy to add the text area, the text boxes, and all the other functionalities to the GUI. However it was much harder to create the GUI itself, coming up with the layout and the style was hard to design and implement. I had a hard time figuring out what type of

pane was ideal to add to our code. I had to figure out the advantage and disadvantages of different types of Panes in order to figure out which one suited our design the most.

Teamwork

- We spent around an hour coming up with the design of the large project as a whole. We decided to keep the back end and front end separated, but connected by our controller. In that way we could split and work on our separate sides without any hiccups. We kept to that design all the way to the end, occasionally asking the other group about what they expect to receive from the controller and what they planned on passing to the controller. Then as a subgroup we met almost every other day to work on how to design the layout of our code and implement the features that we planned on having from the beginning.
- Our team consisted of four members. Two of us were responsible for the back end, and the other two were responsible for the front end. On the front end we split up the tasks as we went along. I build the layout of the GUI while my partner build the turtle class. Then I added a toolbar, buttons on the left side, a resource bundle, and the textArea and hbox while my partner coded and research the functionalities for the buttons. On the back end I am not sure as to how they separated the tasks, but the backend as a whole had to parse the string they received then they had to perform the calculation indicated by the command. We collected that information from the front end after them.
- The communication was just right with the outlets of the social network sites which proved useful when we needed to communicate with our teammates. We asked questions directly to our teammates when we needed answer, and we got the answers fairly fast. The questions were never too much. However, one team member failed to respond to some of our messages and would often go on missing for a couple days, without us hearing from them. But the rest of the group communicated without any problems.

- The plan was for us to complete both sides of the assignment, back end and front end up two days before the parts of the assignments were due. Then once we met we made sure that the code worked congruently as a whole so that we could make sure that everything worked well. We also decided to limit most of the classes to just one person to start, that way we could make sure we had no merge conflicts and that our code was running well with no problems encountered on its path.
- The plan held up well to the extension. We did not have to rebuild our code in order to accommodate for the extension. We simply needed to add some extra functionalities and modify some others. All in all, our approach accounted for the extra features that we had to implement. We did not arrive at a deadlock because our code was not flexible enough to accommodate some extra features. In the other hand the addition of the extra functionalities was one of the easiest part of the assignment.

Commits

- I pushed code 20 times over the course of the project. The average size of the push were around 100 lines of code. Each push usually were the result of the refined implementation of added methods.
- Yes, I think the messages accurately represent most of my contribution to the project. Some of the messages could have been a little bit more thorough, but for the most part most of the code was written exactly as it should have been.
- “Added saving capabilities” this commit consisted of adding the functionality of saving the users preferences. This commit did not cause any merge conflict because we added a brand new class that took care of saving and reading the files that saved the details regarding the users preferences.
- “Added file menu” this commit consisted of adding a file tab on the toolbar of our GUI which allows to save and write. This commit did cause a merge conflict because the class, in which the functionality was called, was changed. Yes the commit was done at the proper time as an extension to our code at the very end.

- “Added key movement” this code allowed the turtle to be moved using the key commands. No this commit did not have any merge conflicts with the rest of the code. Yes the commit was done in a timely manner relative to the rest of the team it was added when the extension functionalities were being added.

Conclusions

- We underestimated the size of this project. We started with a good foundation and good plan however we did not allocate enough time to go through with our plans. We could estimate better in the future by basing our estimation on part of the code being written before hand and evaluating from there as oppose to before we started coding.
- I took on the responsibilities that I could. I was always working on something when we met as a group. I probably could have worked on more if it came to that but everything was kept within their own boundaries. Yes I kept my team informed about my progress as well as kept track of the status of their progress.
- The GUI’s main renderer class required the most editing. It was the class in which we needed to add all extra functionalities or buttons. Just by the nature of what this class had to do it had to be lengthy and as a result we had plenty to add to it as our functionalities grew and other stuff grew with it.
- To be a better designer I should start planning my code a lot more ahead of time. Too many times we did not know the specifics of a method and we ended up with multiple lines of code that did not do anything productive.
- To be a better teammate I could communicate better with my fellow teammates and not wait for them to ask me for my help but to help whenever I am free to help them out or ready and capable of doing exactly that.
- If I could work on one part it would be on the layout of the GUI. We used a borderPane but that proved to be problematic when we wanted to add new tabs to our design. I would try to implement the tabPane instead and see if it improves the extendibility of our code and if we could add more tabs in a much easier manner that way.

Steven Pierre

Design Review

- **Status**

    - Yes, the code is generally speaking consistent in its layout and naming conventions. For example, we created packages that separated each section of our code respectively. A package for GUI, Controller, commands, compiler, dictionary, properties, and simulation objects. The packages contain exactly the information for their respective commands. The classes are also named appropriately like main for starting the program, MainController for the central controller of the program. In order words, their names and style are all within the conventions of Java.
    - Yes, the code is generally readable. We wrote all the classes to represent their exact purpose. The methods were also written to perform the method they were written as. In other words, all the methods were written within convention of what we expected the methods to do.
    - The dependencies in the code are fairly minimal. We can change most of the code without affecting the other parts within it. Most functions that were complex were self-contained in their own class. A good example of a class with good dependencies is the SymbolTableEntry class in the compiler package. This package has the structure that is exemplary of how we dealt with dependencies. The getters and setters, the hierarchy of the code. Global variables are exemplary in the Node class in the compiler package. The type requirements can be found in a class such as the TokenFinder in the compiler package.
    - Features are easy to extend in the front end, which were my teammate and I focused most of our attention. If we need to add another button we just have to call the .add function that is built in in Java. Most of the functions are very easy to extend. The only thing that was hard to implement was the implementation of having different workspaces. We did not have a good design going in to permit adding extra workspaces. Our code can be changed to add extra workspaces but

the way, our implementation method was not very well designed. We learn afterwards of the TabPane which made more sense in terms of design.

- It would be easy to test the parser and some other back end methods. The GUI in the other hand is very hard to test for. There no real methods that return values since they are all use to display images on the screen. The methods in the front end are not really suited for testing.
- There is an error in the back end code with regards to how the lines are returned when the pen is on. The back end returns object that they thought would appear in the scene simply by their specifications. The problem was remedied by re-creating the lines in the front end based on the data provided by the line returned from the back end.
- The MainController class in the controller package is in my opinion the most important class in the program. I serves as the bridge between the front end and the back end. It is the only way they can communicate which in turn keeps both parts of the code separated. It is a pretty good code. I would recommend that he adds more comment to his code to explain what it does, I would recommend that he removes some of the code that does nothing. To make the code reusable you would need to change the type of input it gets, the class it references to do the back end calculation, and the name of the methods.
- The SymbolTableEntry, which is in the compiler class, is another example of a well written code. The code is good it works as expected. I would recommend that the author adds more comments to explain what the information returned will be used for. To make this code reusable you would need to just add some more global variables and change the name of some of the methods.
- The TurtleView class in the view package is a well written class. It has all the functions needed in order to move an object in the GUI. There is a good amount of comments written in the code. To make this code reusable you would need to do nothing to it. You can pretty much use it in a any other program.

Steven Pierre

- **Design**

    - Currently the program is designed so that the front end and back end are separate from each other. The front end launches the GUI. The GUI will take any user input which it will in turn pass to the controller. The controller is the step in between the front end and the back end. It can pass in strings to the compiler, and it can take the turtle from the compiler. It tells the program when to perform its calculations. The back end takes the string input from the controller, parses the input from the dictionary and then performs the calculation passed in by the user. These three sections of the code work in unison to execute the commands that come from the main user.
    -
    - To add new commands to the code, you can do so simply by adding more commands in the library and adding a method for the command in the CommanList class in the commands package. For example, below is the current library list:

```
#
# Turtle Commands
#
Forward = forward,fd
Backward = back,bk
Left = left,lt
Right = right,rt
SetHeading = setheading,seth
SetTowards = towards
SetPosition = setxy
PenDown = pendown,pd
PenUp = penup,pu
ShowTurtle = showturtle,st
```

HideTurtle = hideturtle,ht

Home = home

ClearScreen = clearscreen,cs

So you would simply need to add the name of the class above with the same format. The name on the left has to be the name of the method you will write in the CommandList class. On the right, you will write the command as the user will input it as a string. The second part consist of adding a method with the proper name that will perform the task that you want the command to perform. Like the command below:

```
public static Command<Turtle, Void> penDownP() {
   return new Command<Turtle, Void>() {
        @Override
        public String run(Turtle params) {
           return String.valueOf(params.getPen().getActive());
        }
};
```

}

The front end to add a new component you can just create it and decide which side you want to put it in. The methods are written to tell you which sides the object will appear on. To add anything on the GUI you just have to add code as follows in the GUIScene class:

```
load.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle (ActionEvent event) {
            try {
                layout.setStyle(saver.read().get(0));
```

```
            } catch (IOException e) {
            }
        }

            });
```

- My code is the front end. We have the launcher which we called the GUIStart, its sole purpose is to call GUIScene and start it going. GUIScene in the other hand will create the visuals. There is a class for some of the most versatile parts of the GUI such as the turtle, and the pen. We create those objects and call on them, then we add them to the GUI as we see fit.
- Our code is designed that way because we want to keep everything separated we want to make sure that our code remains hierarchal. The design worked extremely well. We had almost no headaches in order to add extra functionalities to the GUI. The only problem was to run and create the GUI in the same Class, we should have kept it in a different class, and we could have used some different Panes that are far more hierarchal, like the TabPane. The only assumption in our code was that the user would not go fullScreen because JavaFx tends to resize things and sometimes the borders or the grid would get distorted as a result, for that reason, it was unreliable.
- The ability to change the Turtle's image stems from the GUIScene. We first added the button in the GUI, then we added an event handler that references the Buttons class. The Buttons class was written to have all the Event Handler 's function that the buttons needed to perform. You can always add one more Button because the FlowPane will resize to accommodate all of the buttons. The only assumption in terms of flexibility is that the person using the program would need to know how to code, knowing how to would allow them to add extra functionalities.
- The ability to save your preferences is a good example of flexibility. First you need to add the button that will give the user the option to do exactly that. Then

you must write the class that will save a set of preferences. In our build, we elected to save the background information in a txt file. Then we added another button which allowed the user to choose their save file. It is pretty clear how to extend the file format you can save the file as. No assumptions were made in creating that class. It takes care of everything by itself.

- The MainController class is one of the most vital classes for the code. When you call on it the turtle is first initialized. Then when the user puts in their input, it is passed in via pasInput method. The passInput takes in the string and passes it through the back end via the compiler. The compiler takes care of figuring which command the user is trying to run. The turtle is then updated as a result to match the command. The last two turtles return the new turtle to the front end, and a history list to the front end as well.

- **Alternate Designs**

  - Our original design handled the extensions really well. We implemented all of the extension to which we knew how to implement them. The one we did not put in were the one we did not overtly know how to implement them. So yes the original design was a good enough design to add extra functionalities and was very flexible.
  - The original API did not change much over the course of the project. We added functionality and buttons to it but we did not change the layout or the basic ordering of it. We mostly added new functionalities and new link where we saw fit and where we needed. The API was the easiest part to extend in the code/
  - One design choice that was discussed was how to parse through the strings given by the user. At first we thought that we would use a map to get to the methods we needed to perform the user command. However we quickly realized that it would be a slow program and a very poor design. Instead we opted to use a tree

in order to reference which command we would be using. The stack made it easier to traverse and sort through multiple commands. With the tree we could take in an infinite amount of commands at all times.

- Another design choice was the use of the borderPane. We first started by using the GridPane but we decided that the structure of the BorderPane would make for a more rigid design that would make it easier to keep track of all the different sections of the GUI. We also thought that it would allow us to have toolbar. It was easier in order to achieve the layout we were striving for. The trade-offs were that he BorderPane cannot be extended much as oppose to the GridPane which we could add more cells to. The other tradeoff was that the GridPane could take the form we needed for the GUI but that it also required that calculated the layout more.
- At last another design choice we made as a group was how to manage the creation of lines. We could create only one line after a user input, while the back end would sometime need to move a lot of times like in the case of loops. We came to a nice compromise where the back end would keep track of the creation of all the lines that need to be created, while the front end would receive an ArrayList with the information needed to make the lines, and the Lines would be made in the front end.
- One bug that remains in our code pertains to the turtle moving on arrow key pressed. The problem with that is that we need to make the scene active before it can respond to the user input. In order word we need click on the turtle area, or pass in an input, or click a button before the turtle becomes responsive to the key movement. But once that is done, everything works well.
- Another bug that is still in the program is that we have not been able to properly add several workspaces. The problem resides with some of our early design decision. The answer would ask for us to use the TabPane and to tangle a little with the GUIStart class and how and when it references the GUIScene class.

- The last bug that still remains in the project is the fact that we cannot yet save the different preferences for different workspaces. We need to come with a better manner of keeping track of which workspace is being used by creating a map indexing the preferences of one workspace to the name of the workspace.

In addition to the code

I think the GUIScene, Buttons, ReadAndWrite, and MainController class are well written in my code. I like these classes because they were all created at different point and worked well with each other without causing any problems in terms of joining them. For example, GUIScene is well designed, it creates the layout for the GUI. You can always add more functionality to GUIScene without having any redesign of the code. In order to make GUIScene, more legible we created the button class which performs the function of each buttons added to the GUIScene. It simply creates a place where the functionalities of GUIScene can be extended. ReadAndWrite does something similar but it is secluded because it is there for saving and loading which are much easier to refactor in a class of their own. Then the highlight is the MainController class. It was the first class that was constructed, it is the bridge between the front end and the back end of the code. It can be extended if ever needed to be.

I would test the GUIScene to make sure that all the buttons are there. Given that this is a visual part of the code I would run it and see if the functionalities are there. I would test how our trees work in the back by using assertEquals and passing in inputs and expecting the user commands result back when I get them.

I decided to test whether my code gets the proper movement from the keyEvent using Junit. I created the class MovementTester whose sole purpose is to check on the status of the movement resulting from the key handlers. I want to make sure the keys are passed in automatically when I press them. We had an error in our code that stopped us from being able to user the key event as we wanted initially.