Eirika Sawh

SLogo Design Review

October 30, 2014

Design Review

Status

I feel as though the code is generally consistent in that the names of classes follow a pattern and are readable. For example, all of the Features have Feature at the end of their name, all commands start with Command, etc. With regards to readability, I feel as though everything except the parser is very easy to read and understand with clearly labeled variables, methods, and/or switch cases. The parser might be hard to read to me because I’ve never seen something in JavaCC before – it might be really well written and I just don’t know it. Generally speaking, if a piece of code needs something, it is passed either that specific variable or an object which has access to it via a get method. If it only needs one value, typically just that value is passed through, but if it needs multiple values, then the whole object might be passed through. I’m not how bad that is, or if we should always pass the whole object through. There are also some inconsistencies with this. For example, **public** **void** showStage(TurtleHandler myTurtleHandler) has a TurtleHandler passed into it because it uses multiple methods in TurtleHandler (.getTurtleLocation() and .getOrientation()), whereas **public** **void** updateTurtleImage(Point2D newLocation, ImageView turtleImage) has the actual Point2D and ImageView already passed in without needing to get a TurtleHandler and then call the appropriate get methods. If we had more time and had realized this, we probably would have tried to standardize whether or not we should pass in the objects that hold those variables or the variables themselves, or at least better decide when to do which one. The currently implemented features feel easy enough to extend – for example, adding a new command and adding a new Feature seem to be straightforward. However, this is assuming there is only one Turtle; although we started working on generalizing our code to account for multiple Turtles, since we didn’t quite finish it, adding any new features that required having multiple Turtles (such as an Ask or Tell command) would be difficult given the implementation we have now.

Testing different parts of the program would be easier than others. For example, testing if a new command worked would be fairly simple by just generating the input and seeing if it does as is expected in a test, whereas testing how different GUI elements are laid out or interact would be more difficult to do in a JUnit test without actually seeing the GUI itself.

I did not find any huge bugs in anyone’s code, although I know there are probably some bugs in my own code (such as how DrawLine in ImageUpdater doesn’t quite work in every case). I probably found small bugs when I was working with Yoon, but since those were quickly resolved, I can’t remember any that were worth noting.

Class 1: CommandsFactory

Since I did not have any part whatsoever in writing this class, I found it amazing how simple it was to read and how I could understand everything that’s happening in it. Additionally, I thought the use of the Factory Method Pattern was apt in that it allowed other Command subclasses to be created by the factory when they were needed. This makes adding new commands very simple too (will talk about this more in the Design section). The code was also broken up into different sections (shown by Javadoc comments) which also improved readability. The only thing I might say would be to maybe add Javadoc saying what a CommandsQueue, CommandsRepeat, CommandMake, and CommandExecute are. To make this usable in a completely different context, the Factory Pattern can be maintained, and all that would need to be changed are what the individual methods are (such as pickUpTurtle, putTurtle, etc) to reflect what new methods/commands need to be present in the different project.

Class 2: Main

Although I worked with Yoon on the front end, I didn’t actually work too much with the Main method. I picked this code because although I liked how the code sets up a workspace from the beginning and adds keyboard controls, I feel as though there are some clunky parts to it that could have been refactored out into new methods. What I personally would do is move out the for loops from the start method into their own methods, such as a createWorkspaces method and a setTabSelection method, or something similar. I feel that this would improve readability just to know what exactly happens in the start method by reading what other methods it would call and knowing what those methods would do, so instead of (for example) having to follow the for loop to find out that there are workspaces being created, you could just read “createWorkspaces” and know what’s going to happen. If different methods were created to set up different things (like the createWorkspaces method), then if this were to be used in a completely different program, you would just have to rewrite those individual methods or pick and choose which to use and call them in start and everything would work fine. If there weren’t extracted methods, it would look messier as you would have to go into the start method and change all the for loops within the method and add many things to start which would clutter it up.

Class 3: LogoParser

Unlike the other two classes, I chose this class because I personally find it very difficult to understand. That said, it seems to work well with many cases based on every command possible. Again, this may just be because I am not familiar with JavaCC, but I feel as though, since there are already some comments such as /\* block statements \*/, adding more comments such as those would improve the readability of the code. However, I do feel that adding to the parser wouldn’t be too hard, since you could just add new case statements for the new commands, similar to what was done in CommandsFactory. Therefore, although it would probably require rewriting most of it, using the parser in a different program that used a factory would just require (alongside changing the factory) changing the case statements. However, since it appears as though most elements of the parser end up sending things to the CommandsFactory, if the program didn’t use a factory, the parser as it is now wouldn’t be very useful and would probably have to be mostly rewritten.

Design

At the beginning of the program, Main makes a GUI and some workspaces, with various Features that could be added if desired. Once the user types into the text box, that text gets sent to the backend. The basic design for the backend is a Parser that breaks up an inputted string into meaningful words that are then interpreted using case and switch statements (Interpreter Pattern). Then, with the Factory Pattern, the CommandFactory makes a new ICommand based on what it was interpreted as. The various commands act upon a TurtleHandler (Command Pattern) which serves as the controller between the back and front end. The TurtleHandler will update the Turtle (or, if we had it fully implemented, multiple Turtles) based on the command given and then also tell the ImageUpdater whether or not to update by moving the Turtle or drawing a line. When the ImageUpdater receives that instruction, it updates the canvas as needed. Therefore, I feel as though our current implementation embodies the MVC pattern with the view updating as the controller receives inputs from the model in the form of commands.

Adding a new command would be as simple as adding a new case statement to the LogoParser and a corresponding Command, as well as a method in the CommandsFactory that does whatever that command is supposed to do. (If this ends up needing new functionality from the Turtle, this would then require the TurtleHandler needing to be updated as well since it is the controller that the Commands act on.) Adding a new component to the front end would require making a new subclass of Feature and then added to the Workspace so that when a new workspace is made, that feature will also be created.

As stated earlier, my code section was designed to be a controller between the backend and the frontend. Commands in the backend operate on the TurtleHandler that manipulates Turtles and then the ImageUpdater.

I feel that the way my code is designed is good because it limits who can access what pieces of information. For example, the commands don’t have complete access to the Turtle(s), instead just telling my data structures generally what to do and assuming they’ll do their job correctly. I believe that this sort of Mediator is useful because it reduces the communication complexity between the back and frontend. As for parts of my code that I felt iffy about, I’m not sure if it’s a good or bad thing that, when the TurtleHandler talks to the ImageUpdater, I pass in particular values (such as a Point2D from, PenHandler mainPenHandler, etc.) instead of the Turtle itself. I think it’s a good thing since it restricts access to the Turtle, but am unsure if it may clutter up the code by adding many parameters.

Feature 1: multiple workspaces (completed)

To implement this feature, we created a workspace class that houses all the various Features that might be included in a workspace. Everything in the workspace is public, which makes it open. It assumes that when you create a workspace, all the features in the createWorkspace method will be put in. Since this may change from workspace to workspace, it isn’t very flexible. Thus, to extend the code, it would be better if the user could send the workspace a list of Features they wanted included in it and then the workspace could create a workspace based on that.

Feature 2: moving turtles with various inputs (completed)

To implement this feature, the only thing needed is a TurtleHandler, or, if the TurtleGroup had been fully implemented, a TurtleGroup with a list of active TurtleHandlers. Since the TurtleHandler has methods that take in amounts to move or turn, it’s already equipped to handle a variety of inputs. Thus, if the user wanted to add, for example, buttons that when clicked would move the active Turtle(s), they would just need to set up an event handler in the buttons that would tell the TurtleHandler to update in the appropriate direction. Therefore, any button or input can be made as long as it has an appropriate method in TurtleHandler. Thus, I feel that it’s very easy to extend this feature to a variety of inputs and other commands such as buttons that put up or down the pen, etc.

Feature 3: multiple turtles (not completed)

Although we worked on it, we did not get the multiple turtles fully working. However, we had started on making a GeneralTurtleHandler class and I think we were at least somewhat close to finishing up multiple turtles. Adding the capabilities for multiple turtles would in general make the program more flexible because there wouldn’t be any loss in capabilities – if the user decided they wanted to just have one turtle again, they wouldn’t have to rewrite the program. What would be required to write it from the point we have it at is to have a GeneralTurtleHandler be created in the main class and have some TurtleGroups made for each workspace, or maybe just one TurtleGroup that can keep track of which Turtles are on which workspace. If this were implemented, adding multiple Turtles would be easy. However, if the user wanted to extend Octopi and Fish or something, there would have to be a GeneralAnimalHandler or a related class with similar methods to the TurtleHandler so that there wouldn’t be any problems in the Commands.

Alternate Designs

Although we were not finished with the basic design when we received the instructions for the project extensions, I think that the at-the-time design held up fairly well to the extensions. We had already begun working on a Features class that allowed us to add color pickers easily by just making a ColorFeature class, and other Feature classes as needed. I didn’t work on the Parser, so I’m not entirely sure as to how much changes had to be made to that, but it seemed to be well written and just adding case statements to it to reflect the new commands was needed. Although we had accounted for the advent of multiple turtles, we didn’t finish getting that done, so I suppose our design didn’t handle that extension too well.

According to the API Changes document, only a few things were changed to the Feature and the Help Page, and the bulk of the changes were to my sections of the code, the Turtle, TurtleHandler, ImageHandler, and Pen/PenHandler. Most of the changes were made as new capabilities were added – for example, get/set pen size had to be added to the Pen because we had not accounted for that from the beginning. Additionally, some methods such as getPenColor/Position were added to the TurtleHandler because we moved Pens from being a standalone object to being something that a Turtle has. Additionally, some public methods like getTurtleCanvasSize and clearLines were added to ImageUpdater because the TurtleHandler needed them – it was hard to predict what methods would be needed in the classes the TurtleHandler interacted with before starting to write the code, so many changes were made.

Design Decision 1: making a TurtleHandler

The alternative was just having a Turtle without a handler, but we felt that having a TurtleHandler added a level of abstraction and division between the back and front end. The pros of having done this were that it kept the front and back end distinct, it allowed the TurtleHandler to communicate with other parts of the program such as the ImageUpdater where having just the Turtle would be inappropriate, and had there been multiple Turtles, it allowed for a TurtleGroup to be used instead of just one Turtle. However, we could have gotten around that by just making a list of Turtles (not advised though), and it may have seemed easier to code or with less gets/sets if we had just used a Turtle without a handler. Overall though, I think it was a good idea to add a controller between the model and view.

Design Decision 2: adding a Features hierarchy

Instead of making just a ColorPicker or just a RefGrid, we decided to put these into Features so that they would all have that common hierarchy. I think this was a good design because it keeps a sense of uniformity between all of the Features. Additionally, although we didn’t implement this, if we had had the workspace have a list of Features, then we could just add to that list every time we wanted to add a new Feature to the workspace. At the moment, making new Features is a bit hard-coded into the workspace, but we could pass in new Features when we make the workspace so the user could specify what Features they wanted or not. The pros of this is that it makes adding new Features very easy, and the user will have a very general sense of what that class is supposed to be (they wouldn’t have any confusion as to whether or not it’s related to the Turtle, the Parser, etc.). It greatly clears up the code as well since it would have its own constructor, or if they were in a list, they could all have make() methods or something similar to make it easy to create new Features. I can’t see a downside to having our features in a hierarchy such as this, but there might be better solutions that we didn’t think of. Therefore, I highly prefer having them extend Feature as opposed to being on their own.

Design Decision 3: Making a CommandsFactory

Alternatively, we could have had numerous if statements based on what the Parser parsed, but we felt that the Factory pattern was appropriate for the task. We didn’t spend too long thinking about not having one, because it seemed like the obvious right thing to do. The pros of having the Factory is that it’s easy to add new Commands, and it lessens the conditional logic needed. Because of how easy it is to use, I greatly preferred the Factory to just having Commands and conditional logic.

Bug 1: Looping lines drawn doesn’t work correctly

For some reason, going off of the edge of the screen and looping around doesn’t always work in the Y direction perfectly. I’m not entirely sure why this happens, but we could test that by writing tests that figure out where it’s supposed to loop and draw lines, and then see if that actually happens in the program.

Bug 2: Previous Commands might have commands that don’t work

To fix this, we could maybe have something in the CommandsFactory tell whether or not the command will work or not. It doesn’t add to Previous Commands if it’s syntactically incorrect, but a nonsense string such as “asdf 40” will still show up in the Previous Commands. Or, have something in the Parser check to see if the string exists in a list of valid commands.

Bug 3: Can’t create new workspaces

At the moment, we have a set loop create 3 workspaces, but if we extracted that out into its own method and made a button that called that createWorkspace method, then we could probably fix that shortcoming.