SLogo API - Team 9

Chase Malik
David Zhang
Timesh Patel
Kevin Button

Design Goals

The basic breakdown of our API is as follows: (described in more detail below)

Front-end
Modules
- Canvas for displaying API
- User interaction (Commands, Toolbar/buttons)
- Program feedback
- Main Application

Back-end
Modules
- Parser
- Executing commands

For our design, we chose to divide the project into two main modules, a front-end user interface and a back-end model. Within these two modules, we've divided the project design into sub-modules.

For the front end, we've divided the module into a feature superclass and a main application class. Within the feature superclass, we've chosen to divide GUI objects into user input and program output, with each most likely implementing a specific interface.

For the back-end, we've divided functionality into parsing user input and executing commands. The parser will be able to identify commands (such as 'forward', 'sum', …) because they are located within the resource file. It will then create a tree/graph of nodes that represent each of these commands and their inputs will be represented by the children of the nodes. Each node will contain a command and its children. The subclasses for the commands will each implement the abstract execute command, which acts on a specific actor. This design is extendable because you can easily add more commands by creating new subclasses.

Example Code

Front-end:

```
If ( Click ){
    if ( Click.OnRun ){
    String s = TextCommand.readInput();
    myParser.parse ( CommandArea.readInput() )   // read in ‘fd 50’
    }
    // View will be updated by model
}
```

Back-end:

```
Parser
parseCommand(String s){
String[] words = splitWords(s); // split s to get all words
Node node=makeTree(words); //makesTree by putting each word in its own node
//which has a command and passes in the actor
node.evaluate(); //executes the command in the root node which will execute all
//its children nodes to work
}

Node

evaluate(){
    return myCommand.execute(myChildren);
}

ForwardCommand
public int execute(List<Node> inputs){
    int distance = inputs.get(0).evaluate();
    for(Actor a : myActors){
        if(a instanceOf Turtle)
            int oldX = a.getX();
            int oldY = a.getY();
            int angle = a.getAngle();
            a.setX(oldX+Math.cos(angle));
            a.setY(oldY+Math.sin(angle));
```

```
                a.update();
        }
        return distance;
}

Turtle class

public void update(){
        myView.updateActors(new TurtleInfo(this));
}

TurtleInfo class // Passive class with turtle information and getters

        [inside TurtleView class]

        update(TurtleInfo turtle){
        VisualTurtle vTurtle = myVisualTurtleMap.get(turtle.getID());
        vTurtle.setPosition(turtle);
        }
```

Alternate Designs

One alternate design we considered was passing the turtles to the view, as opposed to passing a passive turtle info class. We decided to pass the passive class because it limits the exposure of backend classes with the frontend (i.e. the view would have been able to call methods on the Turtle (such as move) otherwise). The problem with having a passive class is that if we extend the Turtle class we would probably also need to update the info class. This also allowed us to have only one update method vs several so we would not need individual updatePosition(), updateAngle(), etc... Other design alternatives we had were where to update the View. We debated updating the view in the main application vs in the parser or turtle. This allows more separation between the frontend and backend but ultimately is very limiting in extensibility because it becomes much harder to have multiple views and have specific turtles with specific views. Another design choice involved putting keywords inside the parser but we really thought the parser should not change. So to add new keywords instead of extending the parser new classes will be added and then just added to a list in a main class. That way parser will never have to change.

Roles

Front-end:

For the front-end of this project, David and Kevin will work to implement the classes and methods listed above. Both group members will meet and work on the main layout of the GUI as well as creating a basic Feature hierarchy for the items that will populate the GUI. Then, individually, David will work on features that handle user input or user-focused messages, like buttons, menu bars, errors and command history. Kevin will work on creating the window where the turtle moves around, as well as creating any graphic objects that will be displayed in the turtle window, like the pen or turtle.

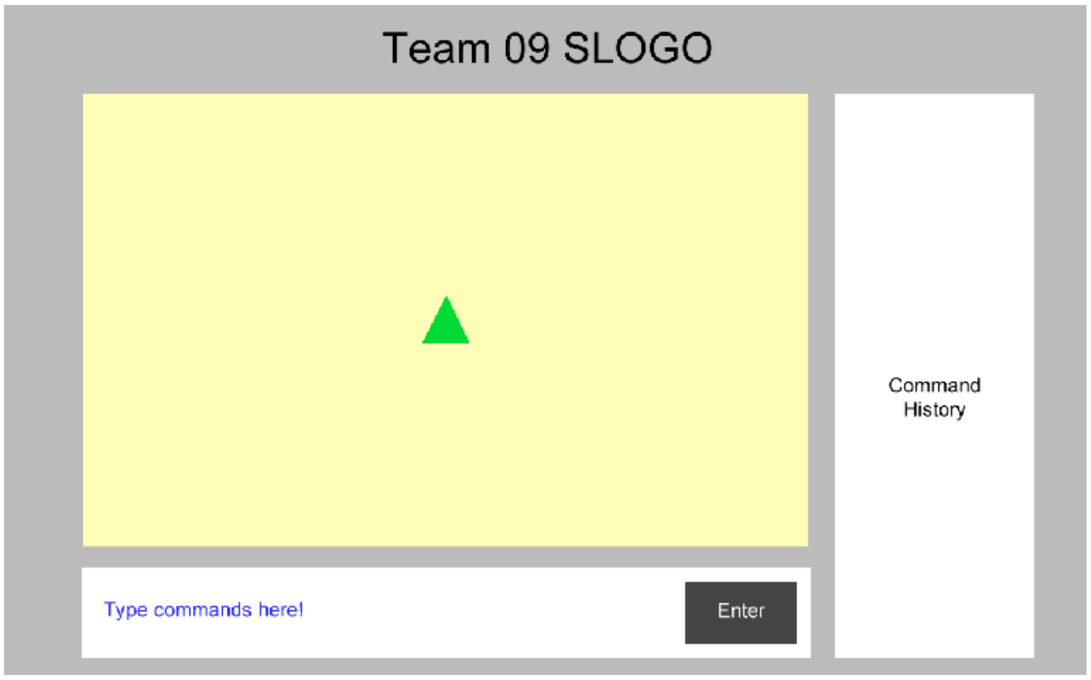
Back-end:

Tim will take primary responsibility for the parsing and Chase will take primary responsibility for the actors. There will probably be a lot of crossover in our work and we will probably end up pair programming a fair amount of it together.

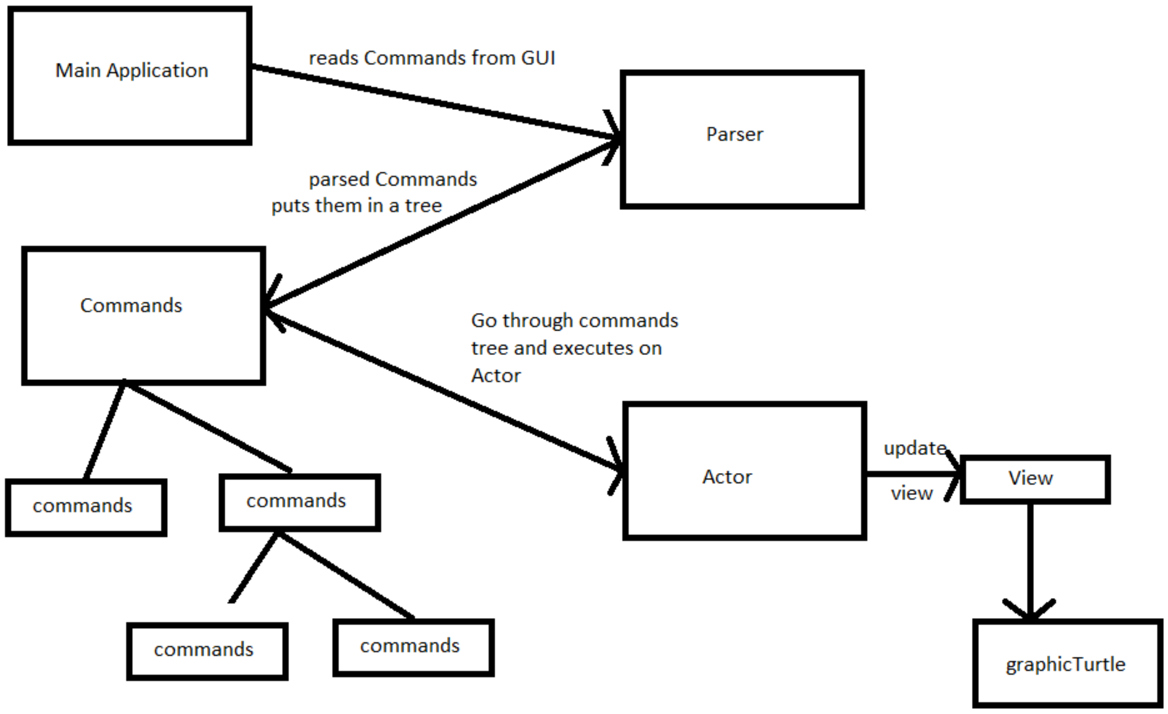
-

Diagrams

User Interface



Module Relationships


Main Application
reads Commands from GUI
Parser
parsed Commands
puts them in a tree
Commands
Go through commands
tree and executes on
Actor
Actor
update
view
View
graphicTurtle
commands
commands
commands
commands