**Team: David Chou, Kat Krieger, Richard Cao, Jeremiah Siochi**

**Introduction**

Describe the problem your team is trying to solve by writing this program, the primary design goals of the project, and the primary architecture of the design.

This section should be approximately three paragraphs and discuss the program at a high-level (i.e., without referencing specific classes, data structures, or code).

**The overarching purpose of this program was to create an integrated development environment that lets users control a “turtle”, an object that would carry out user commands. The main challenge of the project was that labor was divided so that it wouldn’t be possible for any one group member to know the implementation for every part of the project. In this project, the front-end and back-end would have to work together based on design principles agreed upon during the planning stages of the project. Any spontaneous changes to the design could possibly break the project, which is why we were supposed to stick to the original API document as much as possible and carefully record any discrepancies. The split between front-end and back-end emphasized the principle that Java classes should know as little as possible about the implementation of other classes while still remaining functional. In addition, the project requires that you utilize different ways of testing from the usual compile-and-run method, since it was often the case that the front-end and back-end were not at the same stages.**

**The main design goal of the project was to design a simple API that allows other programs to use and extend our implementation.**

**Our design architecture follows the traditional Model View Controller (MVC) pattern. The Model, or back-end, would contain all the internal workings of the API, while the View, or front-end, is the user interface portion. The View is the output representation of the user input, while the Model is “hidden” from the user. These two components work in tandem through the Controller, which updates both Model and View based on user input.**

**Overview**

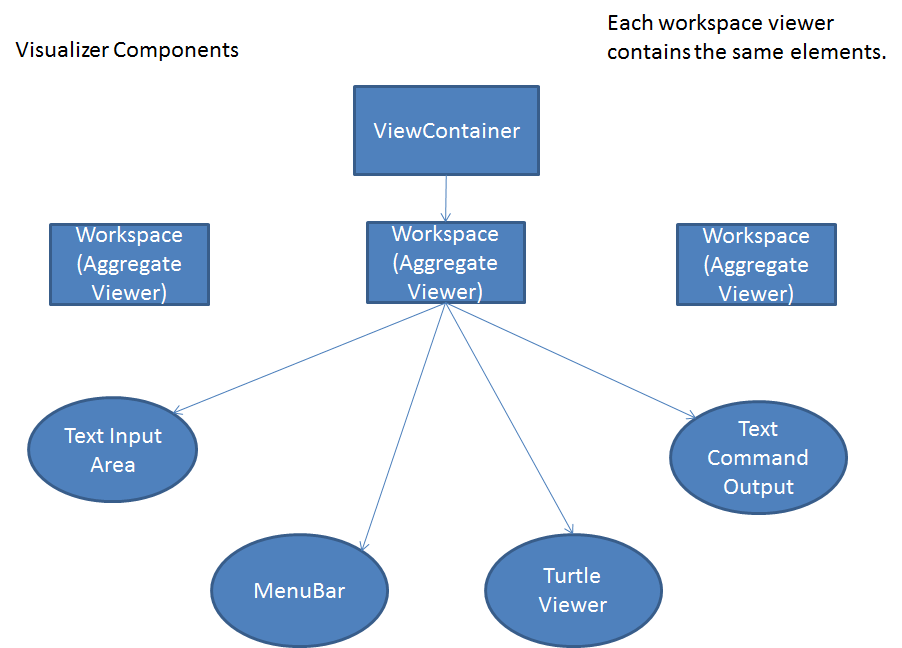
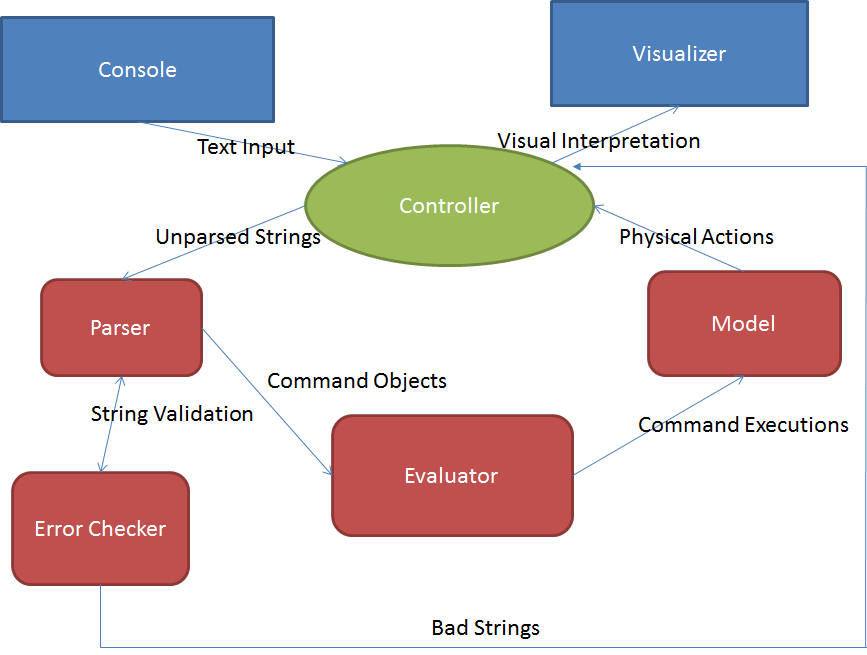
Describe the primary modules in your project, their purpose with regards to the program's functionality, and how they collaborate with each other. This section serves as a map of your design for other programmers to gain a general understanding of how the program was divided up and how the individual parts work together to provide the desired functionality. A module may represent a single class or a group of classes related in some standard way, such as set of subclasses. It should also include a picture of how the modules are related (these pictures can be hand drawn and scanned in, created with a standard drawing program, or screen shots from [a UML design program](http://green.sourceforge.net/)).

This section should be approximately two pages and discuss specific classes, methods, and data structures, but not individual lines of code.

The primary modules of our Slogo project are the Viewer, TurtleViewer, Parser, Commands, TurtleModel, and Controller. Work on the program was divided into work on these modules, as they provided a good degree of separation for extensibility, changeability, and testability. The front-end modules include the Viewer and the TurtleViewer modules. The TurtleViewer module is responsible for drawing the turtle on the screen, and the Viewer is responsible for every other part of the user interaction. The TurtleViewer gets its information about where and how to draw the Turtle from the TurtleModel module, which represents the properties of the Turtle and Pen, such as position, angle, and pen state. This module is also moves the turtle by changing the Turtle’s state. The commands to move the turtle come from the Controller module, which loosely interfaces between the front-end and back-end modules. The Controller’s primary function is to receive the Command objects from the Command module that are passed into it, and relay these commands to the appropriate target object.

When the user inputs a text program to be run in Slogo, this program is passed to the Parser module via the Controller, and the Parser decodes the text and renders it as a syntax tree of Command objects. Each Command object is linked to a specific command in the controller. The Parser module also then calls execute on every Command in the Command tree, which in turn updates the TurtleModel via the Controller or communicates with the front-end Viewer and TurtleViewer via the Controller.

To see the UML diagram for this program and the Module interaction, the diagrams below reference [slogo\_UML.pptx](https://github.com/duke-compsci308-spring2014/slogo_team07/blob/master/docs/slogo_UML.pptx) in the slogo\_team07/docs folder.

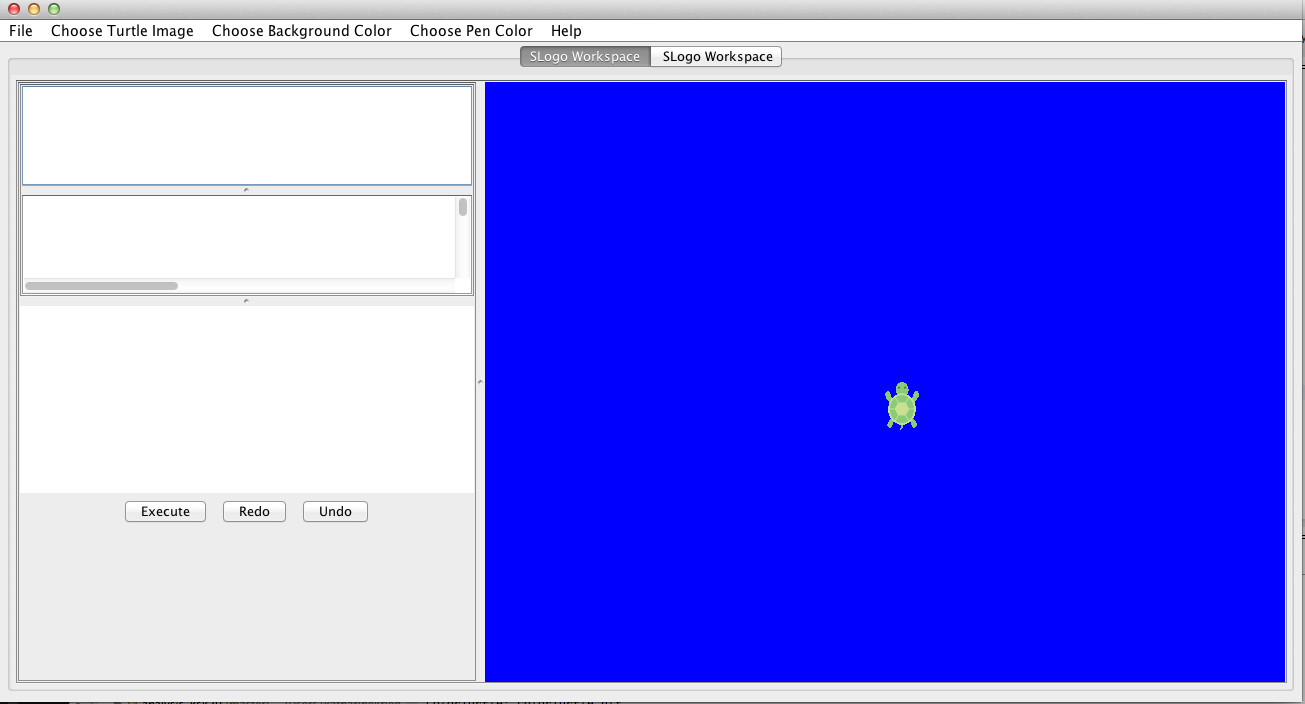


**User Interface Design**

Describe how the user will interact with your program, the overall appearance of program's windows and how users interact with these windows (especially components specific to your program, i.e., means of input other than menus or toolbars). It should also include one or more pictures of the user interface (these pictures can be hand drawn and scanned in, created with a standard drawing program, or screen shots from a dummy program that serves as a exemplar). Finally, it should describe any erroneous situations that are reported to the user (i.e., bad input data, empty data, etc.).

This section may be as long as it needs to be and go into as much detail as necessary to cover all your team wants to say.

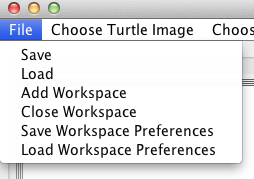
Our user interface is very standard. It contains a menu bar along the top, a TurtleViewer on the right, and a panel of text areas. The goal of our view space was to have an organized and easy to use space for all users. Below I will outline each section, how the user interacts with the parts, and then how they connect to the backend.



The overall structure of the View field is a JFrame. It has one menu bar and multiple tabs for the workspaces. The execute, redo and undo commands are all controlled by buttons below the lowest text field. This is where the user types commands and executes them. In the middle text panel, previously executed commands appear, and they are clickable. In the top panel, if the back end were connected appropriately, variables that were defined would appear there.

The panel to the right shows the turtle, and demonstrates its movements. Again, the turtle doesn’t move on command because the back end work was never completed, but when execute is pressed the moves written in the text box would be executed to the right. Redo would automatically repeat the last action, and undo would reverse the last move.

The menu bar contains all other functionality. The File menu allows users to save and load commands, add and close workspaces, and load and save workspace preferences. The only way to load workspace preferences is after previously saving workspace preferences. Pen color, background color and turtle image are also controlled by radio dropdown menus. The same menu bar controls all the panels, so currently there is a bug where the radio buttons don’t always match the environment, but the menu does accurately change the active tab. The help menu opens the website to the SLogo rules as directed by the instructions.



The coding design of the UI is a group of classes that exist within one another to create the GUI. The highest level and overall shell is a JFrame called ViewContainer that has the menu bar, and creates different tabs of workspaces within it. Each workspace is an AggregateViewer which is a JPanel and contains text areas, JSplitPanes, Turtle’s, TurtleViewers and JLists, to make up the picture above. Each of their functions and overall uses will be described below.

The ViewContainer manages the different workspaces and the MenuBar file. The MenuBar is where all actions and changes take place, from background color, to turtle image, to pen color. Within the ViewContainer, there is a Map that contains AggregateViewers (workspaces) and their tab number. When the controller needs to access a workspace, it finds the active tab within the ViewContainer and is able to access the AggregateViewer it needs through the Map. Within the ViewContainer Tabs are removed and created by adding them to the map, and updating the tab index when they are removed.

Another class is the TextInputArea which is the bottom left corner of the viewer. It contains a JPanel that has a TextArea, and a panel of buttons for Execute, Undo and Redo. There are methods in this file for saving and loading text into the TextArea (loading and storing instructions), and methods for calling execute, undo and redo.

The MenuBar creates all of the dropdown menu’s in one place, and adds all their action listeners.

The code is very straightforward, but is slightly repetitive. In hindsight, I would have liked if we implemented a ViewFactory before writing the MenuBar. The idea for our ViewFactory would have been to create buttons and dropdowns through the ViewFactory so that methods would return the objects and components we needed and we wouldn’t have to create buttons repeatedly in the other files. We were already done with the Viewer by the time we were given this suggestion, so eventually decided not to implement it, but will be integrating that concept into our work going forward.

Each botton or action that is started through the UI is executed through the same chain of events. The button calls a method within the controller, which then cascades the message through a stream of methods within the classes until it reaches the place it needs to be. After the controller, it goes into the ViewContainer, which determines the active tab and then enters that AggregateViewer. From there, if it is to change the TurtleImage, it would go into the TurtleViewer and then the Turtle model itself. If it was to put text into the input space, it would enter the TextInputArea and call a method within there.

**Design Details**

Describe each module introduced in the Overview in detail (as well as any other sub-modules that may be needed but are not significant to include in a high-level description of the program). Explain each module's purpose in the program, i.e., how it handles specific requirements given in the assignment handout, as well as what resources it might use, how it collaborates with other modules, and how each could be extended to include additional requirements (from the assignment handout or discussed by your team). Finally, justify the decision to create each module with respect to the design's key goals, principles, and abstractions.

This section may be as long as it needs to be and go into as much detail as necessary to cover all your team wants to say.

Parser: The parser module handles the recognition of commands and syntax. It translates the text input it receives into a list of Command objects. The parser can handle complicated nested functions due to the recursive implementation that treats all syntax as a command, including single numbers. Once the parser has moved through a user’s code input, a resulting list of Command objects will exist that corresponds to the syntax tree of the user’s program. The Evaluator class, as part of this module, simply calls the execute() method on each Command, and this execute directive is passed down the tree until a leaf is reached (i.e. a NumberCommand, as it has no reference to other Command objects). The justification for separating the parsing and command modules is that it allows for command implementations to be changed without affecting the parser - an abstraction that saves time in many ways, including adding new commands and testing the parser’s functionality.

Command: For every command listed in the SLogo project instructions, there’s a class for that Command. The Command interface contains one method, execute(), that all commands implement. The Command interface is implemented by three abstract classes, ZeroParamCommand, OneParamCommand, and MoreThanOneParamCommand, and NumberCommand. All the Slogo commands then extended their respective abstract classes based on the number of parameters that they are supposed to accept. The execute command carries out the command’s function, whether it’s calling the appropriate Controller method or returning a certain value based on the input parameters. The NumberCommand is unique in that it simply returns its own value. We did this so that the parsing process could be streamlined. The input parameters for certain commands can be a list or tree of other commands that are recursively processed so that the root command has the proper input value.

TurtleViewer: The turtle viewer module allowed for the display of the turtle’s actual movements. It received information from the model and each time a program was executed, the turtle’s movements, trail, and rotations were updated. The turtle viewer module acted as the display that would show all of the user’s settings - the turtle image, the background color, and the color of the pen to be used. In order to allow the user to pick these specifications, it was necessary to allow for these options from properties files that were located in the source folder. TurtleViewer’s methods to update turtle visuals come from the AggregateViewer, which has access to all the other visual aspects in the front end. It was necessary to separate each visual component because of the way swing and awt created Java panes and listeners, but it was particularly necessary to create a separate TurtleViewer in order to separate the model from the view. The TurtleViewer module acted as a way to hold all of the necessary information about the turtle in respect to the background, and it did not need to know any of the actions that were being performed on the model, but only the final result of the actions. It was then able to update the status of the turtle after each program had been performed.

TurtleModel: The turtle model held all of the abstract information about the turtle’s location, angle, and points it had reached. It also performed all calculations when the methods were called and then directly passed the information to the Controller, which would send that information on to the ModelViewer. The Model received all of its information from the Parser’s execution, so the only real calculations that were formed were for the change in location based on the turtle’s angle at the initial location. All command executions such as “sum 50 30” were performed within the Parser as part of the execution process.

*Document last updated at 11:48 am on March 22, 2014*