Scotty Shaw (sks6)
SLogo – Team 13
Computer Science 308

Project Journal

During the SLogo project, I easily spent 150+ hours (maybe up to 175 total) working to help my team complete our program from its start on September 29 to its end on October 26. During the early stages, I focused on the overall design of our project. This led to us adopting a model-controller-view design. The most important feature is a powerful middle comprised of a parser, translators, factories, gateways, containers, and the controller itself. I also conceptualized the Command hierarchy and implemented the booleanCommand package, including its JUnit tests. This let us check the functionality of the Boolean commands. In one of the very few changes we made to the original Command hierarchy, I later merged the booleanCommand and mathCommand packages into the expressionCommand package upon realizing they had many similarities. Of course, this required some refactoring and cleaning up.

For the last few hours of this project, I improved the SLogo linguistics. First, I filled out the missing translations for Italian and Portuguese files and saved them in **ImprovedItalian.properties** and **ImprovedPortuguese.properties**. Afterwards, I corrected spelling mistakes in the Chinese translations of the SLogo commands and saved them in **ImprovedChinese.properties**. Later, I translated the commands into Spanish and German and created shorthands (such as "haz" for "hacerveces" when translating "dotimes" into Spanish) in **Spanish.properties** and **German.properties** to earn extra credit for the team. These files are in **resources.newLanguageFiles** as well as, along with the pre-existing language files, **resources.languages**. Finally, I converted all .logo files into Spanish and German using my translations and saved them in **resources.translatedExamples**.

The rest of my effort in the project involved pair (or even trio) programming. Rahul and I implemented the parser, translators, and the command factory, which formed the basis of all our factories, while Duke developed the controller. For Part 2 of the project, we implemented our design as planned and could simply maintain it, even with new extensions. My Command hierarchy and factory followed the original designs, so our actual roles in this project did not entirely follow our original plan. For multiple turtles, variables, and views, Duke created containers and gateways to hold and move information while Rahul and I connected the InteractionCommands, ViewCommands, and ViewQueries to the frontend. We formed the core of our team and logged 450+ of the 600 total hours by working most days, including weekends.

We also worked with Petra and Jack to connect our ends and make sure our code could continue supporting them. Petra handled much of the frontend design and worked extensively on the GUI. This included buttons for many functions, the ability to add extra tabs and record the command history, and refactoring the front end. Jack worked on the frontend and connected it to the controller. Unfortunately, the frontend focused on implementing features rather than maintaining good design or creating code connections that would have allowed us to test many commands. This led to lots of last-minute refactoring and frustration, especially since we had to

repeatedly help with very trivial features. In retrospect, I should have spent more time on maintaining the communication and connection between our two ends, but we nevertheless completed a polished SLogo product as originally designed.

For this project, I committed code 116 times with varying sizes and purposes. The messages largely describe my contributions to the project, but since much of the code I worked on was part of pair or trio programming, a large amount of my efforts are also documented in many commits under Rahul's account.

"Trio Programming Command Hierarchy Initial Commit" in Rahul's account on October 6 shows that I had learned from my Cell Society team's mistakes. This commit implemented the Command hierarchy as I had designed it, allowing a stable and extensible hierarchy that changed very little. Other than adding new commands during Part 3, the only change we made to my hierarchy design was extracting the VariableCommand type out of ControlCommand and renaming LogicCommand as ExpressionCommand. Because I had designed our Command hierarchy so early in the planning phase and we had implemented it so quickly, any work involving the commands for the rest of the project was very simple.

On October 7, Rahul and I committed code under the title "Created new class for parsing the language resource file" when we implemented a command factory that uses reflection and a language file parser that converts user inputs to keywords. Another translator would convert them into specific classes for their corresponding commands. This commit also happened very early in our work, allowing us to move forward quickly with implementing the middle and the backend. The importance of using reflection in our factory is hiding the details of our overall implementation. It uses Java's ability to inspect and dynamically call classes during runtime. Extensions to our program would not need to "know" everything at the time of compiling and as a result are more dynamic.

A commit of note, "Completed reorganization of ExpressionCommand.java…" on October 21, came from my account. It was the last in a series during which I had refactored the BooleanCommand and MathCommand packages until they could be merged into a single ExpressionCommand package with one single JUnit test file for the newly reformed command type. This eliminated one of the very few deviations from my original Command hierarchy. I also debugged TurtleCommand.java so that our program could handle a series of commands entered as one input. Essentially, similar to the implementation of a linked list, anytime there is another command following the current command, we carry it out. Otherwise, we simply evaluate and complete the current command. This commit would allow us to begin testing our InteractionCommands, ViewCommands, and ViewQueries as soon as the frontend could display multiple turtles and views.

For this project, we accurately estimated the size of this project and planned accordingly because we had a strong design. I took on enough responsibility, even if it was a little uneven. After extensive work on designing the backend during Part 2, I was unable to help the frontend much with design choices and code connections. In the future, I should communicate and be aware of what each teammate is doing so that I am able to insert at any moment when necessary. To improve and be a more effective teammate and designer, I need to become a stronger coder and maintain good design practices. But overall, we worked excellently on the backend.

Design Review

For our project, the backend code is quite consistent in layout, style, naming conventions, and descriptiveness because we focused on readability. It does exactly as we expect, and only requires comments on the extensible classes and methods. As for implemented features, they are quite easy to extend, especially most commands, which are also easy to test. However, some commands, such as the ViewCommands, and the containers and gateways are not as easy to test because they require some interaction with the frontend and not just JUnit tests. The dependencies in the code are largely clear and easy to find because Duke helped us use type requirements, global variables, parameters, and more effectively.

However, our frontend is quite a mess and very inconsistent in all aspects. I unfortunately do not feel I would learn or understand much about the frontend code because very little of it followed good design practices, an expense of implementing as many features as possible without much regard to the original design we wanted. I am not even sure how to assess the readability of our frontend code. Although our frontend was low on numbers, I feel that their lack of communication and emphasis on implementing over designing resulted in unorganized and messy code that will require massive refactoring. The dependencies in the code also sometimes rely on getters rather than parameters. I do not feel that implemented features would be easy to extend.

One bug that I am aware of is that the CLEARSTAMPS command causes the program to crash. As for why, I am unclear and unable to determine why. Also, the files I translated do not all appear in the drop-down menu under Languages because this section was hardcoded. As a result, Chinese, Italian, and Portuguese are not fully functional in our program. And finally, when the user inputs an invalid command, the program simply freezes and crashes. I suspect an incomplete exception handler is the root cause.

A class in the program that interests me is BaseController.java because it is in charge of handling all interactions between the frontend and backend. Its abstract methods indicate that any classes that extend this superclass can receive commands from the View via a string, pause the turtle by altering commands sent to it, stop the turtle by erasing all commands, send errors to the View to notify users of problems without crashing the program, load the language file resource chosen by the user, add turtles and views, get active turtles, set turtles and grids as active, and save or load preferences. Using AnimationTimers allows the controller to start, pause, and stop the program as needed. With all these functions, this abstract superclass sets up everything needed for MainController.java to act as the powerful middle for our program and delegate to the frontend and backend.

The next class is BaseTurtleContainer.java. It contains all the turtles in each grid and defines the allowed behavior for each container. This includes adding and removing turtles, getting all turtles, all active turtles, or all turtles by ID, setting the status of turtles, and more. As for modifying the turtles themselves, this class is able to get and set a large amount of information, such as the position, heading, visibility, destination, etc., or modify the pen objects within the turtles because it implements

ITurtleBehavior. SingleGridTurtleContainer extends this abstract superclass, so it only has to fill in how to carry out the methods specified.

Finally, IInformationGateway.java serves as the centralized location for all information related to the program's backend, primarily for commands. This limits access to information from any extending commands and reduces the risk for error when extending commands more specific than BaseCommand. Centralization also allows the execution method signatures of commands to remain generalized and streamlines the access of data, thus limiting repeated data. The methods in this class provide the ability to add containers or to get the container corresponding to the requested container type or a collection of containers corresponding to the requested container.

Since Duke wrote these two abstract superclasses and the interface, they were extremely easy to understand. Of course, the existence of extensions shows that these classes are easy to extend and implement. To test them, JUnit tests can check various functions, and running the program itself can help ensure all backend and frontend connections function as expected. Since theses class can be extended into subclasses and implemented as needed, they are usable in completely different projects provided that they are modified to fit new needs.

They all fit into the middle of our project. As I described before, the middle consists of a parser, translators, factories, gateways, containers, and the controller itself. User inputs are parsed and translated into components that the factories can convert into commands, to which the gateways and containers are attached. Next, the controller uses the gateways and containers to determine if the commands go to the frontend or the backend or both to act accordingly. However, the frontend did not maintain the original design and ignored our turtle model, so our commands now operate on frontend objects despite using the code connections we had built. In general, to add new commands, a user simply needs to extend the corresponding command type's class. As for categories of commands, they can simply extend a higher level of command type. For the frontend, new components would extend the classes they are associated with. Both ends have factories that would manufacture the new component. Unfortunately, our backend was largely neglected by the frontend during their rush to implement as many features as possible with little regard to planning and design.

For our command hierarchies, all classes are named "*Command.java" and can be a superclass or extensions. This hierarchy makes it very easy to add and define new commands. The entire hierarchy is closed, and the Command Factory hides implementation details from other users without compromising extensibility and flexibility.

As for our parser, it also works with the Command Factory, and makes use of the superclass PropertiesFileReader.java and CommandToClassTranslator.java to do its work. PropertiesFileReader.java is able to extract the name of the file and create a resource bundle that contains locale-specific objects, such as a String, which can be loaded when needed. The program becomes largely independent of locale and can isolate most locale-specific information in resource bundles. This allows translation in SLogo to handle multiple languages and to support even more locales with easy modifications. The parser therefore is able to limit the leaking of information while

maintaining flexibility. Extensibility is also easy because there is not much the user would have to modify. Of course, because our current parser extension specifies the valid grammar any user input must have, another grammar system would have to extend from PropertiesFileReader.java itself.

As for our information gateways, Duke designed them to be closed, yet easily extensible. New gateways can implement IInformationGateway.java, which limits information access in the extending commands. This limits the risk for error when commands, such as TurtleCommands or ExpressionCommands, extend from BaseCommand. With a central point of information, command execution method signatures are more general, and the interface streamlines how data is accessed to limit repeated data.

Our original design handled the extensions to the backend and controller very well. As discussed throughout this analysis, the Command hierarchy was very extensible and capable of handling the new commands. However, the frontend did not handle them well, which resulted in a compromised design. As a result, our true SLogo is very different from our original API. As opposed to a model-control-view design, the frontend largely ignored our turtle model and duplicated a turtle on their end. We also had to add containers and gateways to handle multiple views, turtles, and variables, which was not an original consideration. Our parser and translators had to expand some more than expected, but it was not a problem.

Our first major design decision was to adopt a model-control-view design for SLogo. After discussing how the backend would decide where to send commands, I suggested creating a middle to act as a brain for the program. Similar to a human brain receiving, processing, and responding to stimuli, I felt it would make sense to have the middle parse user inputs, translate and manufacture commands, and send the commands to their respective ends to operate. Conveniently, reflection would help hide implementation details, and resource bundles would help prevent the leak of information. This led to a much weaker and smaller backend and frontend, but it allowed us to consolidate much of the information in one location and more easily protect it while maintaining flexibility.

Our next decision was to eliminate the frontend turtle. We originally planned to have two turtles, but talking to our TA Cody led us to using just one. The plan then became to have the middle do the “thinking” and let the backend model drive what appears on the view. This meant requiring lots of code connections to maintain the lines of communication in our program, but it would be much simpler because all the information about the turtle and its various objects, such as the pen, would be stored in one place. The use of parameters would allow only necessary access to the frontend, which would be able to translate information if necessary. One example is the discrepancy between (0, 0) in the model indicating a turtle in the center of the view, where the same coordinates would place the turtle in the top left corner for the frontend. This would be offset with a translation that adjusts the turtle’s position accordingly.

The last major decision was Duke’s work to limit information from leaking unsafely. This allowed us to maintain our Command hierarchy and preserve each command’s ability to operate as needed on whatever end it reached. The gateways also generalize the command execution method signatures and streamline how data

is accessed to limit repeated data. This is all possible through having a central point of information, and this decision led to him adding resource bundles to our parser. The resource bundle would contain information that would only allow necessary access and prevent information from leaking out. This later allowed our program to support translations and handle multiple languages Extensibility also became easier because there is not much the user would have to modify.

In all three cases, I am glad we adopted those design decisions because they significantly improved our backend and middle, which greatly contributed to our overall design. However, there are some things I would have liked to improve. We did not have enough time to finish consolidating our gateways to push code into the highest levels, and our backend exceptions remain somewhat underdeveloped. In spite of this, the only bug I am able to find is that the CLEARSTAMPS command causes the program to crash. The translation maps were not implemented correctly in the frontend and would simply need the corresponding information to be mapped and spelled correctly. The third issue with our implementation is that the frontend hardcoded the addition of language files. As a result, the files I translated do not all appear in the drop-down menu under Languages. I would use a mapping system similar to how we store commands to hold information on the various languages and pull translations as needed.

Code Masterpiece

For my masterpiece code, I am choosing to display our ExpressionCommand package's JUnit test class **ExpressionCommandParseTest.java** and our superclass **ExpressionCommand.java** for the ExpressionCommand command type. However, for my masterpiece, they have been renamed as **LogicCommandParseTest.java** and **LogicCommand.java**. The extending subclass OneExpressionCommand.java and TwoExpressionCommand.java were then renamed to fit the naming system I created in our original Command hierarchy. **OneExpressionLogicCommand.java** and **TwoExpressionLogicCommand.java** are the names I prefer. Of course, the package **commands.expressionCommands** is now **commands.logicCommands**.

This was code largely designed, refactored, and written by myself, although it later incorporated MathCommand code written by Duke. The JUnit test allowed us to check on the validity and correctness of each ExpressionCommand as we waited for more frontend development. The primary bugs we wanted to detect were if the calculations were done correctly. For example, if the two expressions are equal, then **EqualCommand.java** should return 1, whereas **NotEqualCommand.java** should return 0. The JUnit tests checked the functionality of the commands for us and let us know that the **logicCommands** package was properly and entirely implemented.