Faculty of Science, Engineering and Technology

Object Oriented Programming

Pass Task 3.1: Drawing Program — Adding a Drawing

Overview

Drawing programs have a natural affinity with object oriented design and programming, with easy to see roles and functionality. In this task you will start to create an object oriented drawing program.

Purpose:	Learn to apply object oriented programming techniques related to collaboration and use of framework classes.
Task:	Extend the shape drawing program to have a Drawing that has many shapes.
Time:	Aim to complete this task by the start of week 4
Resources:	

Submission Details

You must submit the following files to Doubtfire:

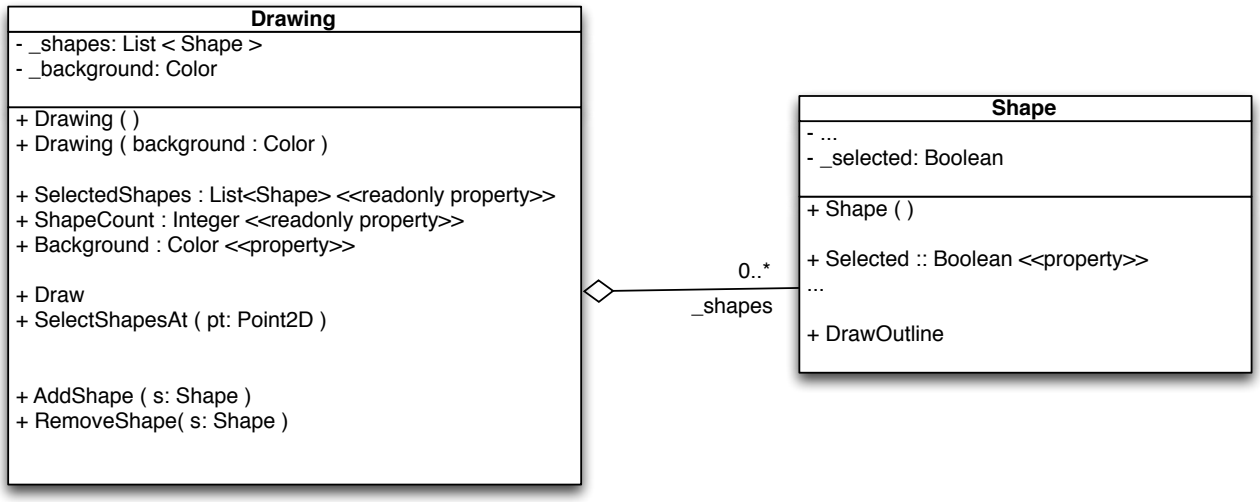
- Program source code
- Screenshot of program execution

Instructions

This task continues the Shape Drawer from the previous topic. In the previous topic you created a **Shape** class that can draw itself to the screen. However, the program requires the ability to draw many shapes to the screen.

In order to achieve this we need new code to manage a collection of Shapes - something we will call a **Drawing**. If you look back at the description of the program, the Drawing objects contain a number of shapes which can be drawn to the screen as a *single* drawing. The Drawing therefore contains code that can be used to manage and draw these shapes.

The **Drawing** class will perform the role of managing and drawing a collection of shapes. In terms of the program we want to be able to *add shapes*, *select shapes*, *remove shapes*, as well as *drawing* them to the screen. All of these aspects can be managed by a Drawing object.



> **Note**: You can probably think of many other operations for the drawing, such as saving to file etc. This will be a useful part of the overall program.

The line between the Drawing and Shape, with the hollow diamond, indicates that the Drawing is made up of Shapes. This is called **aggregation**, where the Drawing is seen as the *whole* which has parts that are *shapes*. This is a permanent relationship and means that each Drawing object will always know its related Shape objects.

At the other end of the line, the 0..* means that the Drawing potentially knows many Shape objects, but that it may also know none. This is further annotated with **_shapes** to indicate that this is stored in the _shapes field.

> **Note**: The _shapes field is actually a List of Shape objects, but we don't show this in the relationship. The collection class is implied by the 0..* aspect of the relationship.

1. Open your **Shape Drawing** solution.
2. Add a new **Drawing** class. This class will need to use a List object to do the storage, so add the code to use the System.Collections.Generic namespace.

```
using System.Collections.Generic;
```

Note: The full name of a class (struct, or enumeration) is actually a combination of its namespace name and its name. So the **Shape** class in the **MyGame** namespace is actually called **MyGame.Shape**. Adding a **using** statement at the top provides a list of namespaces for the the compiler to search when it looks for a class you use. In this case when you use **List**, it will find it and use **System.Collections.Generic.List**, the class' full name.

3. Add a **private, read only**, field to store the list of **_shapes**. Use List<Shape> as the type.

Tip: Mark fields as **readonly** if you are not going to change them after the object is created. In this case we will always be using the same List object, so we do not want to change the field.

Note: A readonly field cannot be changed, meaning that you cannot assign a new value to the field. However, the object that field refers to can change, and will change in this case as we add and remove shapes from the list.

```
public class Drawing
{
    private readonly List<Shape> _shapes;
    …
    public Drawing(Color background)
    {
        _shapes = new List<Shape>();
        …
    }
}
```

4. Add a **_background** private *field* and public **Background** property for the background color.
5. Create the **constructor** that takes in the **background** color as a parameter.
   5.1. Create a new List<Shape> object and store it in **_shapes** field.
   5.2. Initialise the **background** to the supplied background color.

Object oriented programming languages come with **class libraries**. These libraries include a number of classes that you can use to create useful objects for your program. Objects that manage **collections** of objects are very useful, and so all class libraries will come with some classes you can use to create these **Collection Objects**.

A Collection Object will contain the smarts needed to maintain a *number* of objects for you. For example, the **List** class in .NET provides the intelligence to manage a **dynamic array** of objects. You can add, remove, and fetch objects from the list… it has everything you need if you want a dynamic array of some kind.

In .NET the collections have **generic** versions that allow you to specify the **type** of object (or value) you will store within the collection. For example, a List < Shape > is a List of references to Shape objects. This means you can add, remove, and get Shape objects from collections of this type. Whereas, a List < int > is a List of int values. You can add, remove and get int values from collections of this type.

Note: .NET has both **value** and **reference** types. A *value type* stores its value in the variable associated with it. So the variable i, in the case of int i, will store an integer value. Where as classes are *reference types*, meaning that the value in the variable is actually a **pointer** to the object which resides on the **heap**.

When you ask a class for a **new** object, it allocates space on the heap and returns the pointer to this location. The pointer is then stored in the variable, so it is not the object's *value* but a *reference* to the object.

6. Create an additional **constructor** with no parameters, it will also need to initialise the background color and _shapes List (creating a new List for it to refer to). However, you want to avoid duplicating this code, as code duplication is a bad idea. Instead, have this constructor call the constructor with one parameter. The following code shows how.

```
public class Drawing
{
    public Drawing ( Color background ) { … }
    public Drawing ( ) : this ( Color.White )
    {
        // other steps could go here…
    }
}
```

Note: Special syntax is needed in C# to call the other constructor, as this is not a normal method that you can call yourself. The keyword **this** is used to refer to the current object, so public Drawing ( ) : this (Color.White) tells the compiler to call the constructor for this object and pass it a Color value before running the rest of the current constructor.

Note: The constructor without parameters is called the **default constructor**.

7. Add a **Shape Count** property to the Drawing that it is read only, and it returns the Count from the _shapes collection object (this is the number of shapes within the collection).

```
public int ShapeCount
{
    get { return _shapes.Count; }
}
```

> **Tip**: Objects should be lazy! Don't remember things or do things that others can do more easily for you. The List is remembering all of the Shapes, so we can get the count from it. Anyone asks us for the count, we ask our list and return what it tells us.

8. Create the **Add Shape** method in the Drawing class so that it adds the shape it receives to its list of shapes.

> **Hint**: You can Add shapes to a List of Shapes (List < Shape >).

9. Now switch to the **Shape** class and add the **_selected** field and property.

> **Tip**: Refactoring tools can implement the property for you. Right click the _selected field and select **Refactor**, then **Create Property**.

10. Now add a **Draw** method to the **Drawing** class. This will tell SwinGame to **clear** the **screen** to the **background** color, and then loop over each shape and tell it to draw itself.

> **Note**: See how the Drawing class does not actually draw the shapes, it asks the shapes to draw themselves. This is the idea of collaboration in object oriented programming.

Now to update the program's main instructions.

11. Switch back to the **Main** method in **GameMain**.
12. Remove the Shape variable and related code.
13. Create a **Drawing** object outside of the game loop. This will be the drawing object the user is interacting with.

14. Inside the event loop:

    14.1. Check if the user has **clicked** the **left mouse button**, and if they have add a **new Shape** to your Drawing object based on the mouse's location.

> **Hint**: Use the **default constructor** and then alter the X and Y location of the shape using the mouse's location.

    14.2. Change the background color to a new random color when the user presses the space bar.

Now we need to be able to remove shapes from the drawing. To do this we need to add the ability to select objects by clicking on them in there user interface.

15. Add a **SelectShapesAt** method to the Drawing class. This will be called when the user wants to select shapes they have already added to the Drawing. Use the following rough pseudocode to implement this.

```
Method: SelectShapesAt
------------------------------
Parameters:
 - pt: the Point2D of the shapes to select
Local variables:
 - s: a reference to a single Shape
------------------------------
 1: For each Shape s in _shapes
 2:      If s is at pt then
 3:           Tell s, its selected is true
 4:      Else
 5:           Tell s, its selected is false
```

> **Hint**: There may be an easier way to do this without using an **if statement** in the loop. If the expression "s is at pt" is true, what do we assign to s' selected, and when it is false?

16. Use the following pseudocode to implement the **Selected Shapes** getter.

```
Property: Selected Shapes
-----------------------------
    Getter:
-----------------------------
    Local Variables:
    - result - a List < Shape > to return
-----------------------------
     1: Assign result a new List < Shape > object
     2: For each Shape s in _shapes
     3:    If s is selected then
     4:        Tell result to add s
     5: Return result
```

To help the user see which shapes are selected we need to add a visual cue by outlining the selected shapes. Drawing methods cannot be unit tested, so you will have to run and check this yourself.

17. Add an **Draw Outline** method to the Shape class. This will **draw** a black **rectangle** around a selected shape. The outline should be -2 pixels to the left and above the shape, and 4 pixels wider and higher than the shape so that it surrounds it on all sides.
18. Also change the Shape's **Draw** method add some code to call **Draw Outline** if the Shape is selected.
19. Now update the main instructions. In the event loop, tell the drawing to **Select Shapes At** the mouse's position when the **right mouse button** is **clicked**.
20. Compile and run the program and check that you can select shapes… press the delete key and feel the need to make this do something!

Getting delete to work should be relatively straight forward, as the List will do most of the work. We just need to get the message to it, telling it which shape(s) to remove.

21. Adjust the code so that all of the selected shapes are removed from the drawing if the user types the DeleteKey or BackspaceKey.

Once your program is working correctly you can prepare it for your portfolio.

Add a screenshot of the program working, and your source code.

> **Note**: Make sure to distribute the functionality across the program and Drawing. The program should not interact with the Drawing's _shapes list to achieve this.