SLogo Analysis

Wenjun Mao

NetID: wm56

Team 05

Time Review

For this project, I estimate having invested around 50 hours of my time, beginning 9/29/2014 and ending 10/25/2014. I splitting my time around 15% reading directions, code documentation, tutorials, and forum posts to understand how to code what I wanted to code; 15% thinking, planning, meeting and discussing the project design and implementation details with teammates; 55% coding new features; and 25% refactoring and debugging.

Since it is already the third project we finish using GIT, the merging became easier and didn't cause any problem during this project.

The easiest things I had to do was designing the parsing algorithm. Since I have taken a course taught by Prof. Rodger previously about parsing algorithms, the designing basically took only a few hours to figure out a parse table and then I could start to implement the program.

The hardest thing is that since our group have a really sloppy API design, the front and backend communication did cause some problem during our programming process.

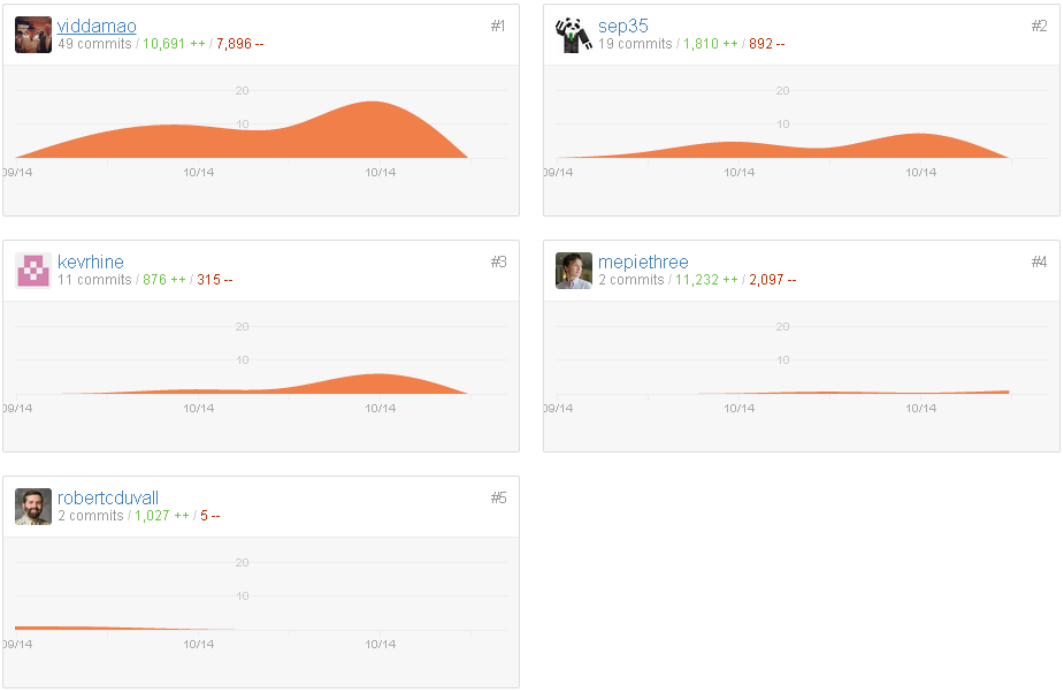
Teamwork

Our team was cohesive overall and open to exchange of ideas. However, there still exists communication issues and, no offence, but I really doubt if everyone have put effort in the project. The overall meeting time is only a few hours in total so lacking of communications caused a lot of inconvenience during the implementation.

Since I have some kind of prior knowledge to the project, the planning was almost done by myself alone, which caused some communication issue later since I might have not made myself clear enough in the design plan and we did not have enough time for team meeting. Although I am aware that the backend might take more responsibilities, our team was kind of procrastinating until the day before the deadline to finish all the basic implementation so the overall UI and graphic looked sloppy.

In our original plan, divided responsibilities for front and backend, which we maintained through the course of the project. Basically Zach and I are responsible for backend, while Steven and Kevin are in charge of the front end. But I think it is also necessary for everyone to have at least kind of an idea for how other parts are working to make the whole project well-functioning, which I think my last group for Cell Society did a lot better than this time.

I really think is we had put more effort in earlier our project would be a lot better. I know that the commits and lines of coding does not directly correlates to the amount of work but it at least shows some idea of that. The graph I took from GITHub is shown below:



Our plan for completing the project involved implementing a front end GUI which displays all the models and return data and gathering user inputs.

The core idea of our plan remained kind of stable except we kind of thrown away the MVC model and ignored the Controller, despite we maintained a package of controller.

The compiler package takes in the input data from the users and outputs a list of commands. The overall extendibility of the compiler is not very good since each time for an addition of command we need to add clauses to two separate functions and also requires modification of the data files containing parse tables.

Through the course of our project though, the implementation details evolved as we better understood how our classes could work together (i.e. we started to use different separate classes for each of the individual class, but we replaced later with a command list and set of

anonymous subclasses). In terms of simulations, the GUI class simply gets the turtle data and displays on screen

Our team quickly divided up the extensions and began work on them after the details were released. However, since we kind of procrastinated during the basic implementation period, we do not have enough time for the extending. Some features we can extend but harder to do due to our original design were given up. But

I developed my code in bits and pieces, developing the core functionality first, testing functionality, and then pushing it. For example, in creating compiler functionality for the program, I created the methods to recognize tokens; then I take the output and send to another method that generates derivation rules,

next I created a class that generates a syntax tree based on the previous derivation rules returned to check the syntactically correctness of the user input; then the tree is post-order traversed to generate a list of executable commands and lastly, I refactored the functionality into a class hierarchy.

Sometimes I would push before my code was functional during my previous project, which caused errors for my teammates, and I learned to stop doing that during this time.

Overall, I erred on the side of more communication to balance any potential loss of communication and make sure everyone understood what was going on in all aspects of the project, especially when the turtle object I managed are used by the front end people and they may not know how that supposed to function.

Commits

In total, GitHub reports that I committed 66 times to the project in the span of the implementation period. I would merge and push least once per day if I update my code to minimize the potential conflicts, and to let everyone know what I was doing/had done that day. Some of my commits were really big (new/deleted classes, refactored hierarchies), while others were tiny (such as documentation and null checks). In the future, it would probably be better to commit somewhere between those two extremes to keep updates and changes more consistent each day, rather than in big or little steps.

I would usually push code as soon as I committed it, to make sure everything was the same and remains functioning for all branches (frequency is thus at least 1 time per day, as with commits above).

There was no obvious merge conflicts this time except some minor ones.

My commit messages accurately represent my contribution to the project.

Three commit examples:

Commit Description: modified API design, added token recognizer and turtle model

commit ca64ac8ee85515649c352586fd3c89d3ece86efd

Purpose: To begin creating functionality of the project. Initialized the classes I was working on.

Consequences: None.

Timeliness: This was the first few commits of the project besides the plan.

Commit Description: Compiler now able to check if the given commands is syntactically correct

commit 73a2c964e8233cf27892a7c2e75e204dcbfcc604

Purpose: finished the SLR parsing algorithm so that it can check the syntax.

Consequences: Some spacing conflicts with my teammates upon pushing. However, nothing major. Most of our conflicts were spacing, naming, or method movement conflicts.

Timeliness: This commit was timely when I half way during the implement of the project

Commit Description:

Revert "have the drawLine method in pen, you can just get list of lines of pen

commit de5807cbb92a907b2fb4e1315fb214e00868aa41

Purpose: tried to initialize the data structure for Lines which allows clear to function, but it caused the whole GUI to be not functioning since their using of Lines are written already so I reverted it

Consequences: None, fixed a problem I introduced earlier when attempting to create a new way of storing lines.

Timeliness: Near the deadline, when sighted a problem I just dealt with that immediately.

Conclusions

Overall, I believe our team worked pretty well together. While there were certain small issues in our time together, and the functionality of full implementation are not all implemented, we have a functional slogo program in the end at least.

I definitely underestimated the size of this project when we first began planning; we had some foresight into what the simulation might require in the future, however we mostly stuck to fulfilling the basic function of the first deadline. In the future, I believe it to be better to overestimate than underestimate, as overestimating leads to more extendable code (regardless of whether we extend it or not), as well as better design overall.

I do believe I took on enough responsibility within the team, although I could have taken on more responsibility. I continued to think about how well our design was progressing as we continued to implement new features, and thus very cautiously added new components and features. Throughout the project, I made sure my team was informed about what I was doing, from commits and pushes, to Facebook messaging and discussions.

The Compiler package required the most editing, because our original parsing algorithm design was not very extendable, and adding new instructions meant we had to update the Syntax Tree, Command, data files and potentially the User Interface classes due to the inherent dependencies among them. Lastly, the model hierarchy required a fair amount of editing, to adding more abstractness to the whole hierarchy.

To be a better designer, I should continue to communicate with my teammates to come to a consensus on a best design. To be a better teammate, I should be clearer in communicating my ideas, continue keeping lines of communication open.

If I could work on one part right now to improve my grade, I would work on refactoring all my code twice over as some of my classes got duplicated codes for the convenience of coding. Also, I would implement more features, which our design is capable of doing.

Design Review

Status

Our code overall is generally consistent in layout, naming conventions, and style. For our code and method layouts, we follow normal coding conventions: instance variables at the top, defining constructors, and then grouping methods and their helper methods together in blocks.

For example, our Turtle class contains its defined instance variables at the top, then a constructor, followed by its update state, setter and getter methods, as well as other function calls. In terms of naming conventions, we tried our best to maintain the same instance variable names across all classes.

However, there might be some hiccups that we missed.

Each class contains similar styled Javadoc documentation. Most of the classes, since we each worked individually, would exhibit code that is perhaps more personal in style. For these classes, we did not discuss the design as a group, and thus slight inconsistencies in coding style may be seen.

For future projects, we will try to be more transparent about our code for classes others are not in charge of, and attempt to code to a similar level of design quality.

For most of the classes, the method's name is self-evident in declaring its purpose;

All methods in the CommandList are corresponds to each of the instructions entered by the user, thus do not need further explanation. There are some instances where better naming conventions, and creation of additional methods could lead to better method clarity.

While most of the dependencies are clear and easy to find, such as the UI pass info to controller then from controller to compiler.

Our design overall allowed us to easily add new features to our existing working simulation. Moving turtle with arrow is easily done since we have a command list to call, other GUI features were easily added to the GUI class;

Further, our design the addition for more instructions, although the extendibility is not good enough but it is within our ability of handling.

The compiler would be easily tested for correctness, as it operate based on local parameters and take on string values which we could print and/or compare in a tester without any involvement of display.

The MainController and the GUI may be difficult to test, as extensive test cases must be created as inputs (various XML files for the Parser and various initializations for the MainController).

Checking through our project, I found no bugs in others' code.

In order to implement the command list, I had to read up the GUI classes for how they supposed to perform changes I send, as well as apply changes. Thus, I learned and understood how they worked. And besides this I think the rest classes are all written by me.

In order to make our code clearer and maintainable, as well as more elegant, extensive refactoring needs to be applied. First, we need to make sure we follow the principles of SOLID, especially the principle of Single Responsibility and Interface Segregation. Most of our classes contain too much functionality, and need to be refactored into other classes to clarify our

design, and sometimes for convenience our GUI just directly get the turtle model which kind of violate our original MVC model. For example, our compiler class should have a clearer division based on the functionality.

Furthermore, button classes as well as factories could be created to generate individual buttons for placement into the UserInterface class; the same could be said for the command factories in the CommandList. Lastly, completely redesign our compiling and Parsing could greatly improve the design of our Simulation overall, and give our program more extendibility.

Design

Adding a new command to the program is simple (although could be simpler with parsing mechanism refactoring). First, modifying the parse table for the program to recognize the command. Next, adding in the syntax tree based on the derivation rule and add the command to command list if the command is differ from the existing ones. (We can FD -1 to achieve BK 1 so no addition needed in this case) Lastly, if the command is dealing with displaying, the GUI needs to be modified but in most case it doesn't. Nothing needs to be changed with the rest of the classes. Having added/changed these things, launching the application and type in new commands should work. We made sure to discuss as many design considerations as possible, in order to make sure everyone was on the same page, and capable of working independently at the start of our project.

To choose a feature and talk about it in detail, I choose my compiler Hierarchy.

Upon receiving the project, we can see that the recognition and execution of the commands are the main part of this project while the rest are just GUI and display stuff on screen.

The compiler package is used to compile and interpret user input (in the form of strings) into runnable and usable code. The compiler mainly contains three parts, a scanner, an interpreter and a syntax tree constructor which I have separated to the AST class.

The compiler takes in the user input string, which could be one or more commands, then it is splitted based on white spaces and passed to token finder to recognize all the tokens,

The modified input is then sent to the interpreter where we can check the syntax for the input.

While we read the lookahead and scan the whole string, we are able to get a list of derivation rules that tells us what command we are engaging.

Then the derivation rules are sent to the AST class to build a syntax tree, which we can post-order traverse to get a command list.

My command list class contains functions that can be added to a list when traversal of the tree encountered certain commands.

The model package mainly has a turtle model that contains all the needed parameters for turtle display, the turtle list that enables multiple turtles, a pen class to hold pen data, a line class to draw lines and a point class to hold two double values for the turtle location.

Ideal Design

Our ideal design is similar to our current one, except I would like to change our current compiling package. Our compiler mechanism is kind of hard to extend so I think check the input token by token is easier way to recognize each command than my current structure.

While our current design is more similar to the compiler for a program that takes in all input as a whole and utilize a SLR parsing mechanism.

Code Masterpiece

```
package compiler;

import java.util.Locale;
import java.util.ResourceBundle;

public class TokenFinder {
    /**
     *
     * Defines the different type of user inputs retrieved from the command line
     *
     */
    public enum Type {
        CONSTANT, VARIABLE, COMMAND, LIST_START, LIST_END, PARENS_START, PARENS_END,
NOT_RECOGNIZED
    }

    private static ResourceBundle regExProperties;

    /**
     * defines the type of the string
     *
     * @param s
     * @return the type of the string
     */
    public static Type stringType(String s) {
        regExProperties = ResourceBundle.getBundle("properties\\commandRegEx",
                Locale.US);
```

```
        if (s.matches(regExProperties.getString("CONSTANT")))
            return Type.CONSTANT;
        if (s.matches(regExProperties.getString("VARIABLE")))
            return Type.VARIABLE;
        if (s.matches(regExProperties.getString("COMMAND")))
            return Type.COMMAND;
        if (s.matches(regExProperties.getString("LIST_END")))
            return Type.LIST_END;
        if (s.matches(regExProperties.getString("LIST_START")))
            return Type.LIST_START;
        if (s.matches(regExProperties.getString("PARENS_START")))
            return Type.PARENS_START;
        if (s.matches(regExProperties.getString("PARENS_END")))
            return Type.PARENS_END;

        return Type.NOT_RECOGNIZED;
    }
```

And since the Command List class is too long I will not paste it here. I will just create a pull request on GITHUB with these two classes.

My Token Finder class and the CommandList class is what I think to be the core of my whole design in this project. Since we are given all the formats for the input commands and variables, etc., we are able to rewrite all the input words in the format of regular expressions for java to recognize. Then the processed string is passed to the interpreter to check if it is syntactically correct.

The Token Finder design is good because instead of checking the string character by character, we can just split the string with white spaces and check each part and saved a lot of effort.

For sure we can see a lot of if clauses and return that can be refactored into a for loop which an index array, I didn't do that because I think by listing these clauses can show how this method functions more easily.

The Command List idea was provided by Zach, I was originally going to make a separate class for each of the commands. But in this case, by using anonymous function we can merge similar methods into one and easily create new commands.

Potential JUnit Tests:

Testing both classes are easy,

For the Token finder, just pass in the different kinds of input types and check the return types,

And for the Command List, we can just create a new command execute, and then check if it successfully modify the turtle object.

The unit test is included in the pull request.