MSO Lab Exercise III: Design and Refactoring

Chiel van Griethuysen (4014200) en Inès Duits(3930971)

January 17, 2014
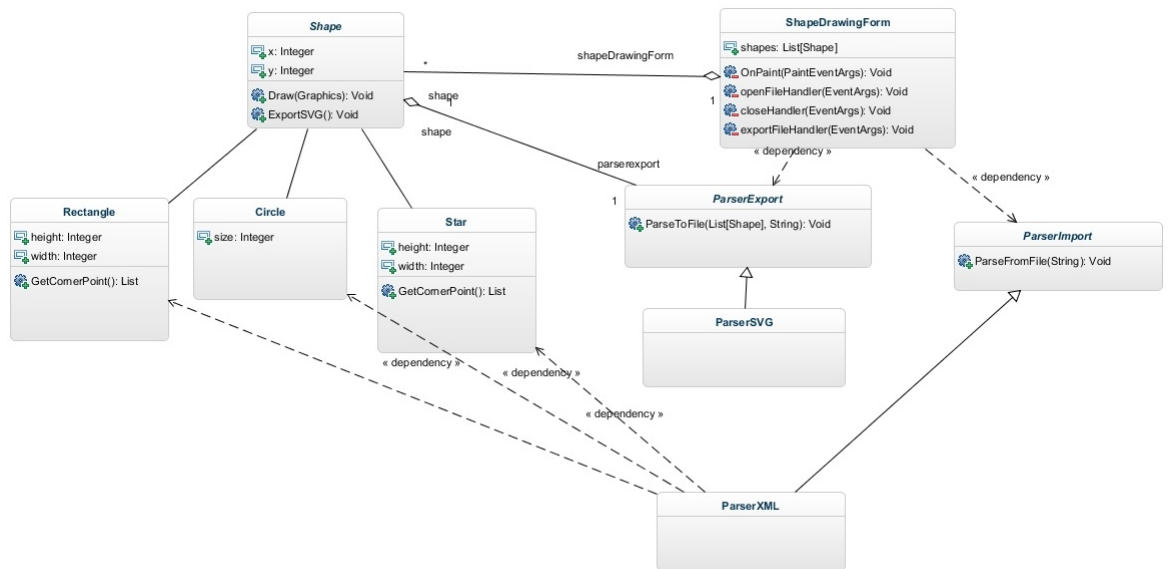
Contents

1 Design Comparison

1.1 Our original Design

We had a quite miss understanding of how to Our original Design looked like

You can see we kept the original Parser, but extend it to an abstract ParserImport, with only one subclass ParserXML. This so we can extend it for further use. We also decided to pulled some variables up in the abstract Shape class, the x and y coordinates which every shape should have.

To solve the main problem, which was adding a way to draw and to export SVG code without duplicating code, we add a new abstract class ParserExport. This class would get a subclass, ParserSVG, which will handle exporting SVG code. We make a mistake by not really understanding how to apply the bridge pattern, so from that point we choose to give every Shape a new method. With that method, exportSVG, we would create a SVG code such that every object could represent itself. But this code would be much the same as the Draw method of each Shape. Our idea was a try, but we quite misunderstand the Parser concept. We thought the actual drawing should be not the same as SVG code, because Exporting and Importing is something different then actually drawing on the screen. Which, after rethought our design, it is not. But we still tried to split the Parser (our ParserExport) from the Visualisation (our ParserImport) although the names may be not good chosen.

So our Original Design is not quit better then the "extend the existing Shapes with a new method to produce SVG", because that is actually what we did. We created a new method for each shape, instead of creating new definitions to draw Lines en Circles.

We were not planning on "copy-paste the existing visualization code into a new class and adapt it to produce SVG", but it could be that our solution used code which would look much like the existing graphic code, the original Draw method already had.

We try to not use any switch statement, to avoid "add switch statements to the existing Shape classes to switch between drawing on screen or generating SVG files". So our original Design, although not good, did not use switch statements in the new found solution.

1.2 Our Design

We will add our Design before starting to code because we make a mistake which need to be solved first. The diagram looks much alike.
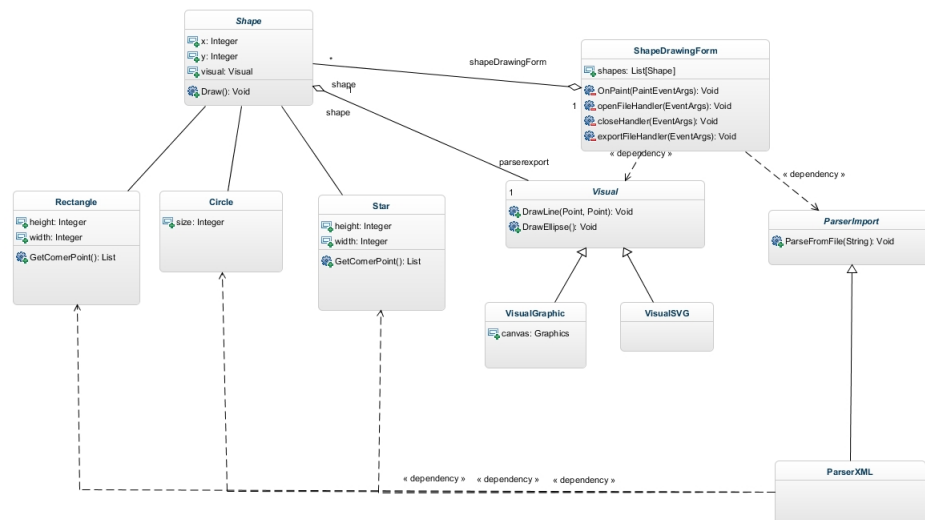
We still have the two abstract Parser methods, although we renamed the ParserExport to Visual, and his subclass, ParserSVG to VisualSVG. We add a new subclass to the renamed abstract Visual class, called VisualGraphics, which new task will be that what the Draw class of Shape did. The Visual class will get other methods, drawLine and drawCircle. After that we will recreate the Shape methods. We removed the ExportSVG method from the Shape class and we modify the Draw method. The Draw method

will be modified, it will now call the methods defined in Visual instead of the Graphics methods. The original code in Draw will be used to write the VisualGraphics class.

We will try to implement this 'new' Design but it may change during the refactoring. By applying the Bridge pattern the good way, we avoid using switches or copy pasting the code. So now the "extend the existing Shapes with a new method to produce SVG" will not occur. Actually by adding a new Visual class, and giving every Shape a visual, there is no need in the future to change the Shape class in the event of a new export file type. Because we only have to add a new subclass to the Visual class, which will describe how the abstract defined methods (drawLine and drawCircle) of the Visual class should be handled.

The new abstract Visual class also avoid "copy-pasting the existing visualization code into a new class and adapt it to produce SVG". By edeting the original Draw class to something which ask the Visual to draw instead, there is also no need to re-use the original Draw method in a new SVG draw class. We will give both the graphic and the SVG visual a way to implement the abstract methods of Visual and they both need a different output, so both a different code. But we do not have to tell both for SVG and for Graphics, that a Rectangle needs 4 lines. The Draw method of a Rectangle will tell the Visual to draw four lines, and the Draw method of the Star will tell the Visual to draw five lines. The Visual, independent from SVG or Graphic, will draw or four or five lines, not knowing that it is drawing a Rectangle or a Star.

By giving each Shape his own (abstract) Visual, each Shape does not have to know to which type it is drawing. The ShapeDrawingForm class will make sure that every Shape has the right concrete Visual subclass, without the Shape actually care about it. The Shape class still will call the Draw method, independent from the Visual it has. So we avoid "adding switch statements to the existing Shape classes to switch between drawing on screen or generating SVG files.". The Shape does not know there are different Visuals to switch between, the ShapeDrawingForm will change to the right Visual depending on which event will occur.

4

2 Implementation

2.1 Pull Up: Shapes

The pulling up from the variables x and y, into the abstract Shapes, didn't give trouble.

2.2 Add Attribute

We looked at the code and we found that the current ShapeDrawingForm class already had a List of Shapes. So we did not need to add this List.

2.3 Extract Variable: GetCornerPoints

In the original refactoring plan, the GetCornerPoints method should give back a List of Points. We changed this in a array of Points. The Points that will be return are in a specific order which we need to keep. We also add a lot of comments, to make clear how the calculation works for a maybe future change.

2.3.1 Rectangle

The subtraction from the four corners of a rectangle into a new function was not very hard.

2.3.2 Star

We already had a complicate code to calculate the Star Points. We used this code in the GetCornerPoints, which make the draw function itself look much less complicated.

2.4 Introducing Abstract Class: ParserImport

2.4.1 Rename Method and Push Down Method

2.4.2 Change Method in ShapeDrawingForm

2.5 Introducing Abstract Class: ParserExport -¿ Visual

Methods: DrawLine and DrawCircle

2.5.1 Add new subclass: VisualSVG

2.5.2 Add new subclass: VisualGraphic

Uses the original Draw method code

2.5.3 Change Method: Draw() in Shape

Rewrite the original Draw method in Shape, so that it use the VisualGraphic instead of the C# oriented Graphics.

2.5.4 Change Methods in ShapeDrawingForm

3 Reflection