Project Journal

Time Review

* Personal hours:
  + Start time: September 29
  + Finish Time: October 26
  + I spent about 80 hours on this project.
* How I spent my time: Most of the work was done together as a group. What I worked on individually was minor implementation details, reading, or coding new features.
* I personally managed code by developing it, testing it from a UI standpoint (as in making sure that what I coded rendered as desired and was relatively robust to user inputs), debugging (in case of errors or unexpected outputs), and then push.
* Tasks:
  + Easy tasks – implementing my own hierarchies, documenting my own work, communicating my progress/developments to others
  + Difficult tasks – the hardest part was when I jumped into the backend to try to implement Query expressions during rush time. I felt like a lot of this was not good use of my time as I wasted some time just trying to understand the data flow in the back end and it took me a while to understand the simple solution I needed to program. Unfortunately, this was done out of necessity.

Teamwork

* Planning & Design – these meetings occurred throughout the first entire week and into the second week. We would have the equivalent of design review meetings about once a week – we always preferred working near a whiteboard so we could flesh out details such as communication or hierarchies/design patterns. I would say 20-25 hours of our project was spent on planning & design.
* Delegated Roles/Primary Responsibilities:
  + Abhishek – front end, command windows
  + Jonathan – front end, turtle animation
  + Will – back end, interpreter/parser
  + Tanaka – back end, model
  + Regarding how much work was put in, I personally felt that Jonathan’s role stretched out to developing a lot of the Model to work with his front end design in animating the turtle. I also felt that Tanaka did not put in as much as the others, and that a lot of the work on the Model was a collective effort rather than his own.
* Communication – I think there was sufficient communication for the most part. However, towards the end, a huge issue came due to lack of communication regarding both the status of serialization (Tanaka’s segment for saving/loading workspace). I personally felt that because of the lack of clear progress updates on this front, it was hard to gauge how much was done and at the end I felt that we were unnecessarily scrambling to integrate what little of that portion worked.
* Team’s plan for completing project – I felt that our roles and delegations remained consistent throughout the project. In addition, having separate regions of the code helped a lot in progressing separately. There were very few moments of merge conflicts/deadlocks in this project.
* Extensions – I think our plan held up fine with the extensions except for the fact that a lot of the extensions became overwhelming for our back-end teammate. I think because the load was not distributed equally towards the end, it would have been in our best interest to wrap up the front end changes as soon as possible so all of us could jump to back end.

Commits

* 67 commits total
* My comments for the commits have become more descriptive, and I tried to push only when a significant amount of code or feature was completed (thus the commit message would reflect that). However, I still made sure to push often, at least once a day when working, and also towards crunch times when we were all working together and there were lots of dependencies, I would push more frequently. In these cases, sometimes the commit message would just be “refactoring”. As a whole though, I believe these represent my contribution to the project.
* Three commit messages:
  + “Refactored to add abstract classes to all window view controllers that have lists”
    - Purpose – this was a new abstraction developed to get rid of duplicate code after we go the ScrollPane working in some of the command windows. This applied to the User-Defined Variables and Methods lists as well as the History window. The list was a common feature to all of these, so that shortened each individual class.
    - Merge Conflicts – none
    - Was the commit done in a timely manner – yes, it was done immediately after the ScrollPane was implemented and also had no impact on any other components of the project.
  + “Finishing up querying expr”
    - Purpose – I began to work on implementing all query-type expressions. At this point, I had concluded writing the code for each individual query-type expression and testing them.
    - Merge Conflicts – none
    - Was the commit done in a timely manner – yes. In previous commits, I had pushed the general QueryExpression class. In this commit, I pushed implementation of extensions to that class. Overall, this was done in two chunks so it reduced the amount of code pushed up at once. Again, this had no impact on any other portions of the project because it was an isolated group of commands.
  + “Help URLs are now backed in the GUI Reference Library”
    - Purpose – this was to remove hard coded links from the code and separate them in a nice manner in the resource files. This was done by splitting up the link into a prefix and ending.
    - Merge conflicts – none, timely – yes and also no dependencies on this code

Conclusions

* Under-estimated the size of the project – not from the front end perspective but from the back end extensions perspective. It seemed that popular belief was that the front end’s extensions were much more challenging; however, I felt that our front end was designed very well and that the back end just had too many requirements/extensions that it was not feasible in the given time period to finish them. In the future, it would be good to implement a general case for everything as soon as possible to gauge the time required so we can budget time better.
* Yes – I took charge of half of the front end, and set up the communication between the front end and the back end as well as inter-component communication in the front end. Towards the end, I began helping out with features in the backend (such as implementing one group of expressions). I also made a fair amount of contribution to the model. I kept my team updated with my progress by sending a specific message to the group whenever I pushed a new feature or a significant amount of code.
* To be a better designer:
  + Start – thinking more in terms of design patterns. I think during this project my teammates helped opened my eyes more to the amount of patterns that can be present, and so I will definitely be thinking in these terms more for future designs.
  + Stop – thinking about implementation details at the start. As we got into this project and had to reconsider designs, I definitely stopped thinking about the specifics of implementation because I realized that those details are easier and meant for me to figure out. I am working on focusing on a higher level on what the overall pattern is or how the data flow will work.
  + Keep doing – meticulous planning. What helped us in SLogo was 1 week of just drawing out on the whiteboard everything that came to mind. This helped me understand the backend requirements and get a sense of how the parsing was going to work as well (we together came up with a two-stack system to separate expressions from things that were already evaluated and are now parameters). When I went to design my side of the GUI on my own, drawing out the ideal diagram helped me extract common features from different components. Ultimately, this led to me
* To be a better teammate:
  + Keep doing – communicating, making time, having specific questions for members of the group, setting milestones for myself and the group (whether hourly or daily checkpoints to cross).
  + Start doing – do more outside of meetings. This project, we did a majority of our work together. While I did set up a lot of foundational elements on my own, I think I could have done more outside of group meetings to prep.
* If I could work on one part to improve my grade, it would be exceptions/error handling. We set up the basic framework for this, but beyond that, we were unable to implement it across the board. This would help make our program easier to run and test from the user perspective – right now, when running commands in the prompt, if one mistake occurs, especially if the user types nonsensical inputs, then the whole program breaks when instead it should just handle that mistake and keep allowing the user to enter commands.

Design Review

Status

* The code is generally consistent in naming conventions. The same style and general layout is consistent – variables declared at the top, comments describing each method.
  + There are obviously exceptions to this here and there. For example, the style is slightly different between the GridViewController and the CommandWindowViewControllers. However, the same level of descriptiveness is present in all classes.
* The most key aspect of this is that the variable/function names are descriptive/speak for themselves. This has allowed us to limit commenting to just a summary for each method.
  + For example –

|  |
| --- |
| **public** **void** **setLanguage**(**String** language) { |
|  | myLibrary**.**setLanguageAndReferences(language); |
|  | } |

* Dependencies are easy to find:
  + Front end – all components of the front end communicate with each other through one front end container class, which is also the portal to pass data to the back end.
    - Regarding order of methods, there are a lot of components that are initialized in the view. These have a specific order that is abstract away from the subclasses. Therefore, the order of the methods in the subclasses does not tend to matter too much. Since Lists of data are being added to a pane or for an HBox/VBox to watch, these data structures can change at any point and will still be refreshed/updated in the View.
  + Back end – dependencies on other methods are explicitly named and attached as methods to instances of that class. For example, expression.loadArguments or expression.evaluate – it is very clear what is happening here.
    - An example of order of methods and dependencies is in the Interpreter. There is a clear order that is defined for data flow.
* Implemented features are easy to extend – the view is easily customizable.
* The backend logic with running each expression and updating the result/transition state as necessary is easy to test. The GUI code is not as straightforward to test.
* Yes, I found a bug in two parts of the code – first, a special case of looping was not working due to a bug in the REPEAT command. In addition, the animations were not working well with wrapping around, and sometimes when wrapping around a lot, the grid screen will jitter too much.
* First file to review – CommandReferenceLibrary.java, within the Interpreter package
  + Why I chose – this is a very significant piece in the interpreting segment.
  + What is interesting – there are lots of Maps here, especially reverse syntax maps. It seems like this is an option for moving back and forth with names. For example, forward maps to fd and vice versa.
  + Suggestions for improvement – if possible, merge some of the Maps or figure out a way to reduce redundancy of having so many different Maps. Along the same note, there are lots of plain get methods, and if these get methods could be tied with some computation, then that might lead to better design regarding openness of data.
  + What would be required to make this usable outside of SLogo:
    - I think the concept of a Reference Library is valuable and can be extended to other projects. Because this Command Reference Library is very specific, it would need to be generalized to work with any set of reference files. However, the point is that if there is a lot of data that is backed up by properties files, then using this mapping will be extremely helpful.
    - To use this in Cell Society, this could essentially be used to map different user input strings to different simulation modes, if combined with a Simulation Factory.
* Second file to review – TransitionFactory.java, within the Turtle/Draw subpackage
  + Why I chose – this is the core of generating a transition state for the turtle to move
  + What is interesting/what is good – this class is organized very well and is very readable. It also seems fairly simple in that it has broken down transitions into just combinations of translation and rotation.
  + Suggestions for improvement – I think this class is good. I think the one thing to improve this would be commenting to clarify what the inputs to createAnimation are, but even this is clarified at the top so there aren’t many improvements that could be made for this class.
  + What would be required to make this usable outside of SLogo:
    - TransitionFactory is applicable to any GUI that has to animate by changing the state/position of what is on the screen. Generally, these transitions can be accomplished through merely translation and rotation.
    - In the case of Cell Society, using a Transition Factory could give more of a dynamic feel to cells spreading. Rather than having a binary feel with just switches turning on and off, this could draw more of a path to the cells motion.
* Third file to review – Deserialiser.java, within the Model package
  + Why I chose – this is a feature I am not too familiar with in the project, related to saving/loading workspaces.
  + What is interesting – this seems to be entirely based on ObjectInputStream, which is a new concept relative to other classes. The model gets saved all at once, and a lot of work is saved from being done.
  + Suggestions for improvement – there is not much explanation to the mechanics of this Deserialiser. Furthermore, the catch blocks do not seem to be the optimal solution to responding to the potential exceptions.
  + What would be required to make this usable outside of SLogo:
    - Deserialiser again is applicable to any program that has state that needs to be stored. Because the premise of the Serialiser/Deserialiser combo is that the models being stored need to be Serialisable objects, the biggest change that would need to be made is to tweak models so that the data is serialisable.
    - For CellSociety, there is a clear distinction between the front and back end of the program. In my CellSociety project, a lot of the Model was just grid data structures and particular state variables associated with each space of that structure. Therefore, as long as these are serialisable, then the Model could be saved, and this would allow for users to pull up a simulation midway. This would have been a valuable asset to my CellSociety project had I been aware of this kind of tool.

Design

* Overall design: The highest level is the concept of Workspace, which includes an instance of the two project splits –front end (ViewController) and back end (Model).
  + The front end consists primarily of container hierarchies that go down to the smallest level (a single unit of the view). Each class in the front end view extends the ViewController interface, and breaks down into smaller components that pass their node (usually a BorderPane) up to the parent container. The front end has two primary sides that are contained within MainViewController.
    - The grid containing the turtle’s display, animation, and transitions. GridViewController updates every loop by redrawing all of the turtles on its screen based on the states in the model. In terms of display, these represent the left side of the screen.
    - The command windows including the command status, prompt, history, user-defined variables, and methods windows. In terms of the display, these represent the right side of the screen, and they are contained within a CommandWindowContainerViewController class. This container class is the medium for communication between individual command windows (data is passed up through it).
  + The back end consists of the Model and the Interpreter. The Model contains all of the state of one Workspace.
    - There are many sub-model classes that represent the state for a particular component of the View – for example, the User-Defined view windows have their own models.
    - The Interpreter takes the commands and uses a combination of the Parser to break the string command into smaller expressions. From here, an Expression Factory is utilized to construct an instance of each expression via reflection. Then, the Interpreter uses two stacks to evaluate the list of expressions. The first stack has every individual expression – even constants are expressions. When an expression is popped from the stack, it loads its arguments and evaluates with those arguments. Then, that result is pushed onto the parameter stack, and every subsequent expression pops off as many results from the parameter stack as necessary.
  + The front and back end communicate with each other via the Model – the MainViewController has an instance of the model and passes the string from the Command Prompt via an open method call in the Model.
  + The front end communicates amongst itself by
* To add a new expression, the following needs to be done:
  + First, you need to determine what type of expression it is (Conditional, Math, Query, etc.).
  + Then, a class for that expression needs to be written to extend that type of Expression (abstract).
  + Then, a result for that expression needs to be written.
* To add a new component to the GUI, the following needs to be done:
  + First, you need to determine where in the GUI this component will be displayed, just to determine which parent container it will fit into.
  + Then, that component in the GUI needs to be written, just to implement ViewController interface and MainModelObserver if it is depend on the Model.
  + Lastly, any string/text in this GUI would need to be backed in the view reference/properties files in order to facilitate changing language.
* My code (for front end command windows) is designed as follows:
  + Each class extends CommandWindowViewController, which exposes that window’s pane.
  + Every string in the display is backed up in properties files. These files are accessed and read through the GUIReferenceLibrary, which just creates a static map between these.
  + CommandWindowViewController further extends into CommandListWindowViewController, another abstract class that is for any command window that is based on a list (i.e. the User-Defined Variables). CommandListWindowViewController extends into CommandClickableListWindowViewController, another class that is for any command window that requires click action on items of the list (i.e. the Command History).
  + The subclasses in this side are (from display-wise top to bottom):
    - UserDefinedMethodsViewController extends CommandClickableList
    - UserDefinedVariablesViewController extends CommandList
    - CommandHistoryViewController extends CommandClickableList
    - CommandPromptViewController extends CommandWindow
    - CommandStatusViewController extends CommandWindow
  + There are two interfaces that my code is based on:
    - ViewController interface
      * This interface is for each class to pass its Node to its parent. It also allows for translation to occur.
    - MainModelObserver interface
      * This interface is for the MainViewController to update based off of the data from the MainModel. A parent class passes the model to every child.
* My code is structured in the way mentioned above because I wanted to be able to separate the components and work on them individually. Because Professor Duvall stressed that GUI code can still have structure to it, I wanted to avoid the common practice of putting all GUI code in one big class.
  + The advantages of my approach include:
    - Modularity, a distinct hierarchy in the GUI structure
    - Very little duplicate code across Command Window components – each parent abstract class takes care of the initialization and the explicitly defined variables in the subclass are just the bare minimum to accomplish the desired functions.
  + Furthermore, this turns out to be very helpful for updating the GUI windows. Each window in the View implements MainModelObserver – therefore, rather than having each window look at its own Model, which would introduce too many entry points into the Model, the Model passes its data to the MainViewController. From here, the update is called through the visual hierarchy, and every child node is updated based on the model. This allows for each window to get the data it needs and pass it on.
  + With a disparate GUI (separated into many files, a big hierarchy), it can be hard working with resource files. However, we were given explicit permission to utilize something among the lines of a static GUI reference library, which would allow us from any class to access GUI string data and load it into the window. Doing this has allowed for no hard coded strings in the GUI. This allows for the user to select the language at the start of the program, and for the translation to be applied while loading the GUI as part of the ViewController interface.
  + The only assumption/limitation to the design is that the concept of Workspace is locked once created. In other words, our design does not support having changes in real time apply across all workspaces. In order to do this, we would need to bring the control up another level. Just to reiterate, Workspaces contain a Model and a ViewController, but Workspaces themselves are contained within an overarching View class. To accommodate for global preference changes during run time, we would need to have the user options a level higher.
* Jj

Alternate Designs

* Our front end was able to handle most of the extensions very well, mostly because we kept our design very flexible and modular from the beginning. For example, incorporating Workspaces was not a challenge because our MainModel and MainViewController were already tied together, and this was just a matter of putting another layer over that. The back end was also able to handle most of the extensions well from a design standpoint – the only thing was that we felt that there were too many instructions to implement and lots of back end requirements.
* The original APIs actually remained quite consistent throughout the project – the same methods are publicly facing between the front and back end. In terms of within the back end,