SLogo_Team02
Analysis

Mengen Huang

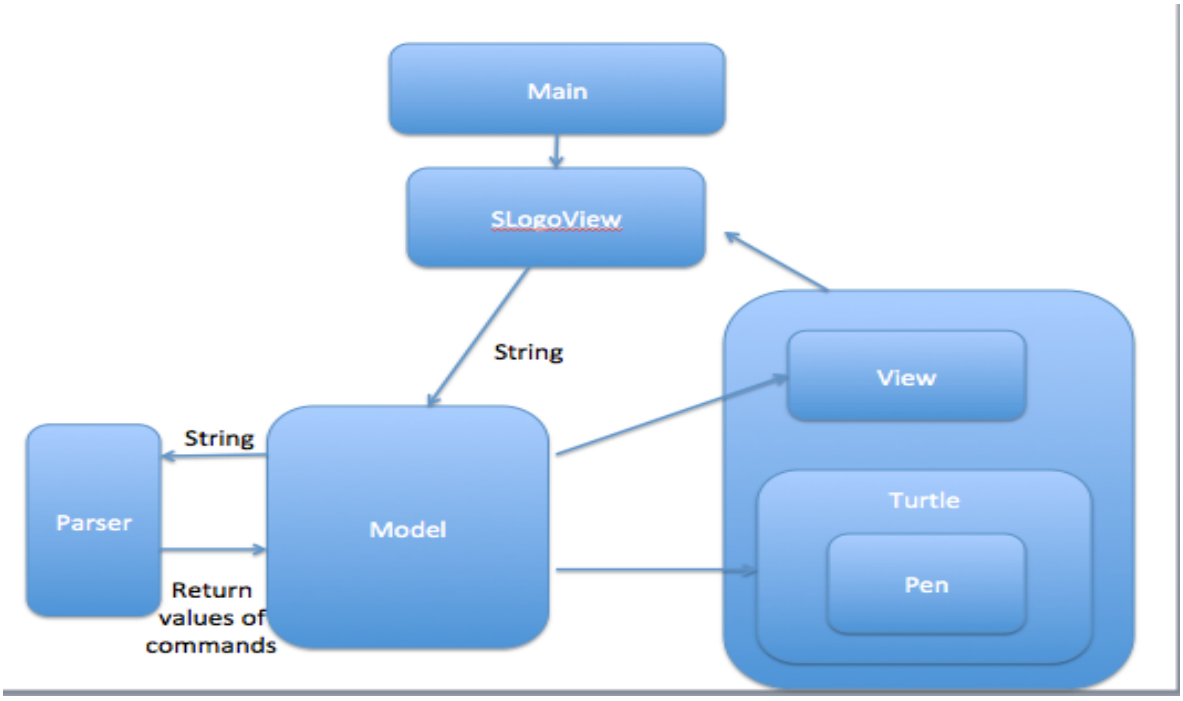
Project Journal

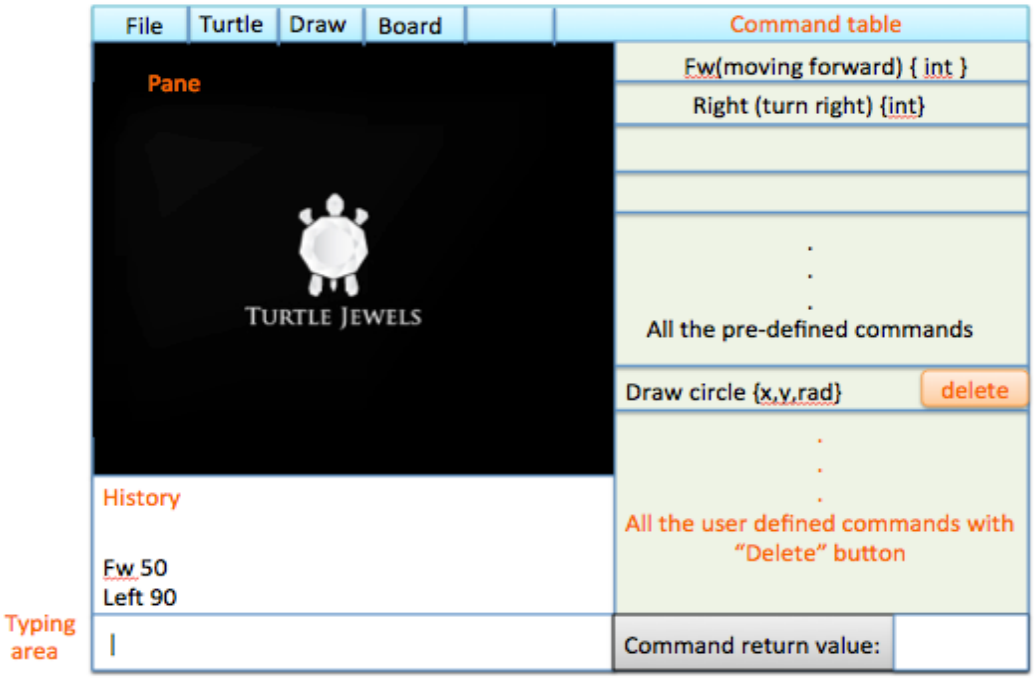
Time Review

- **Plan: 9 hours**

Our first meeting was on Oct 30th for 2 hours. We summarized our lessons of CellSociety project and briefly divided ourselves into front-end and back-end. We also came up with the basic ideas about constructing our slogo project. I drew the interface design that we implemented later and we together designed main classes which later developed into packages. We all met on Oct 2nd again for more than 7 hours to design the structure in details and put all the designs together into a pdf file. We took a long time to brainstorm all the possible problems with our design and made modification. We recorded our final design in a google doc and I made the UML Diagram for our design structure, designed and drew the interface, and wrote all the alternative designs we had come up.

UML Diagram of our original design and Interface design:





- **Basic Implementation: 47 hours**

Oct. 4th night we met and worked together for 9 hours to build up an updated version of API design and built up the basic structure in eclipse. During the meeting, we read all the different design patterns and decided to use Observable/Observer Pattern to connect back-end model package and front-end view package. Before we met, I spent 3 hours to read related codes found online and wrote up a project prototype code which divided project into parts and wrote the public method of each class. During the meeting, we elaborated each classes and made modification based on the prototype code I pushed to api_design branch.

It took me 5 hours to read through javafx documents and write out the basic view package which creates runnable GUI. It took long to write the first version of GUI as it took long to read through javafx documents, decided what to use for each components of GUI and what function of each Java component to apply for our project.

Before the basic implementation due on Oct 10th, I spent around 30 hours in total to implement content in History View and Command View, made an input area with a Submit button which passes the input string to back-end and move the input to History view once clicked. I also read through all the back-end codes to have a better understanding of the whole project while making some edits in model package so that the back-end simulation can work better with the front-end display of GUI.

It took me most of time working on the command view so that the content can be clickable. At first I used javafx component “table view” for command view so that all the commands are well organized and can be arranged according to the user’s preference with default function. It took me several hours to figure out how to use a observable list to create a new data structure so that the table view can automatically arrange all the data in the observable list. However, after it finally worked, I did massive research on table view trying to enable the content run MouseEvent and found out I could not assign mouse event handler to the content of the table view except clicking to edit the content. As I hoped clicking on the commands shows the command in InputView, I decided to use ScrollPane instead of TableView to show the commands. In terms of result, the exploration of TableView is a big waste of time as it took much time to figure out how to implement and it failed to implement features I planned eventually.

The most of time I spent during this period is adding in new features while testing and debugging them. Kevin and I worked together on GUI branch so that we can test the GUI by clicking the temporary test button on GUI to make sure the turtle is able to move and draw lines and all the other GUI components are working collaboratively.

- **Extension Implementation: 63 hours**

Coming back from fall break, I first spent 2 hours refactoring the view package and changing all the inappropriate name convention.

Then view package and model package Kevin and I were working on merged with Parser and Instruction package Jennie and Sandy were working on. So I started working on getting the command return value from the back-end and show it at the HistoryView. After finishing implementing new feature in History view, I got merge conflict about all the classes implementing instruction package as one of our group member change the name “Instruction” from capital letter to lower cases “instruction” to follow the naming convention. Then I change around 50 classes involved instruction package manually to make the project work. However, after I push my change to the github, I found the name of the class does not change and after I pull again the name goes back which causes me to

make 50+ changes again through the whole project. Later with the help with group members, I removed the instruction package, pushed, added back the renamed instruction package, and pushed, successfully fixing the problem. It was surprising that the change made to class name does not update with github and the time changing all 50+ names was not a good use of time. All these work took around 6 hours.

Then I spent around 6 hours everyday on average for the rest of implementation from Oct 19th to Oct 26th. The work includes reading others' code, adding in new features, cleaning model package, modifying model package to adjust direction of turtle's move, added delete button for user-defined commands/variables, fix bugs for the back-end Instruction by changing function of the method in model package, and implement new language settings of the SLogo. Spending time looking for Chinese Regular Expression and translating all the commands and GUI text into Chinese is a great fun. After implementing Chinese, I found it was also quite easy to add in French as another SLogo language by translating GUI text and adding code roughly 3 places in the project.

At the end of the project, I spent another 7 hours to write javadoc and refactor code for view and model package. It took me a couple of hours dealing with the checkstyle error and finally with the help of TA Kevin Jian found out it was the checkstyle version for lambda function that causes the error.

Teamwork

Our team spent a lot of time planning at the beginning_ 9 hours as I previously stated. We spent 2 hours splitting works and tasks during the first meeting. During the second meeting, we spent 7 hours reading design pattern documents, designing the project pattern, brainstorming any exception case that may cause error in current designed structure, and documenting all the alternative designs and final decision. We decided to spend a lot of time planning and designing together because we think a well-designed base for the whole project is quite important. In case any exception cases occur later when we worked in detail or during extension part, we took great effort coming up cases our current design cannot handle well or cases affecting the efficiency, readability, extensibility of our project. Also, we decide to discuss the parser and instruction package together because we considered back-

end parser the most important and difficult part so that each of us can make contribution. Also, we hope each of us be aware of how other parts work so that we can all learn and contribute to both front-end and back-end design.

For each person's role in the team, we generally divided us that I worked at the front end, mostly view package. Kevin worked at the part connecting front-end and back-end, mostly model package. Jennie and Sandy worked on purely back-end, mostly parser package and instruction package. Kevin and I collaborated a lot working about the GUI and we will change each other's package when needed. Jennie and Sandy worked mostly on back-end code together and they will change classes connecting both back-end and front-end like class ObservableData, CommandList and VariablesList. I think each of us has a good idea about others' codes and we will make changes to the classes related to the tasks we were working on and notify each other.

I consider we have great communication among our team. We meet or message each other progress every day so we always keep each other updated. Whenever we encountered a change that may affect other classes other person was working on, we usually meet together to discuss about the structure change. Later as front-end has lots of implementation to do, we made adjustment so that we do not have to get four of us discuss together on one problem but we still keep each other updated. The pros are that we always learn and solve the problems together either the problem is at front-end or back-end. The cons(or maybe not cons) are that we spent much time helping others so that in terms of result both back-end and front-end are on the same page because one who finished the current task would help others instead of proceeding to the next step.

We mostly follow our original design though we later move Turtle and Pen class from view package to model package. We also later changed the SlogoView into a higher arching class which contains multiple SlogoWindow class which plays the role SlogoView originally did when we are asked to make multiple tabs.

Our plan for team role holds the same throughout our project as we made all our roles quite flexible at the beginning. The plan did not quite change for the front-end. We kept the original division for the interface and the multiple tabs only requires generating multiple model-view combinations.

Commit

From the github documents, I did 46 commits with 4277 addition and 2446 deletion. I think they accurately represent my contribution to the project. I constantly committed and two commit peaks occurred when I worked on basic implementation and extension. The deletion peak occurred in the middle of two implementation periods as I was cleaning up view and model packages before starting implementing extension part.

Three sample of my commits:

1. Commit message:" basic view structure"

   It was the fist time the GUI was created with several classes of methods in model that will potentially simulate the action of the GUI elements. I uses BorderPane for the GUI elements arrangements and I divided the GUI into four parts at the first commit: SPane where the turtle and the lines were shown, DisplayTextArea extending TextArea which shows the history and commands return value, CommandsTable extending TableView which shows all the commands and variables, InputTextArea extending TextArea which enables the user to type in commands, and the main class which will run the SLogoView class where all the GUI components are instantiated.

   As we first put all the front-end codes in "gui" branch and it was the first commit of runnable GUI, there is no merge conflicts. It was done in a timely manner relative to the rest of team as Kevin was experimenting how the view and model communicates and back-end started designing and coding parser.

2. Commit message:" clean up view package"

   After meeting with TA Kevin Jian discussing about the view package, I decided to clean up the view package as each of GUI components need to set up the width and the height, bringing up much duplicated codes. As each class has to extend different javafx components and each class can only extends one class, I dealt the problem according to Kevin's advice by making an interface View class so that each GUI components will implement it and override to set up the

size. I also used lambda function to replace the one-line EventHandler function code in SettingsView class to make the code more concise.

The commit was originally made in gui branch and later merge with master branch so it did not cause merge conflict for others. I did the clean up before merging with back-end codes so I consider this commit done in a timely manner relative to the other group members' work.

3. Commit message:" change direction of turtle moving according to shown positive coordination direction"

   We notice that when we asked the turtle to move forward, it actually moved downward though its head is upward, and the y coordinates in turtle information shown on GUI is countering the Cartesian coordinate system. The actual coordinates representing the view on vertical axis is from top smallest to the bottom largest, so I made the calculation and created a new variable called myRelativeCoordinates which labeled the center of the TurtleView (0,0) and followed Cartesian convention.

   The commit did not cause merge conflict for others according to github commit records and it was a timely appropriate debugging change.

Conclusion

I consider we estimate the size of the project in a good sense though implementing some javafx components requires more time reading the documents and figuring out how to implement them into our project than I expected. I think I take enough responsibility within the team as I took charge of the view package, modified model package, implemented two other languages, and read all the back-end codes. As others in our team, I always update others about my progress. Implementing Chinese to our project required the most editing as I needed to translate all the commands and texts on GUI to Chinese characters and it took me a long time to find the right regular expression for Chinese characters. And I worked on CommandsList, and InstructionFactory back-end classes to figure out the best way to implement the new language and worked with Jennie to make the VariablesList work better to implement new language features.

To be a better designer, I think I should do more readings about design patterns and experimenting more javafx components. I should keep reading others' code to have a good idea about the whole project. To be a better teammate, I think I should keep the high quality and quantity working and I should have a higher frequency communication with teammates as there was one time Kevin and I worked on the same feature. If I could work on one part right now to improve my grade, I would work on the challenging extension part as I think it was not hard to implement if we had more time.

Design Review

Status

The code is consistent generally in its layout, especially after Sandy ran the checkstyle for parser and instruction packages and I ran the view and model package. The layout and naming conventions and descriptiveness are generally consistent as we always discussed together and had agreement on the names and function of each class. The original style is mixed as we all make modifications through the packages but the final version is also quite consistent in style as each of us took charge of refactoring one part of project. The code has a good readability though as we use a special way to deal with the parsing, the comments in Parser class helps a lot with understanding the algorithm of processing the input string. It is not too hard to find the dependencies in the code as we put all the components of GUI simulation in "ObservableData" class and made them all implement "Feature" to make the code cleaner so that we only need to get ObservableData to make all the changes instead of implementing multiple model classes. However, the sacrifice is using several type castings. I noticed that though Turtle is an important feature, we did not put it into the feature map but use getter and setter because we intended to implement multiple turtles so we have to separate it from the map. It is good that we do not have global variables and not many get method. We also did not use Controller package but instead we use Observable Observer Pattern so that the Observer's update method connects the model package and front-end view package. It was not too hard for front-end to make extension though the back-end found it frustrated to implement multiple turtles in a neat way as they made the

InstructionFactory and Parser really clean so that the back-end instruction has to do more, which makes implementing multiple turtles bringing up many duplicated codes.

I think it is easy to test the Tokenizer and Instruction class as they return the result and do not have much to pass in. The test for view package is a little harder as we cannot write a Junit test for testing the result but we have to run the application every time I tested it to see if the features are successfully implemented. I think it will be hard to test the parser and InstructionFactory as the result are mostly Instruction, which is an interface of classes. The correctness of the parser has to be tested together with specific instruction.

I found a bug and fixed it in Turtle class. The detail is included in the third sample commit I talked about. Basically I made the calculation from the actual coordinates to a user-friendly Cartesian coordinates which I called RelativeCoordinates. The actual coordinates iare from the top left smallest to the bottom right largest. The user-friendly coordinates set the center of the view (0,0) and go positive going up and right, and negative down and left. I also change the SetCoordinates method from using actual coordinates to using RelativeCoordinates because it was easy for us to using actual coordinates to add in lines into Group, but the back-end instruction will also call the SetCoordinates method according to the cooridnates the user defines. The instructions SetHeadings and Home went right after I fix the bug. I also find that though we can use “Sum 3 5”, our SLogo does not recognize “+ 3 5”. It was because in the InstructionFactory when we are judging if the input string is a command, we only use English RegEx or Chinese RegEx without including expression “+-*/~”. I think we can either delete these symbol expression from the properties file so that they will no longer be shown at the command list view on GUI, or add in the RegEx of these symbols so that the parser will process them as a command. Another bug at the back-end is that when I create and defined a user-defined command, I cannot call it in the same input string right after. It means I have to submit after I define it and then I can call it in the next submit to actually run it.

To review three classes in the program I did not write or refactor:

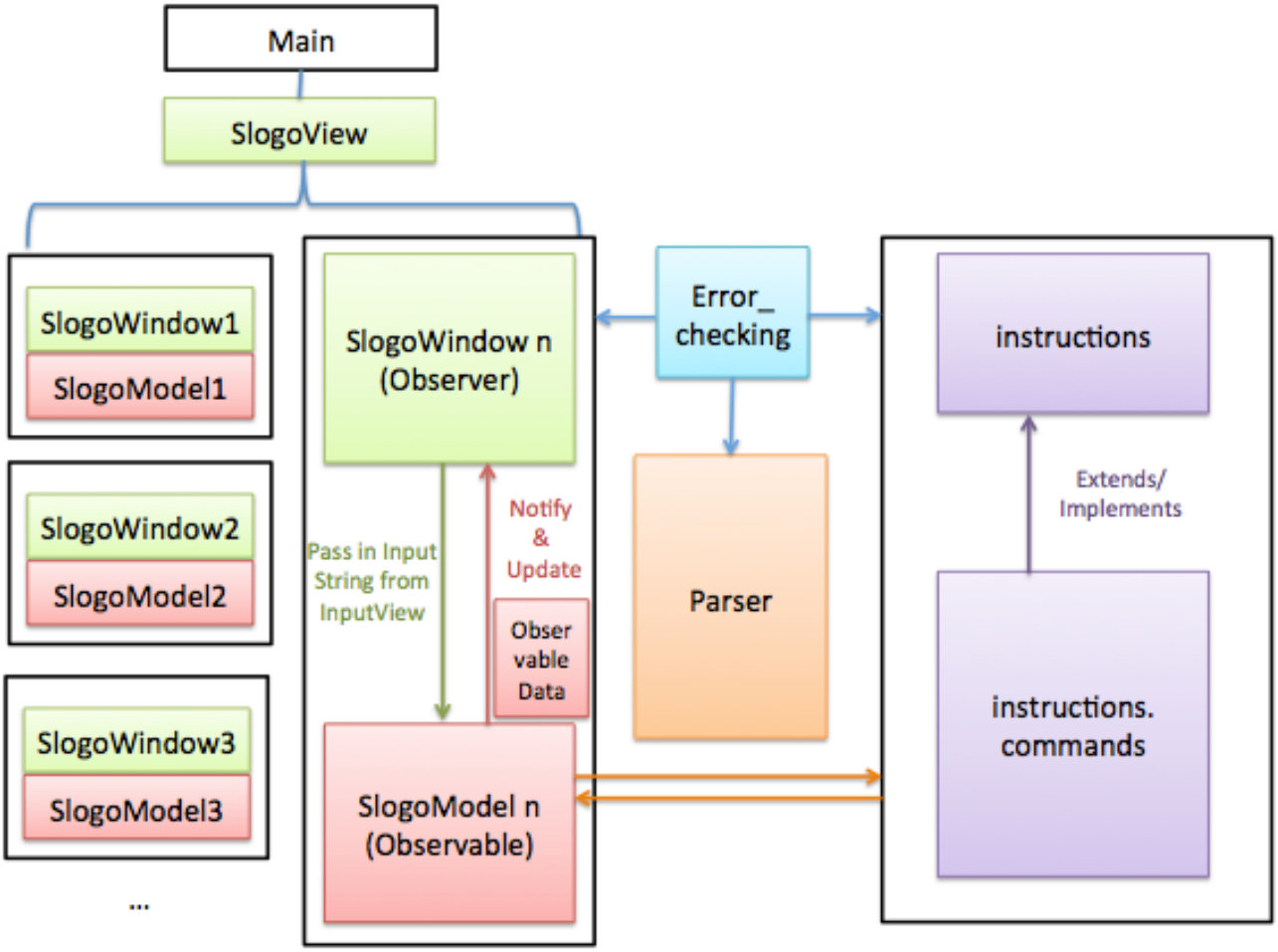
I would like to first talk about InstructionFactory class because I think it is extremely clean and is implemented in a wise way. It uses Regular Expression to categorize the parsed string and transfers the String to a general type of instruction class. To improve this class, we can store the RegEx into a properties file and read them in using ResourceBundle to make the code even cleaner. I also like the implementation of constantInstruction and variableInstruction so that constant and variable are executed in the same way as command and they can implement the same structure as command. The other project like CellSociety may not use the Regular Expression in this class but it is the same structure that we make classes that functions the same extend or implement the same super class and it makes easier for us to categorize and implement.

The ObservableData class also plays an interesting role in our project. It contains all the GUI component models also used by back-end so that we can just pass ObservableData class to the instructions for execution instead of implementing different model classes for commands making changes to different GUI components. In the Observable and Observer pattern we implemented, ObservableData class is also the class passed from the model to view so that SlogoWindow class do not have to implement all the model classes and update them all. Instead, we just get the ObservableData from SlogoModel and update each feature in ObservableData. The tradeoff for this useful class is many type castings. We also used a new java8 feature in this class to implement a variable number of Feature parameters. "`private void addAllToMyFeatures(Feature... features)`" which enables us to take in a variable number of Feature in the method without specifying a specific number of parameters taken in. This feature is extremely helpful when implementing a number of Objects parameters and I think we can definitely use this feature when we need to do same thing with different number of Object parameters.

I would also like to talk about the CommandsList class. I implemented part of the class which related to front-end GUI and changing language, but I think the way the other group members implemented the map is wise and the Java Reflection is quite useful. To group commands in the map instead of randomly distributing them, we made another properties file to group all the commands according to

their functionality by making the function genre the key and all the command syntax the value. The Java Reflection is also quite useful as we easily passed in a String representing the name of the class and got the whole class returned if the class exists. I think the Reflection pattern will greatly reduce the "if" statement and could be widely applied in other project when we want to get a class from its name.

Design



The overall design of the complete program is shown above in the diagram. There are five main packages which I marked in five different colors and a Main class to start the whole project. Five packages are view, model, parser, instructions, and error_checking. The Main class will call the

SlogoView to start the GUI which enables the user to create new tab, open existed file, save current work and choose the language. Once a new tab is created, a new set of SlogoWindow and SlogoModel is created. The active SlogoWindow will pass the user input to the corresponding SlogoModel, The model then calls the parser to turn the string input into instructions and the instruction classes execute on myObservableData in SlogoModel. We use the Observer Pattern to update GUI when changes occur in model. The SlogoModel class where all the model components are initiated is the Observable and the SlogoWindow where all the view components are initiated is the Observer. Whenever there is a change in SlogoModel, it notifies the observer SlogoWindow and passes ObservableData containing all the possible changed components. When the SlogoWindow get notified that the Observable class is changed, the update method is automatically called. The error_checking package checks the exceptions and errors in all the other four packages.

To add a new command to the language, one only need to go adding the command in the instruction.commands package and extend the right abstract class according to its function and parameters the command takes in. To enable the command runnable in different language, one only need to go to three properties file stored in resource.language, and add in command Syntax as key and the command in a specific language as value. Three classes needed to add in the new command are Commands_en_US.properties, Commands_cn_CN.properties, and Commands_fr_FR.properties which records all the commands in English, Chinese and French.

To add a new feature at front-end, the simplest way is to create a new class, set up the feature and define the update method, and find a class to implement it. For example, though we put the toggleGridLine in TurtleView, we can actually make another class called toggleGridLine, and we set up the grid lines in this newly created class. Then we can construct toggleGrideLine in TurtleView class and add it as a child node of TurtleView. If we are going to add new features to the SlogoView instead of features in SlogoWindow, we can do the same addition as in SlogoWindow, but make our new feature as a node of SlogoView.

For this project, I constructed the view package, made addition and modification in model class, add in KeyControl which enables the user to press “W”, ”S”, ”A”, “D”, and “X” to move forward, backward, turn left, turn right and set the turtle head back to the default straight upward. I also implemented different language so that our project can run in English, Chinese and French.

I use BorderPane for the window to organize all the GUI components. Borderpane can contains at most five parts so I put settings view where all the button locates at the top. The center is the TurtleView where the turtle and lines are shown. The left is HistoryView where the InputHistory and the command return value are displayed. On the right shows all the predefined commands, the user-defined commands and user-defined variables. At the bottom is the Input TextArea and a Submit button by clicking which the content string in Input TextArea is passed to the backend. After experimenting different javafx components, Kevin and I decided to use ScollPane for both side views. Each Text content of the view is stored in a HBox. The History content and predefined commands are Text in the HBox while I also add a delete button in the HBox for all the user-defined commands and variables. All the content in the side views are clickable and the user can click to get it in the input area instead of typing the commands and variables. For user-defined commands and variables, the user can delete it by clicking the delete button at GUI to also delete it from the back-end map. Also because the commands are stored in the map so that if the commandsView simply shows the KeySet of the map, it shows all the commands in a ramdomly organized order, so I also made a list that read in the properties file in order. In general, the inputView only passes information to the back-end, the turtleView, the HistoryView, and the settingsView receive the updates from the back-end and communicate with other components of GUI, and the commandsList change the back-end information while communicating with other components of GUI. Kevin and I refactor the view package in the end so that view on the same hierarchy extends the same class and the components with multiple features are divided into multiple subclasses.

I also add in the turtle information view on the turtleView which can be switched on and off. Later Kevin change it to ToolTip so that we can save space of our GUI and the user can still see the turtle information while move the mouse on the turtle.

I then add in the language of the SLogo by making multiple language properties file. First I made the Chinese commands runnable. After comprehending how the parser package works, I find it was not hard to implement Chinese commands as both Chinese and English commands share the same command syntax in properties file. After finally finding the right Regular Expression for Chinese characters using Chinese searching on Google, the problem is solved. Then I made another properties file containing all the GUI text with different language version called Display_locale and add locale into all the classes needed to vary based on locale. I passed in the locale to SlogoWindow which then passed the locale to the particular front-end classes. Only CommandList class needs locale in the model class so I passed the locale from SlotoView to SlogoModel, and to ObservableData where CommandList is instantiated. Then to change the language of MenuView in SlogoView, I made another set of properties file called Menu_locale, which records all the Text on MenuView. Once the language option is changed, the MenuView will change immediately but the SlogoWindow will not change. The user can create a new tab with the newly selected language and the new SlogoWindow will run in the new language. In this case, our SLogo application is able to run with tabs in different languages.

After the language settings are changed, the KeyControl needs to be changed too as I move the turtle with key by parsing commands I set. For example, press “W” will ask the parser to parse and execute “forward 10”. When the language changes, the previously set command no longer can be parsed in another language. I have considered moving the key action totally to the front-end that directly call the Turtle to move. However, as we save the file by saving all the history commands and open the file by re-executing all the commands according to Professor Duvall’s suggestions, make the key event directly call the turtle will leave no trace of command called by the key. So I also make KeyControl class take in the locale and read in the properties file according to the locale so that any key event will also be recorded by back-end.

Alternate Designs

Our original designed structure for the project did not change till the end as we spent long time discussing about it. But we add in lots of features, classes and some design patterns in the end. Below are some general alternate designs during the plan.

1. Make a pane with every grid as a pixel. Pros: Same implementation as in CellSociety. Cons:Each grid has to be very small and the implementation is a waste of memory.
2. When drawing the line, get the starting point and the ending point and let the turtle jump from the original location to the target location. Pros: Faster to process. Cons: Unable to see the process of drawing and no way to change the speed of drawing.
3. The front-end gives list of string the user types into the parser and executes the commands at the back-end. Then the back-end returns a list of commands to the front-end and the front end will execute all the commands to update turtle or add in user-defined commands. Pros: The turtle class will be totally at the front-end and will not involve in command execution at the back-end. Cons: The line between front-end and back-end gets vague as part of command execution which related to the turtle is at the front-end. It also makes the parser hard to decide when to return the command because some of the commands do not involve turtle. Also, as we decide command like “forward 50” returns a value 50 and “forward forward 50” equals “forward 100”, it is hard for the back-end to process the commands with several layers and decide if the commands have been simplified to those only related to the action of turtle.
4. We used to decide to put the error checking of front-end and back-end into the same class and show the error through the GUI. Pros: Group classes of same function together to enhance the readability of the codes. Cons: The front-end and back-end have to share the access of GUI, increasing the visibility and influence between two ends. Solution: The back-end will throw the error from the pop-up dialog window to avoid accessing and changing elements in GUI.

Three design decisions:

1. Using the Observer Patter by adding in the class ObservableData which contains all the model components potentially needed to be updated by the commands. We considered using Controller package to include all the ugly codes in it and build the connection between model and view package while making the other classes look cleaner. But Professor Duvall says the Controller package is unnecessary as javafx has the EventHandler and Observer Pattern is recommended. I think using Observer Pattern to replace the whole Controller class is a good try and the code is not too ugly after refactoring.

2. We considered discarding ObservableData class and make all the components of SlogoWindow Observers. In this way we can get rid of numerous type-casting in our project. However, we have to load all the components for each instruction as not all the commands only involve turtle. This would make the back-end instruction quite messy. Personally I prefer using ObservableData because it makes the code much cleaner and I personally think type-casting does not make the code any harder to read or extend.

3. We have discussed about how to save the environment and considered serializing everything in the current working environment. However, it is hard to serialize turtle and lines in the view as all the other components are features while the turtle and the lines are components of a bigger feature. After talking with Professor Duvall, he suggested us to save the command history as a textfile instead of serialize everything. We took professor Duvall's advice as there is not much time left before due and we still need time to figure out how to implement serialization, though we think we will eventually figure out using serialization because Duvall says we will implement it in the next final project.

Three most important bugs:

1.Key events for turtle works after the user executes commands or use the button in settingsView but it does not respond right after we enter a new SlogoWindow. I believe the bug can be fixed by somehow setting the focus on a certain view component so that the key event can be triggered. I have

tried set the SlogoWindow, the class instantiated KeyControl, get input Focus once set up but it does not work.

2. The turtle will go out of the boundary of the view though it correctly return the right coordinates. I think we can fix it by set a boundary of the view.

3. When the environment work in Chinese, we can execute the Chinese command by clicking on the Chinese commands displayed on the right side of GUI and type in parameters, but the user cannot type in Chinese commands in InputView. I thought it could be solved by set the properties of TextArea, but I did not find any built-in function that set the locale of typed-in text. After massive research, it seems the Chinese characters are recorded and represented in specific unicode. As in properties file it is already in unicode and displayed as Characters when it is read in by GUI. However, the TextArea cannot directly take in Chinese characters without changing anything further than built-in function.

Code Masterpiece & JUnit

Code Masterpice

I chose to refactor the KeyCode class as I consider it plays an important role between both front-end and back-end and implemented the different languages while refactoring is needed.

The class is a front-end feature, that user will use the key to control the movement of the turtle. However, to save the movement of the turtle, we have to pass the commands that key triggers to the back-end so the back-end can stored the turtle's movement by storing the commands pressed key called. As this feature involves commands, I also have to ensure it applies to different locale settings. Also, I used to hard coded the key command in the program such as "W" as "forward 5" and "A" as "left 10", I decided to move these hard-coded command into a properties file so that it is easier for user to modify and define their own shortcut key commands.

I refactor the class significantly and create the file in resource.parsing "DefaultKeyCode.properties" which records shortcut key code, instructions, and the parameters the instructions take in. The user

can now easily adjust the use of shortcut key according to their preference. I also refactored all the duplicated code by either making a method or creating an array and iterating through. All the magic number and hard-coded string are made into constant. In general, I think the code masterpiece is now well designed.

JUnit

I work at the front-end and mostly in view-package so I would like to test the front-end part. However, it is hard to test GUI components in JUnit, and the test is mostly going on by running the GUI, I would like to test the language implementation. I will test the locale, the front-end display and the back-end CommandsList class to see if once the locale is changed, they are implementing the language correctly. I would like to test the command map to see if the commands has switched to the other langauge by reading in another language file, and if the key event triggers the command in right language.
In my current implemented JUnit, I test if the label on the language choosing menu returns the right string representing the selected locale. However, the JUnit gives error saying that toolkit error occurs during the execution. After doing online search and communicate with group members, we find we have the same problems running the JUnit because the class we implemented is a javafx components. However, I believe the test will give the right answer once the toolkit error is cleaned as the whole program runs correctly with different language settings.