Ellie Programming Lang.

Drawing Shapes Made Easier

Çelik Köseoğlu, Berk Evren Abbasoğlu, Fatih Sabri Aktepe - 28 November 2016

Report

Introduction

As our CS-315 group project titled “Design a Figure Drawing Language” we designed a programming language and named it Ellie. We used the requirements as our guide through the development process. In this report, it will be discussed how we addressed each part of the requirements, and the handling of the problems will be tried to be presented clearly.

Why Use Ellie?

Ellie offers simple mechanics to draw highly advanced and custom shapes. While maintaining its easy to use properties, it falls no short of supporting complex operations. The users can define custom shapes in addition to those already provided. With simple parameters like the location and size of the shape, it is possible to immediately draw them. These parameters are what Ellie calls “optionals,” meaning the user should use these parameters only if they want to draw customised shapes. Ellie is easy to use, highly functional, and is sure to address any demands regarding drawing shapes.

The name Ellie, comes from one of the best video games recorded in recent history: The Last of Us. Ellie is the name of one of the protagonists. You can find her picture on our cover page.

Ellie, The Mother of All Classes

In Ellie Programming Language, all Shapes are kind of an Ellie. When trying to create a new shape, the user instantiates an Ellie object, giving it a name, and then calls the constructor of the desired shape for this Ellie object. Following is an example of this.

```
Ellie line = Line();
Ellie oval = Oval();
Ellie rect = Rect();
```

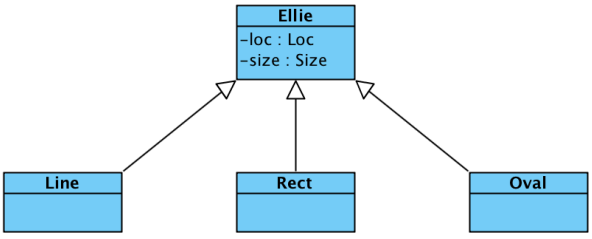

Figure 1: Ellie and the classes that inherit from it

The Bounding Box

The bounding box is an important term relevant for the functions of Ellie. The entirety of the drawing space is named the bounding box, which might be thought as a window with fixed height and width. The importance of it is that it will be possible to draw shapes relative to the parameters of the bounding box, rather than specifying a certain width and height for a shape. Perhaps, the concept is best explained with an example.

```
Ellie rect = Rect();
rect.drawEllie(filledState = False);
```

The above code creates an Ellie object called rect and gives it Rect() properties, where Rect is kind of an Ellie. The method called on the next line draws a rectangle matching the size of the bounding box, as no absolute width or height is specified. The syntax is expected to make more sense after reading the rest of the report.

Requirements Criteria

1) Support of Entry Point and Essential Variables

Like the main(String[] args) function commonly used in Java, Ellie makes use of a method named init() to initialise all the variables relevant for drawing shapes at the launch of the program. The contents of the init() method are always processed first.

As for our essential variables, we defined three. First is **width**, and integer value that stores the distance between the leftmost and rightmost places of the intended shape. Much like **width**, **height** stores the distance between the top and bottom spots of the shape as an integer. **stroke** (short for stroke width) is our third essential variable. Like the others, it is an integer, and it defines the thickness of the lines we will use to draw shapes. This will have a default value, so it is not needed to be given by the user during the use of the program.

A sum of our variables:

```
width : Int
height : Int
stroke : Int
```

2) Drawing Independent of Scale

The issue was discussed when defining the Bounding Box term. To further develop the concept, we will provide a different example using our code. Investigate the following:

```
Ellie oval = Oval()
oval.drawEllie(filledState: False, count: 10)
```

The provided code segment creates an Ellie object named oval and gives it Oval() properties, where Oval is a kind of Ellie (Oval inherits Ellie). The "count" parameter of drawEllie() method, an Int, specifies how many objects the user wants to fit in the bounding box. Considering the given example, the code would generate 10 ovals next to each other (horizontally), fitting into the bounding box.

3) Dynamically Typed Language

Ellie supports integers, doubles, strings, and booleans. Find a set of examples below.

```
Int examGrade = 100;
Double gpa = 4.0;
String success = "This report got full credit";
Bool passed = True;
```

Ellie supports basic arithmetic and logical operations. These include '+', '-', '*', '/', '||', '&&', '<', '>', '<=', '>=', '==', '!', '!='. Ellie also supports using '+' to concatenate two String objects. A simple set of examples are provided below.

```
examGrade = examGrade + 10; // examGrade becomes 110
whether (examGrade < 100) {} // read section 8 for more
String successTail = "from Buğra Gedik.";
String bottomLine = success + successTail;
//reads "This report got full credit from Buğra Gedik."
```

4) Three Essential Types: Loc, Size, Colour

There are three data types of utmost importance for Ellie to work as expected. These are defined as **Loc** (location), **Size**, and **Colour**. Like the data types discussed in Section 3, these will be readily available for the user to harvest.

Loc holds two integer values, x and y. These two parameters define the top left spot of the shape. If x and y are both assigned 0, then the shape will start to be drawn from the absolute top left of the bounding box. Any increments of x and y will respectively move the shape downwards and rightwards, a pixel for each addition. Negative values are not accepted. If the values are greater than what the bounding box allows, the start point of the shape will be considered the bottom, top right position of the bounding box. This command, inevitably, results with a single dot on the bottom and rightmost spot, as the rest of the shape does not fit inside.

Size, our second data type, holds two of our essential values (discussed in Section 1): **width** and **height**. This means it hold two integer values. Provided with the **Loc** and **Size**, there can only be drawn a single shape. This settles our parameters should the user want to specify absolute values for their shape rather than drawing relative to the size of the bounding box (see Section 2). Another thing worthy of note is, if only one of the values are provided, then the other is set equal to the given one. For example, if the user enters a value for **width** but leaves **height** empty, **height** takes the value of **width**. This is one of the examples for optionals in Ellie. If the user wants to draw a circle, the user could leave the width parameter empty to get a circle from an Oval class.

The final data type of note is **Colour**. It defines what colour the stroke will be when drawing the shape. Its constructor takes three integer values: red, green, and blue. These values are optional though, so if none are provided, the default values for all of them are 255. This results in a black stroke colour.

Colour also has a predefined set. Instead of entering specific red, green, and blue values, the user can enter simpler commands like Colour.Pink, Colour.Yellow etc.

To solidify the concept, find the relevant code examples below.

```
Loc shapeLoc = Loc(x: 0, y: 0);
Loc anotherLoc = Loc(x: 3, y: 5);

Size shapeSize = Size(height = 30, width = 20);
Size widthEqualsHeight = Size(width = 10); // height is also
10

Colour shapeColour = Colour(); // black
Colour pinky = Colour.Sienna; // sienna
Colour rainbow = Colour(r: 70, g: 90, b: 130);
Colour optionalRed = Colour(r: 185); //green and blue are 255
```

5) Drawing Shapes

Ellie has three built-in shapes, including Line, Rectangle, and Oval. They require no parameters to initialise, except Line**.** Line requires a single one, which is mandatory. It takes a String called **Direction**. The accepted forms of this parameter are limited to “S”, “N”, “E”, “W”, “SE”, “SW”, “NE”, “NW”. This determines where the line is facing, which is a crucial term should the line be considered like an arrow.

The drawEllie() method accepts three optional parameters for each shape: **Size**, **Loc**, and **Stroke**. If they are provided, the shapes will be drawn with desired size at the intended location. If they are not, their drawing will take into consideration our bounding box for either or both (depending on which are given). The same principles apply for **Stroke**.

As discussed in Ellie, The Mother of All Classes, all shapes are an Ellie. This allows to unite their instantiation with a common syntax. Take a look at the following code, where a Line object is created, and then drawn.

```
Loc lineLoc = Loc(x: 22, y: 90);
Ellie line = Line(direction: "SW");
line,drawEllie(loc: lineLoc);
```

Here, line does not take a **Size** parameter, which means it will start from the given location of (22,90) and travel in the southwest direction until it reaches the end of the bounding box. **filledState** is an unnecessary parameter for a line, since there is nothing to fill. **Colour** is an optional parameter. In this case, the line will be drawn black, its default value. If the user wants to manually pass the parameters, such action is also possible and welcome, as can be seen in the following example.

```
Loc lineLoc = Loc(x: 22, y: 90);
Colour lineColour = Colour(r: 70, g: 90, b: 155);
Size lineSize = Size(height: 100);
Ellie line = Line(direction: “SW”);
line.drawEllie(loc: lineLoc, colour: lineColour, size:
lineSize);
```

Now, let’s examine Oval and Rectangle. The initialisation of both and their drawing are presented with examples below. Note all the optional parameters are also passed. That does not mean they are necessary, like **fillColour** and **size**.

```
Loc ovalLoc = Loc(x: 5, y: 3);
Size ovalSize = Size(height: 10, width: 8);
Ellie oval = Oval();
oval.drawEllie(loc: ovalLoc, size: ovalSize, stroke: 5,
filledState: False, fillColour: Colour.Pink);


Colour rectColour = Colour(r: 70, g: 90, b: 155);
Loc rectLoc = Loc(x: 10, y: 25);
Size rectSize = Size(height: 30, width: 10);
Ellie rect = Rect(cornerRadius: 4);
rect.drawEllie(loc: rectLoc, size: rectSize, filledState:
True, fillColour: rectColour);
```

We discussed the instantiation of these shapes, which all follow a common syntax of the creating of Ellie objects, defining their optional and required parameters, and providing them to their drawEllie() methods. The absence of certain optional parameters (**size**) means the shapes will be drawn with relevant size to the bounding box. During the instantiation, the shape is scale free in this case. Line takes a String named **Direction** as a parameter in order to handle the case where arrows might play a role. In the given code example of the rectangle, it is seen that the user can also provide an optional parameter to specify their **cornerRadius**. This makes the corners rounded with respect to the Int value provided. The default value is 0, which means strict corners with 90 degree angles.

6) Support of Basic Drawing Statements

Examples of using draw methods were given in advance to this section, but we will now dive deeper into the subject, specifying the relevance, importance, and optionality of all variables necessary to draw. **Loc**, **Size**, and **Stroke** (stroke width) are discussed in detail in Sections 1 and 4. **fillState** and **fillColour** are two other parameters called for the drawEllie() method, which are discussed below. All are optional.

Ellie grants the user the chance to choose whether a shape will be filled when drawn. To address this, we require a boolean parameter named **fillState**, which is set to False by default. If the user specifically specifies this to be True while calling the drawEllie() method, the shape will be filled. The functionality bears the question though, with what colour? By default, this is set to black. However, we allow the user to enter yet another optional parameter called **fillColour**. The user can create custom colours by using the **Colour** data type, or using a preset of colours already defined in Ellie (see Section 4). With these optional parameters, Ellie aims to allow the users to draw with ease, while also providing more detailed options.

Even though examples of drawing statements were already provided throughout this report, the title demands another one here.

```
Colour rectColour = Colour.Pink;
Loc rectLoc = Loc(x: 10, y: 25);
Size rectSize = Size(height: 30, width: 10);
Ellie rect = Rect();
rect.drawEllie(loc: rectLoc, size: rectSize, filledState:
True, fillColour: rectColour);
```

In addition to shapes, we can also draw Strings. To accomplish that, we create an Ellie object and instantiate it like a String, and call the respective draw method. The following shows how to do this in code.

```
Ellie message = EllieString("Mr. Snowman");
message.drawEllie();
```

The above code draws "Mr. Snowman" onto the screen, with respect to the bounding box. It is also possible to pass the **Loc**, **Size**, **Stroke**, **filledState**, **fillColour**, parameters to the drawString method, but for simplicity, are not given in this case.

7) Loops

Ellie supports two forms of loops, namely the famous **while** and **for** statements. A simple example for each is below. Their functionality is identical to what you would expect from Java or Python, so we are not going to dive into much detail in this section. The only detail is the range specification shown in the for loop. To define a range, simply write **1...5** to iterate between [0,4].

```
Ellie rect = Rect();

for index in 1...5 { //equal to (int i = 0; i < 5; i++)
     rect.drawEllie();
}

Int x = 5;

while (x > 0) {
     rect.drawEllie();
     x = x - 1;
}
```

8) Conditional Statements

In Ellie, we have two conditional statements. They have identical functions to the famous if and else conditionals, but are renamed. If is replaced by the word **whether**, and else is replaced by **proviso**. An "if else" conditional can be represented by saying **whether proviso**. As said, there are no differences between them except the naming. An example of code is provided below, though it would be advised to check Section 3 again to remember our **examGrade** variable.

```
whether ( examGrade == 100 ) {
     // congratulate
}

proviso whether ( examGrade <= 99 ) {
     // punish
}

proviso {
     // congratulate even more
}
```

9) Modularity at the Level of Functions

Ellie has plenty of methods, like the init() method when the program first starts running, or the drawEllie() methods which are used for all the things related to drawing shapes. In addition to them, through the creation of custom classes, the user is allowed to define as many methods as they see fit (see Section 10). For these reasons, Ellie offers modularity at the level of functions. See documentation for more.

10) Support for the Extension of Shapes

While Ellie comes with default data types such as **Line**, **Oval**, and **Rect**, we are aware they will always fall short of satisfying the creative artist. For that, Ellie supports the extension of shapes to define new ones, all properties defined by the user! The only requirements that need to met for the creating of new shapes objects is, like all the others, the shape objects must inherit from Ellie (see Ellie, The Mother of All Classes). This means they will have the optional values of **Loc** and **Size**. They will also be instantiated as an Ellie object like the examples given throughout this report. To recall, below is another one.

```
Ellie rect = Rect();
```

Now, onto the definition of custom shapes by the user. First, instead of the keyword "extends" from Java, Ellie utilises "**->**". For example, the class name line for creating a custom shape might look as follows.

```
Snowman -> Ellie {
     // some things
}
```

Next, what are those "some things" we can do inside a defined class? First and foremost, there has to be a constructor for the class. Here is one possible constructor for the Snowman class.

```
Snowman(loc, size, message);
```

At first sight, the constructor begs the question, "What are the types of loc, size or message?" Common sense dictates loc should be of type **Loc**, size of type **Size** and message of type **EllieString**. However, there is no such definition within the class boundaries, so how do we know what to take? In fact, Ellie does not! Since this a dynamically typed language, we hand the responsibilities of managing custom classes to the user. This is explained in detail of the Documentation Section 4.C.

Now that we have constructors covered, let's define some properties. The example Snowman class has three properties. Two Ellie objects instantiated as ovals for the head and the body. Another Ellie holds the message field. Once the constructor is called, the interpreter creates a copy of each parameter and matches them with the relevant properties. This way, the class holds its own private variables. Again, read Documentation Section 4.C to learn more about the process.

```
Ellie snowmanBody = Oval();
Ellie snowmanHead = Oval();
Ellie snowmanMessage;
```

Our Snowman class is slowly fleshing out! Still, our class demands the addition of one crucial method: the custom definition of drawEllie() for this class. In the following code sample, we will override the drawEllie() method by defining it anew.

```
drawEllie(filledState = False, fillColour = Colour.Black,
          loc = Ellie.DefaultLoc, size = Ellie.DefaultSize) {
     //head over body ratio is 0.4
     Size headSize = Size(height: size.Height * 0.4)
     // draw the head
     drawOval(fs: filledState, fc: fillColour, l: loc, s:
     headSize);
     // draw the body
     drawOval(fs: filledState, fc: fillColour, l: loc, s:
     snowmanSize);
     // draw the message
     drawString(message);
}
```

For the drawing of the custom shapes, it follows the common syntax all others do. The newly created object is an Ellie, so the parameters will all be the same, including optional and mandatory ones for the draw() method. Here is an example of drawing a custom object.

```
Colour snowmanColour = Colour.White;
Loc snowmanLoc = Loc(x: 20, y: 32);
Size snowmanSize = Size(height: 50, width: 40);
String snowmanMessage  “Do you wanna build a      snowman?”;

Ellie snowman = Snowman(loc: snowmanLoc, size: snowmanSize,
               message: snowmanMessage);

snowman.drawEllie(filledState: True, fillColour:
snowmanColour);
```

As seen, the custom objects follow the same procedure when being initiated and drawn, with the same optional parameters and methods.

Conclusion

In conclusion, Ellie Programming Lang. incorporates a Dynamically Typed Language's advantages with a very focused syntax design. Since the language is centred solely on drawing shapes, we were able to create a highly specialised design pattern. With the Core Features provided, we believe this is a competitive candidate over complex programming languages.

With Ellie, we did our best to address all the requirements specified in the course page. Throughout the report, we tried to explain how we handled each part of the requirements and believe none were left unattended to, or inconclusive.

Combining simplicity, regularity, ease of use and high functionality, Ellie is the right language for any person willing to work with drawing shapes. Also with the possibility of creating custom shapes and using them like the predefined ones, Ellie is sure to administer the needs of all users.