Justin Carrao

Computer Science 308

Slogo Analysis

**Project Journal**

**Time Review**

The project start date was the day after it was assigned. We finished it basically at the due date and time. Including time spent designing, reading, coding, debugging, etc. I would say that I spend roughly 100 hours on this project. In the beginning time was spent discussing how we wanted to design the project. Specifically we started with the division of code, classes, packages between FrontEnd and BackEnd. After we felt like we had a good delineation of code and responsibilities we split into two teams to work on it. I was on the BackEnd Team with Ashwin. In the beginning I spent a lot of time coding new features. Basically this means I was coding a bunch of node command classes and trying to implement the logic for them first. This turned out to be a mistake, we ought to have fully implemented the syntax tree until we were satisfied with it and then tackled the specific nodes of the tree, but we did it backwards unfortunately. In terms of managing and packaging my code, I would create new packages to house each of the “groups” of node types (commandNodes, booleanNodes, conditionalNodes, etc.) so that they would be easy to find. The specific implementations of the nodes might have been the easiest part of the project for me to code, but as a result of doing them first, the syntax tree was the hardest part to implement. My partner really wanted to implement the tree iteratively, instead of recursively. We talked and actually debated quite a bit about this, because I wanted to do it recursively, as did Carlos, our TA. However, ultimately the argument for doing it iteratively won out, and this resulted in headaches for both of us throughout the project.

**Teamwork**

Our team spent a relatively small amount of time up front discussing the overall design of the project. We discussed very broadly what we wanted the Front End and the Back End to be doing, and we also discussed the “connection point,” meaning the specific class that the Front End would use to pass the Back End its input and from which the Front End would receive said input. Beyond that, our design discussion was emergent. We discussed issues, problems and changes as they arose. This afforded us flexibility and made it easier to make progress, because we weren’t always able to meet as a group of four and code for hours at a time. The ability to communicate while apart about updates was helpful. However, I don’t think this process was ultimately the best approach was the right one. It wasn’t worth it, after all. Each of the small tweaks and changes that we made on the fly without sitting down as a group and fully enumerating all the consequences of said changes resulted in our design getting slightly more out of our control. We still had a pretty solid grasp on the overall idea and design of our project, but little details needed to be changed repeatedly, such as what data structures needed to be used where, and why. This was difficult to keep track of, since a relatively small amount of attention was paid to each one in turn. Ultimately our initial plan was solid, but with each tweak our overall design became less and less stable, to the detriment of the quality of the finished product. The biggest hold up for implementing the extensions was the implementation of our syntax tree. The iterative implementation made it really difficult to account for additions of different node and syntax types. Carlos, our TA, even said this would happen. That tree is by far my biggest regret of this project. The responsibilities were split like this: the Front End was handled by Greg Lyons and Rica Zhang, while the Back End was handled by me and Ashwin.

**Commits**

I submitted 66 times to Master. The average size was roughly 40 lines of code. Looking back over my commit messages tells me that my commit messages were very concise, maybe a little too concise. But I think I did a good job of explaining what I had changed or implement almost every time. I had explanations like “loop refactoring,” “repeat node works now,” “done with AST,” etc. I think these were reasonably informative so ultimately I think I did a good job here. I’m going to analyze these three messages. The first one just indicates that I rearranged and changed some code to make it more readable, more usable, etc. The second and third are examples of completing the implementation of some feature that was necessary to the completion of the project. Merge conflicts weren’t really an issue with my commits.

**Conclusions**

I think we underestimated the time this project would take. If we had implemented the syntax tree recursively, it would probably have taken a lot less time, so our initial estimation of the time needed would have been accurate, so a big part of correct estimation is approaching the design in the best possible way. That’s how we can estimate better in the future. I think I took a good amount of responsibility with the team. Again my biggest regret is not being more convincing with the idea of implementing the syntax tree recursively. I did a good job keeping the team informed. I definitely stayed in closer contact with Ashwin than Greg and Rica, because he was working on Back End with me. In retrospect I should have kept Greg and Rica more informed. The parts of my code requiring the most editing were definitely the tree and the repeat node. Our tree was constructed using a lot of special cases and that made it very difficult to follow the logic of getting the repeat node command to work within the tree.

To be a better designer, I should do even more design work up front. More than I think will be necessary. Clearly the past few times I’ve increased the amount of thought put into the design, but it feels as though it’s never enough. I think I should go above and beyond and spend a huge amount of time on it for the final project and beyond. I should stop accepting pieces of design that my TA warns me against if I want to implement well-designed code.

To be a better teammate, I should keep in better contact with all members of the group. This even includes those who are working on parts of the project that seem far removed from what I’m doing. I don’t think it ever hurts to know what’s going on throughout the project, even parts that I don’t specifically interact with.

If I could work on one part to improve my grade, it would be the syntax tree. It’s poorly designed, in my opinion. I don’t like its design and I think it will hurt us, so I would change that. This would make the project more extensible because each individual node would then be easier to implement.

**Design Review**

**Status**

I think the descriptiveness and layout of our project is consistent and easy to follow at a high level. I think names of classes, packages and functions are easy to follow and understand all the way through. We did a good job with that, in my opinion. Given this, I think much of the code is pretty readable. Some exceptions include some one-off functions in various classes (such as the RepeatNode) and the buildTree function is tough to read. I think the dependencies are pretty clear throughout the project except (again) with the exception of the syntax tree.

Review three classes that I didn’t refactor or write, in detail:

1. TurtleInformation.java extends VBox:

This class starts with four field variables: myTurtle, turtleIndexLabel, turtlePositionLabel, and turtleDirectionLabel. It’s constructor instantiates the three labels and then uses an update( ) method and calls a superclass method to add the labels to this.getChildren( ).

Methods specific to this class include:

emptyInfo( ) which sets the info in the labels to basically nothing

changeTurtle(turtle) sets myTurtle to turtle then calls update( )

update( ) which calls emptyInfo( ) if myTurtle == null.

otherwise it gives the labels relevant information using the turtle

truncate( double ) which changes the coordinate values of the turtle slightly when update puts that information into the labels.

I chose this class because it’s very easy to follow what it’s doing. It’s a great example of very readable code, in my opinion. The exception is the truncate( double ) method. It’s not terribly hard to read, but it’s just not entirely clear what it’s doing until you see it used. I would call it truncateCoordinate( ). To make it extensible it should take an object type instead of a turtle, with no more specificity than that. This would make it an effective class for storing information necessary to display something well on screen, no matter what kind of object it takes (not just turtles).

2. Controller.java implements Observer:

This class takes a View and a Model in its constructor and uses them to create a TurtleInformation object and a TurtleWindow object. Its executeCommand(command, activeTurtles) method is well named and straightforward. It uses the model to parse the user input and updates the view. The problem with this class is that executeCommand also calls the interpret (sceneUpdater, command) function. My problem with this is it’s not the clearest thing in the world. I mean I guess you would figure it out relatively quickly because it takes a command, but I think a simple fix to make it more *immediately* readable is the change the name to interpretCommand. Finally it has the update method that calls the executeCmomand method (which calls interpret) so this class is easy for others to use (one need only ever call Controller.update(observable, object).

3. CommandLine.java extends TextArea implements Observer:

This class is the user interface for users to type commands into. Its constructor directly takes the user’s input and a HistoryBox. The constructor sets the size of the box and all that stuff. It has a private method handle(KeyEvent event, HistoryBox h) that is used to evaluate buttons pressed by the user. It has an update(Observable button, Object commandText) that is it’s only public method. It clears the textbox if the user presses enter. This class is really well enclosed such that it doesn’t have any weird dependencies. Rica wrote it, and I think it’s a good example of encapsulating the function of some code into a single well-defined class.

**Design**

The overall design of our project is split into 4 main groups: the FrontEnd Package, the BackEnd Package, the Nodes packages, the Pen Package, and the TurtleView Package. The Pen and TurtleView packages are ultimately FrontEnd stuff, and the Nodes packages are all BackEnd stuff. The code is delineated as follows: basically everything concerning displaying information on-screen and taking input from the user is handled by those frontend packages I just named. When the user enters text input into the input section of the GUI, the frontend passes the string input back to the backend, where the input is parsed and then transformed into a bunch of node commands that are placed in a queue based on the order in which they were evaluated. This queue is then used to construct a syntax tree that will be traversed in the correct order to as to affect the actions specified by the user’s input.

In order to add a new type of command that users of our program can use, we would create a new class with the name of that command that extends our node class, or if the function of the command can be more specifically identified then it could also implement one of Node.java’s subclasses (LoopNode, MathNode, etc.)

An example of my code and its design is the LoopNode class. At the time I felt as though it was well designed, but as I looked back over it I realized its main method is the iterator method. And that isn’t a very good name for explaining what it does. We needed a way to loop through a certain portion of our list of nodes without destroying the list (Q) and that would stop when it reached a certain type of node. I should have called it something like extractExpressions or something similar that actually indicated what its function was. I regret the naming of that method. Our overall code is designed relatively well with that constant notable exception of the tree. Honestly we went with the iterative tree because I wasn’t convincing enough when arguing for the recursive tree. So we stuck with the shaky iterative design that was already implemented, and that limited us in what we were able to implement going forward. This is the huge roadblock of our code.

Describe three features in detail:

1. Model.java: This is the class that the front end’s Controller.java class uses to communicate with backend. Model takes the input of the user, creates a Parse.java object to parse it, and constructs the syntax tree in a makeTreeandAddValues method that is then called in Model.parse that uses this information to return a SceneUpdater. Model is not a subclass of anything, but it needs several field variables: a string input, a parser, a Tree, and a list of printValues. A model object is instantiated in the Controller.java class as a way for front end to communicate with backend and have the backend evaluate user input. I think this class is extensible because it’s acting as a manager for a host of other classes. The design of this class would lend itself well to manipulating data using classes other than parser.java, AST.java, etc. because it’s basically just managing the order in which those classes manipulate some data input.
2. Parser.java: This have a fair amount of field variables. It has a list of strings that represents the names of various packages that it uses with reflection to construct the Nodes without the use of switch cases or if statements. It has a string array called splitWords that is the array resulting from the splitting of the initial string input that it gets from Model.java. It has a turtle object. It’s newInfo method that functions basically as a constructor, setting up the values for the fields. This is poor design, in my opinion. That entire method ought to just be the constructor. It has a convert(String[] ) method that just removes the unnecessary parts of the input from splitWords (marks or punctuation that our tree will ignore). The most important method in this class is the getQueueOfNodes( ) method that iterates through splitWords and uses reflection to generate the correct Node type and then puts those nodes into the Queue of nodes that are ultimately used to construct the tree. This class is basically designed to take some input and change it into another data structure. We would need to change the getQueueOfNodes( ) method to get a List<Objects> for it to be more extensible. I think that would be a very doable change, though. Depending on what you wanted to do, this could be the class you use to manipulate and change your data types into something usable. Its flexibility is limited though, due to its specific nature: it’s really just designed to parse some data. It is probably overly specific in its implementation.
3. UserCommands.java: this class stores a list of commands that the user has created that the user can then click to rerun them. It only has one field variable: a ComboBox object. The constructor instantiates this object and the class itself has one public method that adds new commands to the ComboBox. If the command is already there, it obviously doesn’t add it. This class is well designed because it is a good example of encapsulating one valuable process into a single class. It adheres to the “One axis of change” concept that we learned about in one of the early readings of this class. To make it more extensible we could make the field variable just an unspecific object and use reflection in our addCommand method to check if the method’s input could be added to that object’s information or ‘stuff.’ This would allow the class to accommodate other types of objects besides ComboBoxes. This is the class that I was not part of implementing.

**Alternate Designs**

The original design didn’t handle the projects extensions as well as we would have liked. I sound like a broken record but the reason the design failed to support some of the extensions is the syntax tree.

Three Design decisions discussed in detail:

1. The communication between front end and back end. We wanted the communication to be as little contact as possible. We wanted the front end to have some class that could simply use one back end object and invoke a single one of its methods that would give the front end back all the information that it needed to animate the turtle’s pen. We did this by encapsulating all of the functions of the backend and it’s various classes within the model class, and the front end made a controller class that created a model object and called its parse( ) method. That’s how the front and back ends communicated and I think that’s a very simple and good way for them to work together. There aren’t many dependencies to consider in this design and I think it worked very well. It also lent itself very well to implementing multiple turtles. Because all the front end needed to do was iterate over the turtles and call model.parse() on them. One con of this design is that the turtle class in the back end did end up needing to be shared by both the frontend and the backend meaning there were slightly more dependencies and things to be aware of than we initially intended. We considered not having a model class and just having the front end invoke the different backend classes in the correct order to achieve the same result, but we ultimately decided that was irritatingly complicated and so we went with the simpler approach using the model.java class.
2. The node packages. For the Node hierarchy, we initially weren’t sure if we needed to create more abstract classes that extended Node.java before we implemented more specific node classes. For example we weren’t sure we would need to implement CommandNode.java as an abstract class. There was initial discussion of simply having all of CommandNode.java’s subclasses extend node, and implement individually the logic for each node. The upside of this was that we could clearly see what each needed to do at the time, so it felt like it would be the most straightforward approach. However, we realized that by having all of the command node types extend the abstract command node class, we could give them all the same functionality in the parts where it counted (like how they know when they’re done evaluating, the way in which they clear their info, etc.)
3. The Abstract Syntax Tree. This is the most frustrating part of my while project. I’ve already said a lot about it. We did not implement this recursively, which was a poor decision. The trade-offs were extremely one sided. I talked to my roommate, who is also in the class, and he implemented his tree recursively. He said it was simple, and it only took 40 lines of code without a million if/else cases (ours had a lot of those). Basically every new type of node required a tweak to the logic of our tree implementation, and this made adding new features take an outrageously long time. This also means that our tree is not very extensible at all. It actually inhibits extension.

Three most important bugs:

1. The tree
   1. I would simply redo and it make the tree recursive
2. The LoopNode class
   1. I would make LoopNode abstract, put its functionality into RepeatNode, and to also change the tree to recursive
3. Lack of variables and functions
   1. I think again we would just make new nodes that hold the right logic for this, and implement the tree recursively so this could be added easily.

**Code Masterpiece**

I decided to implement the IfNode.java class for my Code Masterpiece. That was a feature lacking in our initial project design, and given that it was a basic implementation I felt like our project needed to have this. It involved adding a few methods to the Nodes superclass. Specifically a method for extracting expressions that come before commands, and a method for extracting commands that follow methods. This is well designed because it is an extensibility feature. This is because there are many commands where it would be useful to pull out chunks of the input (expressions and groups of commands) to evaluate as their own tree (like recursion even). Given the poor implementation of our tree, it also meant another special case of (if (current instanceof IfNode)), but given the design of our tree and all the irritating dependencies it would take to change it (beyond the size of the code masterpiece), that’s a tradeoff of this masterpiece that I have to live with for now.

As for JUnit testing this I would test that the code for extracting expressions and extracting commands from a Q of nodes works properly, since that’s the part of this that the implementation of IfNode.java hinges on.