School of Science, Computing and Engineering Technologies

Object Oriented Programming

Week 5: Drawing Program and SwinAdventure Iteration 4

Overview

In this week, there are **two assessable tasks** 5.1 and 5.2. **Each task contributes 2%** to your final grade. Noting that you need to complete these tasks before coming to your allocated lab. In the lab, there will be verification tasks and short interview to verify your understanding.

In task 5.1, you extend the ShapeDrawer application that you created in in Task 4.1 to draw multiple shapes. In task 5.2, we will optimize the Iterations 2 and 3 that we created in previous two weeks.

Purposes Task 5.1 pages 5-7)	Learn to apply object-oriented programming techniques related to collaboration and the use of framework classes. Extend the shape drawing program to allow for many shapes to be drawn on the screen. **The task contains personalized requirements.**
Task 5.2 pages 8-9	Learn to apply Inheritance principle and object collabration in OOP and conduct the NUnit test. Understand the case study program and implement iteration 4. **The task contains personalized requirements**

Note: If you are unable to complete theses tasks, you will struggle in this unit. Your tutor may be able assist you with suggestions on how increase your knowledge. You can also participate in HelpDesk sessions

Task 5.1 Instructions - ShapeDrawer - Multiple shapes

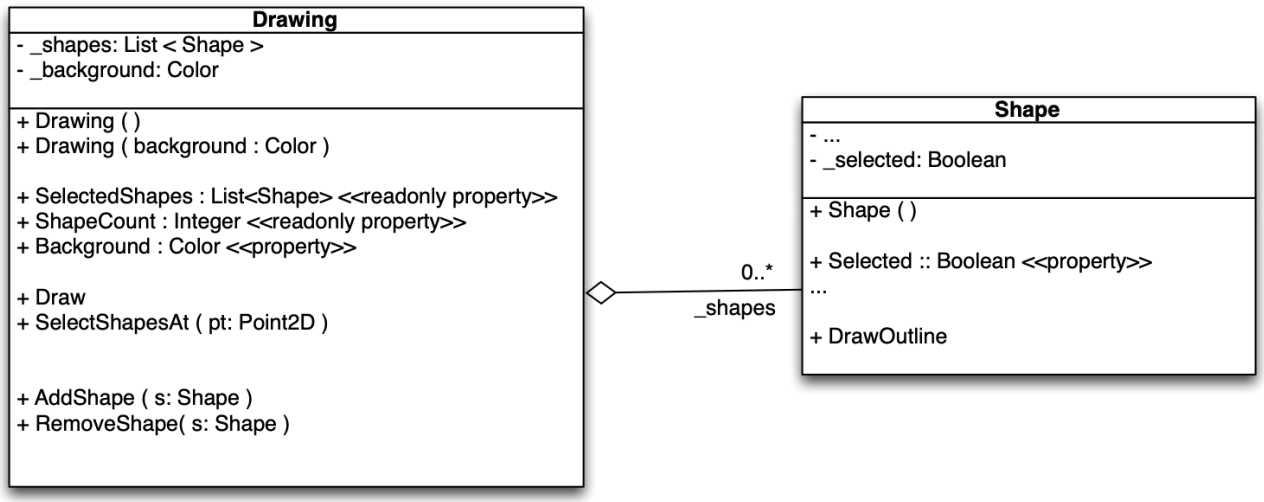
1. Review the ***SplashKit Installation Instruction*** document in Week 0. It outlines what you need to create a sample project and can display a graphic window on your screen.

2. Continue with the ShapeDrawer application that you developed in task 4.1 Drawing Program – A Basic Shape. You may want to archive your previous work before extending the application.

3. Presently, your program only allows you to draw one shape (i.e., a rectangle at the current mouse pointer position). However, a general drawing program should be able to draw many shapes to the screen. To achieve this, extend your program with a new class, called Drawing. Class Drawing defines a container of Shape objects and provides mechanisms to add shapes to the collection, to select shapes for drawing, to remove shapes from the collection, and to draw the selected shapes to the screen. In addition, class Drawing supports a set of properties to query information about the collection or to alter drawing attributes.

> **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 following UML diagram illustrates the new system design.



In the class *Shape*, we introduce one new property *Selected*, and a method called *Draw utline* (see pages 6-7 for more detail specification). The line between class *Drawing* and class *Shape*, with the hollow diamond, indicates an *aggre-gation* between class *Drawing* and class *Shape*. It states that objects of class *Drawing* have a "***_shapes***" collection of zero or more objects of class *Shape* as attribute.

> **Note**: The ***_shapes*** field is actually a list of *Shape* objects, which can have zero of more elements.

1. Open your existing **ShapeDrawer** solution with Visual Studio Code.
2. Add a new class, called *Drawing*, to your application.
3. Add a private, read-only field ***_shapes***. Use *List*<*Shape*> as the type for ***_shapes***.

> **Note**: Class *List* is a member of the *System.Collections.Generic* namespace. This namespace is implicitly available to every C# project. Hence, you do not need to add a ***using*** declaration for this namespace at the top or your C#-class file.

> **Tip**: Mark fields as ***readonly*** if you are not going to change them after the object is created. In case of ***_shapes***, 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 private ***_background*** field and public ***Background*** property for the background color. Use type *Color* as type. You need to add ***using*** *SplashKitSDK*; at the top of your *Drawing.cs* file to make type *Color* available.
5. Create the constructor that takes in the ***background*** color as a parameter.
    - Create a new *List*<*Shape*> object and assign it to the ***_shapes*** field.
    - Initialize the ***_background*** to the supplied background color.

Object-oriented programming languages come with a rich set of class libraries. These libraries include a set *collection classes*, like *List*, that manage *collections of objects*. Collection classes define the smarts needed to maintain a collections of objects for you. For example, the *List* class provides the intelligence to manage a *dynamic array* of objects. You can add, remove, and fetch objects from the list, etc. A *List* has everything you need if you want a dynamic array of some kind.

In C# and the .NET framework, collection classes are *generic datatypes*. Generic datatypes provide a uniform set of operations for the objects contained in the collection across all possible instantiations. For example, the specification *List*<*Shape*> instantiates the generic type *List* with class *Shape.* The result is a list of *Shape* objects. Generic collections "*remember*" which type of objects they store. So, if you can add a *Shape* object to a *List*<*Shape*> collection, then you get *Shape* objects back if you ask the *List*<Shape> collection to return an element.

6. Add a second constructor with no parameters to class **Drawing**. A constructor with no parameters is called *default constructor*. Default constructors initializes objects with predefined values.

   When defining the default constructor, you want to avoid code duplication. For this reason, you use a ***this****-call* to another defined constructor to initialize the object. In case of class Drawing, use the constructor that takes a background color for the ***this***-call, using *Color.White* as argument.

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

Note: When you use a ***this****-call*, all member variables should be initialized in the target constructor. For example, calling ***new*** *Drawing*() must return an object that is equal to an object returned by ***new*** *Drawing*(*Color.White*).

7. Add a read-only ***ShapeCount*** property to class *Drawing* that returns the ***Count*** from the ***_shapes*** list collection object.

> **Tip**: Follow the *single responsibility principle* (SRP) in your object-oriented design. For example, which actor in your system is responsible for computing the number of objects in ***_shapes***? You do not reimplement this feature in class *Drawing*. The responsibility rests with class *List*. So, you ask List to compute its Count and return the result as value for read-only property ***ShapeCount***.

8. Create the ***AddShape*** method in class *Drawing* that adds the shape it receives to its list of shapes.
9. Create the ***RemoveShape*** method in class *Drawing* that removes the shape it receives from its list of shapes.

> **Note**: List's ***Remove*** method returns ***true*** if it successfully removed an item. You may want to use a discard assignment "_ = …" to ignore the return value.

10. Switch to your *Shape* class and add a private ***_selected*** field and a public ***Selected*** property. (You do not need to change the constructor. A Boolean field is initialized to ***false*** by default.)
11. Return to class *Drawing* and add a ***Draw*** method. Tell *SplashKit* to ***ClearScreen*** using the ***_background*** color as argument and then loop over each shape and tell it to ***Draw*** itself.

> **Note**: 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.

12. Go to the ***Main*** function in *Program*.cs.
13. Remove the ***myShape*** variable and all code referring to ***myShape***.
14. Create a *Drawing* object ***myDrawing*** using the default constructor outside the ***do-while*** loop.

15. Inside the ***do-while*** loop in ***Main*** (after *SplashKit.**ClearScreen***):

> **Note**: The ***do-while*** loop in ***Main*** is also called *event loop*.

15.1. Check if the user has clicked the left mouse button. In this case, add a **new** *Shape* to ***myDrawing*** using the current mouse pointer location.

> **Hint**: You should use a local variable to create a *Shape* object using the default constructor, then alter the ***X*** and ***Y*** location of the shape using the mouse pointer location, and finally add the newly created shape to ***myDrawing***.

15.2. Change the background color of ***myDrawing*** to a new random color when the user presses the space bar.

15.3. Tell ***myDrawing*** to ***Draw*** before *SplashKit.**RefreshScreen***.

16. Compile and run your program. Add shapes and change the background color.

17. Switch to class *Drawing*. Add the ***SelectShapesAt*** method to your *Drawing* class. Use the following pseudocode as a guide.

```
    SelectShapesAt(pt)
1:  // parameter pt is a 2D point
2:  foreach s in _shapes
3:      if Tell s to IsAt(pt)
4:          s.Selected := true
5:      else
6:          s.Selected := false
```

> **Hint**: Avoid the ***if***-statement inside the ***foreach***-loop. The result of ***IsAt*** is a Boolean.

18. Use the following pseudocode to implement the read-only property ***SelectedShapes***.

```
    SelectedShapes:get()
1:  Let result by a new List of Shape objects.
2:  foreach s in _shapes
3:      if s.Selected
4:          Tell result to Add(s)
5:  return result
```

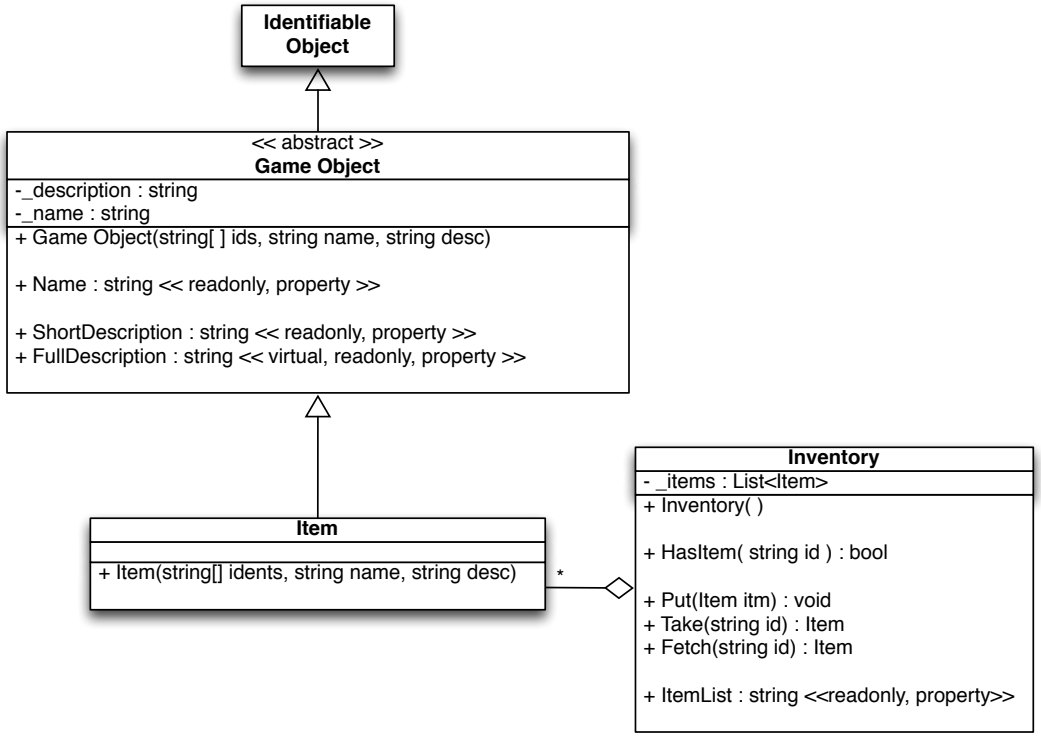
19. Switch back to class *Shape*. Add a ***DrawOutline*** method to the *Shape* class. This method draws a black rectangle around the shape. **The lack rectan le has to e 5 pi els wider on all sides o the shape, where presents the last di it o your student I .**
20. Change the Shape's ***Draw*** method add some code to call ***DrawOutline*** if the shape is selected.
21. In function ***Main***, inside the event loop, add an ***if***-statement to check whether the user has clicked the right mouse button. If this is the case, tell ***myDrawing*** to **SelectShapesAt** the current mouse pointer position.
22. Compile and run the program and check that you can select shapes
23. Adjust the code so that all of the selected shapes are removed from the drawing if the user types the *KeyCode.DeleteKey* or *KeyCode.BackspaceKey*. Make sure your work is still consistent with the provided UML diagram — you do not need to add anything to class *Drawing* to make this work!

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

Task 5.2. Instructions - SwinAdventure Iteration 4 - Inheritance

1. Review the ***Case Study Requirements*** document. It outlines what you need to create.

2. In the previous Iterations 2 and 3, you may have noticed that there is a duplicated code between the classes IdentifierObject and Item. In addition, there is no a baseline abstract class yet so that all classes in our SwinAdventure game can extend from it. If there is no such class, we have to manually modify the code for each class if there is a new game requirement or update.
3. With above observation, the goal is to optimize the existing code using Inheritance principle and to promote the relationship among classes.
4. We provide you with the newly updated UML class diagram as follows.



This iteration adds a new abstract GameObject class which helps to create a number of key classes for the game. The **Game Object** class will be used to represent anything that the SwinAdventure player can interact with. It is an abstract so that the game cannot create a real-world object directly from this class.

You can utilize your existing code implementation in Iterations 2 and 3 to complete this task.

- Game Object - "anything" the player can interact with
  - Name - this is a short textual description of the game object
  - Description - a longer textual description of the game object
  - Short Description - returns a short description made up of the name and the first id of the game object. This is used when referring to an object, rather than directly examining the object.
  - Long Description - By default this is just the description, but it will be changed by child classes to include related items.

A number of GameObjects will need to contain Items. This functionality is encapsulated in the **Inventory** class. This provides a managed list of items.

- Inventory - a managed collection of items
  - Items can be added using Put, or removed by id using Take
  - Fetch locates an item by id (using AreYou) and returns it

Item Unit Tests	
Test Item is Identifiable	The item responds correctly to "Are You" requests based on the identifiers it is created with.
Test Short Description	The game object's short description returns the string "a name (first id)" eg: a bronze sword (sword)
Test Full Description	Returns the item's description.
Test Privilege Escalarion	The item returns correctly the first ID as your tutorial ID if the inputed pin matches the last 4 digits of your student ID.

Inventory Unit Tests	
Test Find Item	The Inventory has items that are put in it.
Test No Item Find	The Inventory does not have items it does not contain.
Test Fetch Item	Returns items it has, and the item remains in the inventory.
Test Take Item	Returns the item, and the item is no longer in the inventory.
Test Item List	Returns a string containing multiple lines. Each line contains a tab-indented short description of an item in the Inventory.

- Re-test your existing unit tests for the classes Item and Inventory.

When you arrive at your lab, you will receive the verification tasks. See you very soon.