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 keywords (such as 'To', 'Repeat', etc) because they are located within a keyword class hierarchy. The subclasses for the keyword class will store a specific keyword as well as actions to perform for that command. This design is extendable because you can easily add more keywords.

The commands will be executed through a hierarchy that will have basic implementation of a Turtle. This turtle class will contain methods based on the commands given. Currently, to add more commands, you will need to add methods to the turtle class.

Primary Classes and Methods

Front-end Classes

- Main Display - This class will hold the layout of each of the implemented features within the GUI.
- Feature Hierarchy - This interface will be used to compose lists of each item on the GUI for updating and positioning for view.
    - Buttons - This will be a superclass of all input buttons in the GUI.
        - Menu buttons - Self-explanatory, class for buttons that will be found in the menu bar.
        - In-game buttons - Class for buttons that will be found throughout the GUI not in the menu bar.
    - TextArea for input - Class to handle typed commands by user and send them as strings to the back end parser for parsing.
    - TextDisplay superclass - Superclass for all items that will display program output text to the user.
        - Command History - This class will hold a list of all previously executed commands, for display and re-execution.
        - Errors - This class will catch and display any errors thrown by the model.
        - Current Variables - This class will receive from the model and display current variables and their values within the GUI window.
    - Turtle Window - Class to display turtle's motion.
    - Turtle - Turtle graphic object that is a visual representation of the Turtle backend object.
    - Pen - Pen object to mark where the turtle has traveled, toggled on and off by user.

Back-end Classes:

- Parser
    - parse method // Looks for keywords
    - update method
- KeyWord // extendable
    - Individual Keywords (To, repeat,…)
        - update method
- Actor // extendable
    - Turtle
        - Contains basic commands
            - forward
            - right
            - left
            - back
            - etc...
        - Contains Turtle info // position , angle etc..

Example Code

Front-end:

```
If ( Click ){
        if ( Click.OnRun ){
        myParser.parseCommand ( CommandArea.readCommand() )   // read in
        ‘fd 50’
        }
        // View will be updated by model
}
```

Back-end:

```
parseCommand(String s){
String[] words = findDelimiters(s);
interpretNonWhiteSpace(words);
reflectToMethod(myTurtle, command); // calls myTurtle.forward(50)
}

Turtle class:

        forward(int x){
        myX=appropriate calculation
        myY=appropriate calculation
        myView.update(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 the turtle and pen objects that will be displayed in the turtle window.

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.

Front-end API

External API

public void update(List<TurtleInfo>)

- method called by the backend that passes a list of Turtle objects which will then be updated in the GUI

public void clearScreen()

- method that instructs the main TurtleView window in the GUI to be cleared and reset

public String displayText(String)

- method that displays certain text in the GUI, returned from model during parsing or by View after specific user interaction

public displayError(Exception, String)

- method that displays the current error in the GUI, returned from model during parsing

public void updateDisplay()

- method that updates all objects in the display, called from the model after parsing and updating of model is complete

Internal API

Feature // Implementable interface that allows other programmers to create new types of features without modifying current features

    public void update()

Button // Extendable class that allows other programmers to create new buttons without modifying current code

    public void onClick()

TextDisplay // Extendable class that allows other programmers to create new regions for displaying text without modifying current code

    public void onClick()
    public void update()

Back-end API

External API

public void parseCommand(String)

- method that sends the command string to the back end parser for interpreting

public void onClick(double x, double y)

- method that sends an X and Y position within the GUI window to the back end for interpretation relating to specific commands, i.e. “on click” commands

public void onKeyPress(String)

- method that sends a String representing the key pressed on the keyboard by the user to the back end for interpretation relating to specific commands, i.e. “on key A” commands

public void onLanguageButton(int id)

- method to change language of commands, sends the id of a specific language to the back end to change how the parser works

Internal API

Actor // Extendable class that allows other programmers to create new types of actors without modifying current actors

    public void update()

KeyWord // Extendable class that allows other programmers to create new keywords without modifying current code

    public void update()

JUNIT Testing

Back-end

```
	@Test
	public void testPosition(){
		Parser parser=new Parser();
		Turtle t=new Turtle(x=0, y=0, a=90);
		parser.parse(“forward 50”);
		TurtleInfo info=new TurtleInfo(t);
		assertequals(“50”, info.getYPosition());
	}
	@Test(expected=ParseException.class)
	public void testParseException(){
	parser.parse(“asjklfhas;hgoask’dhfaga”);
	}
```

Front-end

```
	@Test
	public void testWindow(){
		myDisplay = new DisplayWindow(400, 400);
		assertequals(“400”, myDisplay.getXDim());
		assertequals(“400”, myDisplay.getYDim());
	}
	@Test
	public void testTurtleGraphic(){
		circle C = new circle(0,0,50);
		myTurtle = new TurtleGraphic(0, 0, C);
		assertequals(“0”, myTurtle.getXPos());
		assertequals(“0”, myTurtle.getYPos());
		assertequals(“50”, myTurtle.getmyShape().getRadius());
	}
	@Test
	public void testPen(){
		myPen = new pen(0,0, false);
		assertequals(“0”, myPen.getXPos());
		assertequals(“0”, myPen.getYPos());
		assertequals(“false”, myPen.isUp());
	}
```

Diagrams

User Interface

Module Relationships