PROJECT JOURNAL

Time Review

I spent on average, about 20 -25 hours a week working on SLogo. Our team met for about 3- 4 hours a night during the week. We started planning the project on September 29, and finished the plan on Friday October 3 . We finished the basic implementation of the game on the Morning of October 13. We then worked on the project extensions, till the morning of Sunday September 29.

Plan - 10 hours

Basic Implementation - 45 Hours

Refactoring ~ 4 Hours

Testing ~ 3 Hours

Coding ~ 38 Hours

Complete Implementation - 40 Hours

Refactoring ~ 4 Hours

Testing ~ 2 Hours

Coding ~ 34 Hours (Adding New Features)

Documentation ~ 2 Hours

Initially, I managed my code using my own branch, in which I made my working set and developed my parts of the project. However, I sometimes encountered merge conflicts when merging my branch into master.

The planning phase of our project proved to be very crucial. The time we spent working on how we would structure our program, and what design patterns would best serve our goals was time very well spent, because the design decisions we made early on in the project were key to determining how easy it would be to implement SLogo effectively.

Working on the model part of the program was a great experience, and it wasn't too difficult to figure out what I needed to do. However, the same cannot be said for trying to save and load the workspace, this part of the project was rather challenging.

On average, we met for about 2-3 hours every night, to plan, and design the project. The goal of these meetings ranged from simply going over design decisions, to actually coding certain features of the project.

Roles: I worked on the project's model, includes consists of the main model, the turtle model, userDefinedcommandsModel, userDefinedVethodsModel and a number of other classes. I also worked on saving and loading the workspace to file. In addition, I also helped Will with implementing some of the interpreter expressions.

Will worked on the command interpreteer, which included the command parser and command reference library. He also worked on the commands and he command language.

Jonathan worked on the animation, multiple turtles, the workspace, grid/turtle and menu bar.

Abhisheck worked on the command prompt, the method/variable views, history view and language view.

Outside of our group, we consulted our TA, John Godbey with regards to design decisions, and program functionality.

Overall, communication within the team was good. We met often enough and communicated via groupme and so it was easy for us to get in touch and consult each other even when we were not in meetings. This made it easier for us to work on the project in our own time; if you ever had a question about another part of the project during such times, you could just post a grou message, and other members of the group would respond. During meetings communication was always clear and we were able to work well as a team.

Our plan of completion was rather successful. We our roles such that 2 members worked on the backend, and the other two worked on the front end, the plan was that we would work in these subgroups for most of the project, and then help each put everything together. This proved to be rather stable, and efficient. Because we worked on different parts we were able to prevent deadlocking and it was easy to update the different elements of the code.

The plan and roles were okay when considered for the project extensions. However, because other parts of the project such as the interpreter had substantially more work to do compared to others, it mean that we had to help out Will with some parts of the interpreter. However, the was not easy as, it involved having to trace the work he'd already done, and fully understand the functionality of what he had already implemented. Because I had worked on a different part of the prject prior to that, it made it a but challenging to help him out with parts he'd worked on.

Commits: For some reason, GitHub is not displaying my commits, and i was unable to resolve the issue. However, I committed on average, 5 times a week, but this number greatly increased towards the end.

Conclusions

Overall, our team worked well together. After overcoming a number of initial setbacks due to unclear planning and task divisions, we worked very well together, as we knew what each of our roles were, and what we needed to do to produce a good project.

Initially, we underestimated the amount of work we had to do for the project. It was only later into the project, that we realised that we still had quite a long way to go and a lot to do.

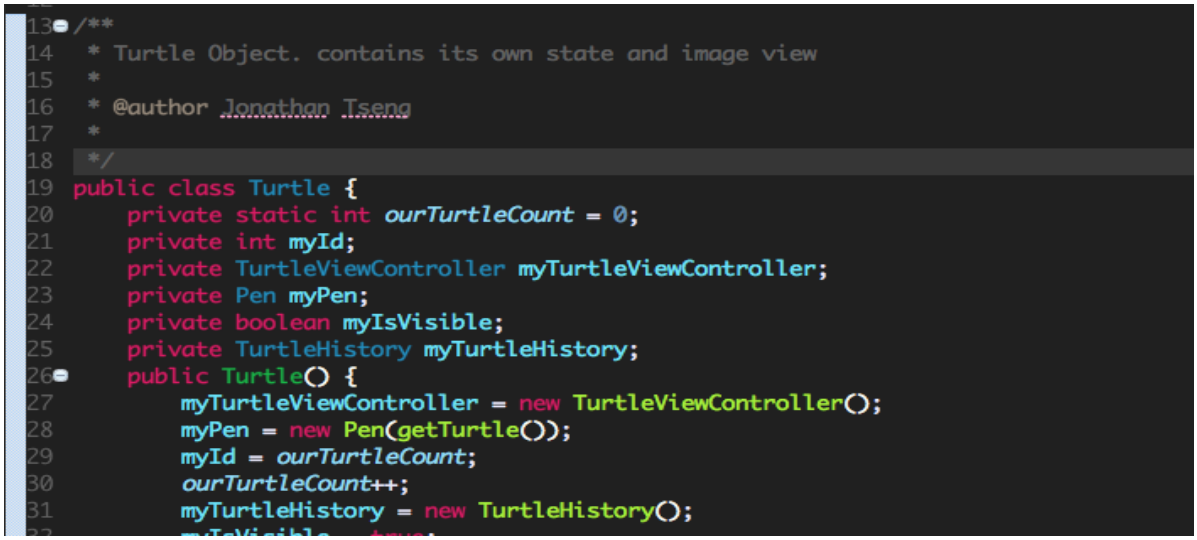
DESIGN REVIEW

Generally, the code is consistent in it's layout, naming conventions, descriptiveness and style. I think we did a good job of choosing a consistent trend in the style of our code for different parts of the program. The code is generally easy to read and understand, our different classes are separated into different packages, making it easier to find files, and understand how the program works. Dependencies in the code are pretty clear to find. Classes which make up front end features are grouped together, and those which make up back end features are also grouped together. Inheritance relationships are within the same packages such as increasing clarity and they are used to create sensible relationships between different parts of the program. Extending the SLogo programme we have at the moment should be relatively easy, this was seen in the extension we made from the basic to the extended part of the assignment. Most of the elements in the back-end should be relatively easier to test, however, the backend is likely to be more complicated in this regard. In the backend, the Parser and the Interpreter should be testable pretty easily because these parts take in SLogoCommands and are supposed to return int values as results of the command parsing and evaluation. Tests can be easily implemented by using input to which the correct results are already know, and these can be compared against the results that the interpreter produces at the end of evaluating the commands. On the other hand, because the

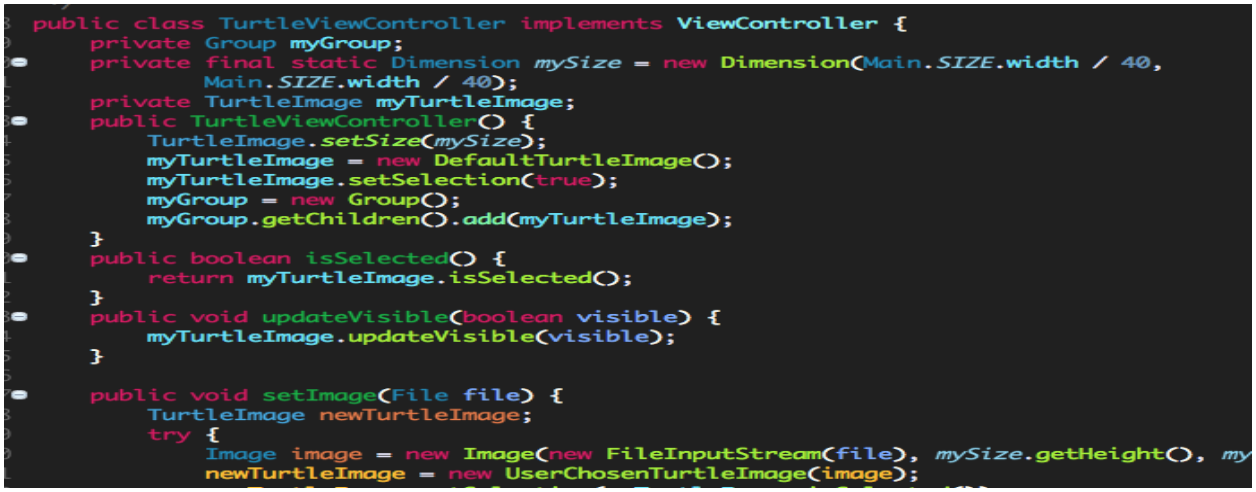
front end of the project is made up of mostly JavaFx components, it would be rather challenging to effectively generate tests to check the correctness of implemented methods in the View related classes.

Class Review:

Turtle



TurtleViewController



TurtleImage

```
public abstract class TurtleImage extends Group {
    protected final static Color ourSelectedColor = Color.GOLD;

    protected static Dimension ourSize;
    protected boolean myIsSelected;

    public TurtleImage() {
        this.setOnMouseClicked(event->toggleSelection());
    }

    public boolean isSelected() {
        return myIsSelected;
    }

    public static void setSize(Dimension size) {
        ourSize = size;
    }

    public void toggleSelection() {
        myIsSelected = !myIsSelected;
        selectedChanged();
    }

    public void setSelection(boolean isSelected) {
```

I have chosen the three classes Tutle, TurtleViewController, and TurtleImage. I chos ethese three classes, because they show great use of objects and their behaviour in OOP. The Inorder to represent the turtle that is on th screen, the author makes about 8 different classes. The turtle class is mostly used to store the state of a turtle.

Each turtle has it's own instance of the TurtleViewController class. Insetad of trying to control the turtle graphic or view details within the class, the author chooses to make a separate object, whose job is to take care of the turtle display and it's graphical components.

The TurtleViewController itself holds all the visual aspects of a given turtle, and controls the visual aspects of the turtle. However, even within the TurtleViewController, the author chooses to further establish behaviour seperation and abstraction by creating a TurtleImage class to take care of the Image of the turtle. This way, the TurtleViewController doesn't become cluttered but remains very clear and easy to understand, as it is not crowded by operations on the individual images. This resulted in a better overall design of the turtle features, and clarity in how the Turtle Objects and its fields behave.

I think these 3 are well designed, however, I would still advise the author to consider getting rid of the static setter in Turtle image. In addition, maybe getting rid of the getters and setters in Turtle, if possible.

In order to make the code extensible, for a completely different project, we would need to add sub models for each of the CA models being simulated. A given simulation would have it's own submodel which would have an instance of it contained in the programs main model. That way, the main model would still act as an Observable, with new simulations easily added and adapted. In the front end, instead of having a specific Turtle class, we would need to have a more abstract "actor" class which would then allow for not only the display of turtles, but also different actors in the different simulations. Every other class directly related to the Turtle would simply be abstracted to support an actor instead of only a turtle.

In choosing the design for our project, we considered the different Object Oriented Programming Design Patterns. After reading through the OOD reading from class on oodesign.com, we decided to create an intuitive and extendable Logo API through implementing the Model-View-Controller (MVC) paradigm in unison with the Observer pattern.

The program is divided into 3 main modules, the ViewController module, the Model module, and the Interpreter module. The ViewController handles updating the view and giving functionality to the buttons and input fields of the UI. It is be an observer of the Model. The Model hold all of the state of the program. This includes things like the list of transitions that have occurred to the turtle from the SLogo function calls and the position and properties of the turtle (position, rotation, visibility, pen

down/up, etc.). Finally the Interpreter works to understand and evaluate the user-inputted SLogo commands and pass this info onto the model. Overall, the flow of code is thus; the user sees the View supplied by the ViewController and can input SLogo commands. The input string is then be passed to the Model which passes it to the Interpreter for evaluation. This updates the Model, and because the ViewController is an observer of the Model, the View is in turn be updated as well.

To add a new command to the language: when a user defines a new method, initially the command is passed through the view into the parser. The parser then parses the input, and creates an SLogoExpression is created from the method. This SlogoExpression is then stored in the UserDefinedCommandsModel. From here, user defined commands can be easily accessed in future evealuations to determine the SlogoExpression of input from the user.

To add a new turtle to the front end: the user initially selects the option to add a new Turtle, under the "Grid" dropdown menu. When the user clicks this, the event results in the creation of a new Turtle in the main model, and this turtle is stored in the mainModel's list of turtles. When this this happens, the ViewController, and other observers are notified of a change in state of the Observable and hence the the existence of the new turtle is registered, and the view controller displays the turtle on the screen.

We separated our program into three modules - the Model, the ViewController, and the Interpreter–which together execute a read-eval-print loop. We combined the traditionally separate View and Controller into our ViewController user interface module to represent the read aspect of the loop. The Interpreter module then serves as the eval step, applying any logic necessary, represented by

our SLogoExpressions hierarchy, to transform user input into usable states. The Model completes the loop, saving the state and updating the ViewController, performing the print step and allowing future user inputs. Furthermore, the Model servs as the Observable module, saving states and updating our list of Observers in the ViewController. In designing the program, we closed to modification the order of method call within this loop and the message types passed between the modules, however we left open to modification the types of SLogoExpressions users might want to add to SLogo. Further, as an Observable, the Model should be extendable to additional graphical features (such as CellSociety). The Interpreter acts as a Black Box to parse input (from our ViewController) into user defined state changes. Further, we assumed that users may want to customize our ViewController, as well as our read-eval-print loop for their own applications and so we tried to make these parts as clear and extensible as possible.

I think the MVC design pattern coupled with the Observ/Observable design pattern was a good choice in designing our program. Because of the nature of the program ie it takes in input from a view and the input is processed, using MVC proved togood because we were able to separate the different parts of the execution and close them off. In addition, the Observer/observable pattern meant that all we needed to do was store the state of the program, add new states, and respond to changes in the state. This proved rather useful and ane made the program easier to manage.

Alternate Designs:

In our chosen design, we decided to combine the traditionally separate View and Controller into our 'ViewController' user interface module to represent the read aspect of the read-eval-print loop. The

alternate design would have been the usual MVA approach of having the view and controller separate and treating them as separate entities, instead of having them encapsulated as one component. However, we instead decided to have 3 distinct modules - View Controller, Model and Interpreter, we chose this in order to keep the reading, evaluating and printing parts separate. The alternate design would have had reading the input as 2 parts, and would have made our separation of modules less uniform so and blurred the division lines between modules The alternate design would have consisted of having both inputting commands, and updating turtle as different components such that both type of functionality would be under different parts of the program. This would have included the command line, input buttons and options, the view of the turtle, and the view of previous commands. However, as the functionality would got lengthier and more complex, this may have resulted in a scattered design. So in order to prevent this and keep our design clear, simple and distinct, we joined the view and controller and utilised the shared properties and similar functionality between them.

Code Masterpiece

I chose my MainModel Class to be part of my masterpiece. I chose this because it demonstrates a sound understanding of inheritance the MVC design pattern, and the Observer/Observable design pattern . I tried to close off as much of the main model as I could. I also demonstrate good examples of code independency as I just “Tell” the other code what to do by simply notifying my observers of my changes, and letting them do the rest.

The most important methods to test are those which relate with my observers, have to do with adding observers, and notifying them of changes in the state of the program. This is rather crucial as the

program hinges on the Observer/Observable design pattern in which the model is the Observable, and so changes in state must implement correctly, and also must be transmitted to the respective observers.