Project Journal

Time Review

I would say that I worked about fifty to sixty hours total on this project. We met mostly every night and worked about two to four hours each night. Some of the nights, especially towards the end, we worked maybe four to seven hours each night. I started on this project the night of the day it was assigned and ended this project early in the morning of the day it was due.

At the beginning our time was spent planning and writing the API. As features needed to be implemented, I spent my time pair programming with Chris Bernt on the front-end. This involved planning for the design and approach for implementation, coding, testing using the GUI, and refactoring as needed. Towards the end of our project, we were still adding a few new features, but we had also spent time making some minor refactoring and helping the design of our back-end. On the last night, we mostly did just documentation.

I did not really personally manage my code seeing as how I was pair programming. We, as a sub team for the front-end, decided to have one package at the beginning and add more as needed. Often, we start with the easiest of features and progress to the harder ones – for example, changing the color of the background was done much earlier than showing the variables. Additionally, we would wait for the back-end to implement some of their key features, such as variables and user-defined commands, before we would put it on the GUI because we needed that information in order to present it.

We would often use the GUI to test our implemented features instead of writing JUnit tests because it was easier to visually see if something was working. We would also push every time we got a new feature or set of features to work, which would happen a few times each night we met.

Most of this project was easy for me since we worked at it a bit at a time. The hardest part was helping our back-end improve on the criticisms made in class because it would take me a fair amount of time to understand their code. Refactoring on our part was fairly easy though, because we had a general idea of how to do it when we wrote the poorer code. One other hard part that we spent a lot of time on was whether or not to redesign our Turtle class, which will be discussed later.

Teamwork

The team spent about seven to ten hours planning design, mostly in the beginning of the project. One or two of the hours planning were spent after the back-end code was criticized in class. Chris and I worked on the front-end and each had a part in writing every class we were responsible. The back-end was dealt with by Mike and Meji but I am not sure who worked on what class. Communication in the team was fairly good though at some important time (like at the end of the project) there were a few hours where communication was much needed but that can be forgiven.

The team's plan for this project was decided very early on and we kept it that way. This worked out well and we did not run into any problems, even after the extended implementations

were given. If we had to wait for the back-end in order to do a feature, we would simply just work on another feature (of which we had plenty to implement) or do some minor refactoring.

Commits

I personally pushed 31 commits worth 3,482 additions and 2,385 deletions. The message of my commits are accurate of those commits but not of my work contributed to this project. One small reason for this is because some of my commits were to fix merge conflicts. Another much bigger reason is that I was pair programming with Chris. If you combine all our commits together, then you would get an accurate representation of what we did as a single sub team with each member contributing to the other's work.

One of my commit messages reads "Added help page functionality." This was to show that a new feature was implemented. One thing to note is that this was "due" in the basic implementation but was actually implemented after that deadline. We figured that this is actually due at the "complete" implementation deadline and that it was fine to wait to actually implement it. This commit did not, fortunately, cause a merge conflict for others.

Another one of my commit messages reads "Added functionality of being about to graphically toggle activeness of turtles." Though this message does not fully make sense (it was late), this was to show that a new feature was added. This commit was done timely, a few days before the deadline. This commit did not, fortunately, cause a merge conflict for others.

One final commit message I will mention reads "Documentation for frontend.abstractFeatures package." This is simply showing that some documentation was done. This was done timely according to me because I believe documentation should happen at the end of a project in order to reflect on what you have done. This commit did not, fortunately, cause a merge conflict for others.

Conclusions

I did not either over- or underestimate the size of this project; however, I will say that I felt like there was a lot to do on the front-end. I also took enough responsibility on this project: Chris and I worked well together on the front-end. The back-end could've helped us on the back-end part of the turtle but we ended up doing it because we needed to test our implemented features. Additionally, Chris and I kept each other very well informed about what we did on the front-end. The back-end was also kept informed about what we on the front-end did but they did not seem to really need the information always.

The part of the code that took the most editing was the back-end. It was poorly designed initially. On the front-end, the most edited part of the code was the Turtle class. We extracted a fair amount of code out of it to make more abstractions and fix some bugs. One example of this is the Pen class. Originally its functionality was in the Turtle class but we saw that there was a small bug: all the turtles had the same color, style, and width for their pens. We made all these edits to improve functionality and design.

In order to be a better designer, I should keep reading the material assigned and deeply thinking about design before doing any coding. I do not believe there is anything I should stop or start doing.

In order to be a better teammate, I should keep communicating, thinking about design, and helping out. However, I should start helping other sub-teams out more if they really need it. There is really nothing that I think I should stop doing.

One thing I would do to improve my grade on this project is to implement some more features – both on the front- and the back-end. There are some features we did not fully implement due to a time constraint but we at least took an elementary pass at parts of those features.

Design Review

Status

The code in the front-end is fairly consistent though there are some spacing and formatting issues. Names, however, were thought of by both Chris and I, making them fairly consistent. However, there is stark difference between the front- and the back-end in practically everything: naming, spacing, formatting, and documentation. This difference also applies to the readability I believe – the back-end is somewhat scattered and hard to read whereas the front-end is fairly organized and pretty readable. However, this may come off as bias since I worked extensively on the front-end.

There are a lot of natural dependencies in our code: things are passed between objects and they talk back and forth and whatnot. However, there is one important and annoying unnatural dependency in the front-end: the GUIFeatures that are implemented are instantiated in a method instead of in the constructor or as instance variables. This is because some of the parameters passed to their constructors are dependent on other objects that are created like the SLogoCanvas and its size. This is really annoying but we needed to do this in order to design the GUI the way we wanted to. Another unnatural dependency is that the TurtleCollection class has some getters and setters like getLastActiveTurtle() and setActiveTurtles(); however, we decided to make these because they are needed: many instructions are dependent on the last active turtle on the screen. Additionally, the ask, tell, and askwith commands need to set the active turtles and we decided that it was better that the TurtleCollection class took in the indices of what turtles need to be active and handle the operation itself.

Features and new instructions are easy to extend or implement if they are simple. The only reason for why such an addition should take more than ten minutes is if the logic behind the new feature is complex itself. New GUI features take one extra class being made and one line in the SLogoWorkspace class. New instructions need just a new class to be coded. Simple and easy.

Most features could be tested for correctness on a preliminary pass by just looking at the GUI and running the program. However, it is hard to test things using JUnit tests because I am simply not well-acquainted with them. It is especially hard to use JUnit tests on the frontend because if we were to use them, we would basically be testing Java. I discuss this later on.

I found a fair amount of bugs in our code, most of which were in the front-end code Chris and I worked on. However, one specific bug I found in the back-end was that the variables, user-defined functions, and history were shared across workspaces. This is due to the fact that the data structures that hold this information are based on the singleton design. I pointed out this bug to the back-end but they did not have time to fix it.

One class that I did not write but seemed interesting was the Parser class. It is pretty simple: one public method that takes in a string, splits it based on newline characters, sends it to the ExpressionTreeBuilder in order to get an output list of instructions/commands/ExpressionNodes based on that string. Then it adds that output list to its own list of ExpressionNodes that it will pass to the front-end. I really like this code and would tell the author that they did a good job on it because it is simple and easy to understand. It also seems to act as a mediator, which is a good design pattern. I believe that this class could be used for other programs that need parsers but the meat of the parse() method would need to be rewritten in order to suit the program that is using them. This rewrite is needed because this class delegates operations to classes that can handle the operation and different programs will have their own classes for their own operations.

Another class I did not write but wish to talk about is the UnrecognizedFunction class. This seems like a null object but for slogo commands. It has two constructors that both call the superclass to set the numberOfChildren equal to 0. One of the constructors takes in a string to set as an identifier of sorts. I am not really sure what this identifier is for but it has its own getter method so it must be important. Also, I do not think that this class could be extended to other project because it seems pretty specific to slogo. I would tell the author of this class that it is good but that it needs some documentation to understand why there are some things like the identifier string.

One final class I would like to talk about but did not write is the ArcTangent class. This is an incredibly simple class. It has one constructor that calls its super constructor with an argument of 1 in order to set the numberOfChildren to 1. It also has one method, evaluate(), that simply returns a double which is equal to Math.toDegrees(Math.atan(theLeftNode.evaluate())). This actually seems a little confusing at first but if you follow it carefully, it is easy to understand: evaluate the left node and then return the degree value of the arctangent of the evaluated node. Also the class is short and simple: basically no real code at all. This shows that the inheritance hierarchy is used well because it reference protected methods and instance variables. I would say to the author that their code was good here: it is clear and simple. However, it could use some documentation.

Design

In our design the front-end get the commands from the user, translates them, gives them to the mediator, which then passes it to the back-end to parse and evaluate, which creates a list of basic instructions (like move and rotate), passes this list to the mediator which them loops over the list to call the front-end to adjust accordingly. Additionally, the front-end handles some buttons and other tools to make things happen on the front-end like turning the grid on or off and change the turtle image. These buttons and other tools are within an inheritance hierarchy of front-end features. Also on the front-end are a turtle and a data structure of all turtles but that only iterates over all active turtles.

New instructions are simple and easy to implement, especially if they are simple. The only reason for why such an addition should take more than ten minutes is if the logic behind the new feature is complex itself. New instructions need just a new class to be coded. Simple and easy.

Our design has some pros and cons. One good part of our design is the iterator in the TurtleCollection class. This allows other objects to access a data structure that contains all the turtles but not manipulate the actual list directly, thus giving some flexibility but also some protection. The InputController class is also good design, as it is a mediator between the front- and back-end. This allows separation that is needed but allow not any duplication like similarly named methods. It also helps to keep the size down of an otherwise big class that would have existed. Another good part of our design is our front-end feature inheritance hierarchy. This hierarchy makes it extremely easy to add front-end features – just create the class needed and add a line in the SLogoWorkspace class! This is similar to another good piece of design in our project: the instruction hierarchy. This hierarchy also makes it easy to add new instructions (reference above).

One major design flaw we had was that instructions are passed to the back- and front-end in bunches instead of singularly. This caused most of the bugs in our project: ask and askwith did not full work, turtle queries would only be based on the last active turtle's information but that was only before any bunch of instructions is inputted. These are major flaws that would have been fixed if we passed instructions one at a time.

One possible design flaw is that the Turtle class is quite large and does not separate the front- and back-end. These things should be normally separated but in the case of our Turtle class, the front- and the back-end are highly dependent on each other. This caused us to keep the Turtle as one class instead of possibly three. This alternative is discussed below.

One feature I did not implement was how to save and load variables and user-defined functions. The classes need to do this are CustomReaderWriter, FileWriter, and CustomParameters. Nothing is really closed – all methods are public and only the instance variables are private. Most of the methods however, have the good functional design of taking in nothing and giving back nothing. I do not really understand how to extend this feature or code – it is quite unclear but it may not necessarily need to be extended.

One feature that I did implement is the toggling of gridlines on the canvas. To do this all you need is the SLogoCanvas and ToggleGridLines classes. The ToggleGridLines class simply places the button on the GUI and the accompanying handler as well. This class is also a part of an inheritance hierarchy of GUIFeatures as mentioned previously. The SLogoCanvas class is what actually creates and toggles the gridlines. Also in this class, one can change the grid square size by simply changing one constant. This is a sign of extensibility on this feature.

Another feature I implemented is the "add new workspace" feature. The only classes needed for this is the SLogoWorkspace class and its nested AddWorkspaceButton class. This class is nested so that I could use one of the SLogoWorkspace methods without passing the button the entire SLogoWorkspace class. Right now there is one bug with this feature: one cannot add more than ten workspaces. I believe this is due to some crazy, unknown JavaFX stuff. The only extendibility for this feature that I could think about is if one wanted to add more than one workspace at a time of a button click. To do this, all you would have to do is call createTab() more than once in the action() method of the AddWorkspaceButton class. Another extension can be if you wanted the GUIFeature to not be a button but something else like a drop-down menu. To do this, simply extend another GUIFeature abstract class like SLogoDropDown.

Alternate Design

Our design handled the extensions fairly well. It was very simple to add either an instruction or a front-end feature. If it really took more than ten minutes to add either of these, it's because the actual logic behind the new feature is somewhat complex and requires something more. Take, for example, the new feature of saving workspace preferences. It is easy to add the button to the front-end but to take all the information form all the objects on the screen takes some extra work. Another example is the ask command: it was rather easy to change a list of active turtles once but to do reset the active turtles list to what it was before the ask command is more difficult because in our design, it is hard to figure out when a single command ends.

The original API did change but it only really grew. We added more classes and public methods as new features and abstractions were made. If we were to actually rewrite the API now, it would look differently but the basic underlying concept of the interactions would all be the same.

One design decision we made was to get all the instructions that are given at one time, interpret them, pass it to the back-end, change the necessary data, and then hand it all back to the front-end to display graphically. One advantage to this is that ere is minimal talk between the back-end and the front-end. Another advantage is that some methods would not have to be repeatedly called, which is inefficient. One disadvantage is that we would never be able to know when a single command is finished running, which causes a bug in our program and I discuss later. One alternative design would have been to pass one command at a time, which is what I now prefer to do, even though it may be less efficient. If we chose the alternative, we would have been able to keep the functionality we had as well as rid our bug.

Another design decision we made was to have the Turtle class be one class that handle both the front- and back-end for the turtle. We decided on this because there was a high, pretty much almost, one-to-one interaction relationship between the front- and back-end. We tried to separate the front- and back-end and show how that would have looked in the BothTurtle class, which would have been a mediator between the front- and back-end. An advantage to the way we did it is that all the code is together yet that also leads to its disadvantage of it being a big class with a lot of responsibility. However, I still prefer the way we did it because it looks better than the BothTurtle class.

One final design decision I will mention is that we decided to keep each instruction as one new class. The advantage to this is that all you have to do in order to add an instruction is to simply make a new class for that instruction and implement up to three methods (but usually not even that many!). One disadvantage is that both the front- and back-end have to work on the new instruction class but that is still separated by methods at least. One alternative design would have been to separate the front- and back-end and have a mediator between the two. However, I prefer our design because of the ease of addition that it creates.

One bug in our program is that turtle wraparound does not fully work, especially for large distances where the turtle moves past multiple border (be it either x or y or both). A way to test this is to simply run the problem – it is visually easy to see. To fix this, we would probably add some more if statements in our EdgeHandler class for these regions and scenario but more

thought must go into it. We felt that it would be better to implement some other features than to spend our time perfecting this small feature.

Another bug is that user-defined commands are not fully implemented – they cannot take in any parameter arguments; however, you can still make a static, constant function at least. I do not really know how to fix this bug since it is more of a back-end issue.

One final bug that I will mention is that history, variables, and user-defined commands are shared across workspaces. This is easy to see using our GUI: simply add some workspaces, add a variable and a function, update all the workspaces by running a simple command, and you will see that all I just mentioned. To fix this, we need to change the VariableNodeMap, FunctionNodeMap, and HistoryCollection classes from singletons to simply regular objects that can update (which is can already do). We mentioned this fix to the back-end team but we ran out of time.

Code Masterpiece

My code masterpiece shows off our inheritance hierarchy. It also incorporates the delegation design principle and the template method design pattern. I am showing off the main inheritance structure in the front-end: the GUIFeautre hierarchy. I will specifically show off the GUIFeature abstract super class, the SLogoButton and SLogoScrollPane abstract classes, and some examples of each of those abstract classes. I did some minor refactoring to get rid of one line of repeated code. I knew how to do this refactoring before the deadline but I ran out of time to implement it beforehand. These classes had gone through refactoring during this project but I was too tired to figure out how to take out the one extra line of code.

The abstract super classes have a template or basic implementation of methods that the subclasses can use but overwrite if need be. Additionally, the superclass may delegate some action to the subclass through some protected method that the subclasses have to implement.

Additionally, I am not including any JUnit tests. This is because I cannot really figure out what to test on the front-end that would not just be testing Java. I would like to say that I really need help on testing and test-driven design. The lectures on it did not really help me understand this topic. I would really like to learn about this because it seems important to the topic of design. It also seems practical in industry as well.