Faculty of Science, Engineering and Technology

# Object Oriented Programming

## Topic 3: Object Collaboration

### Resources

The following resources can help you with this topic:

- C# Station Tutorials
  - Generic Collections
  - Indexers
- Tutorials Point C# Programming Tutorials

### Topic Tasks

Before starting to work on the tasks, first read through this entire document to get a sense of the direction in which you are heading. Complete the following tasks and submit your work to Doubtfire for feedback before the deadline (check Doubtfire for the submission deadline).

| | |
|---|---|
| Pass Task 9 - Shape Drawer | 2 |
| Pass Task 10 - The Spell Book | 14 |

Remember to submit your progress, even if you haven't finished everything.

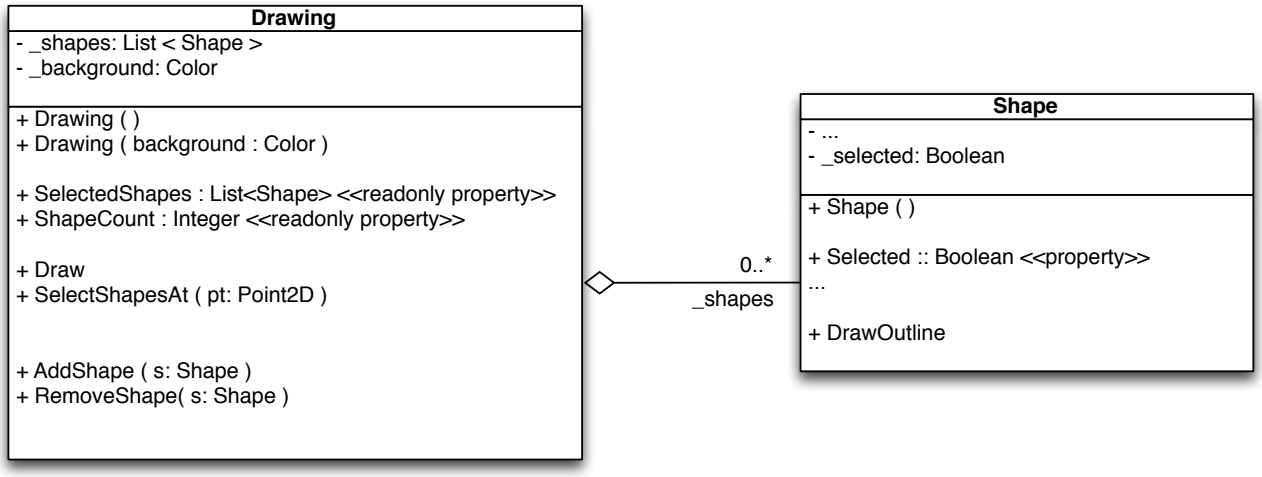
After you have **discussed** your work with your tutor and **corrected any issues**, it will be signed off as complete.

# Pass Task 9 - Shape Drawer

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 the Drawing knows about Shape objects.

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

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 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 the **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. Create a new **Drawing Unit Test** file, and add the following tests:

   7.1. **Test Default Initialisation** - test that you can create a Drawing and that it's color defaults to the Color.White.

   7.2. **Test Initialise with Color** - test that when you creates a Drawing with a color passed to its constructor, that becomes the color of the background.

**Note**: Remember to add **using** clauses to give you access to the Color class and SwinGame code.

8. Run the unit tests and make sure that they pass.

Next we will add the code to add shapes in to a drawing. We can do this using **Test Driven Development**, which means that we write the test first and then build the code to make the test succeed. This helps ensure we come up with valid, useful, tests.

9. Create a new **Test Add Shape** unit test in the Drawing Unit Test class. Use the following code, it creates a Drawing and then adds two shapes to it. Testing the ShapeCount at the start and after adding.

```
[ Test()]
public TestAddShape ( )
{
   Drawing myDrawing = new Drawing();
   int count = myDrawing.ShapeCount;

   Assert.AreEqual( 0, count, "Drawing should start with no
shapes" );

   myDrawing.AddShape( new Shape() );
   myDrawing.AddShape( new Shape() );
   count = myDrawing.ShapeCount;

   Assert.AreEqual( 2, count, "Adding two shapes should in-
crease the count to 2" );
}
```

**Note**: This code represents the way we want to work with the Drawing. Working out how you want to use an object can help you work out how it should work. It will not compile as it is, but that is ok. Continue with the tutorial to see how to easily add these features.

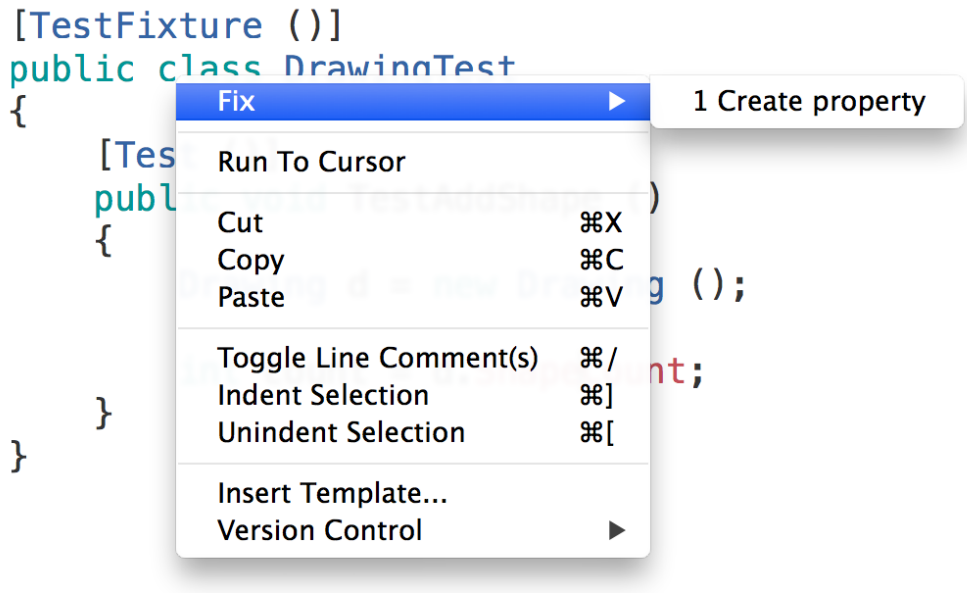
10. At this point the IDE will indicate that there are a couple of errors as the Drawing does not have an **Add Shape** method nor does it have a **Shape Count** property. The great news is that the IDE can start to create these for you.

> **Note**: The refactoring tools in the IDE make it easy for you to create stubs of the methods and properties you need to implement your programs.

11. Right click on the call to the **Shape Count** property, select **Fix** and **1 Create property**.

> **Note**: If the options menu shown below does not appear when you right-click, then you can left-click and press Alt+Enter to access this menu.

This will go ahead and create a stub property for you with the details IDE can work out - typically guesses at parameter types, and return types.




12. Use the cursor to position the property below the fields and constructors, but above your methods.
13. Use the **Fix** tool to create the method for Add Shape as well. Position it below the properties.
14. Now review the code that the Fix tool has created.
15. Change the **Shape Count** property so that it is read only, and it returns the Count from the _shape object.

```
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.

16. Change the **Add Shape** method 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 > ).

17. Re-run the tests and make sure they pass.
18. 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**.

19. Create a new test in **Shape Test** that checks that the object correctly remembers its **selected** value, and that changing the property changes this value.
20. Run the unit tests and ensure they success.

21. 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.

Now to update the program's main instructions.

22. Switch back to the **Main** method in **GameMain**.
23. Remove the Shape variable and related code.
24. Create a **Drawing** object outside of the loop. This will be the drawing object the user is interacting with.
25. Inside the event loop:
    25.1. 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.

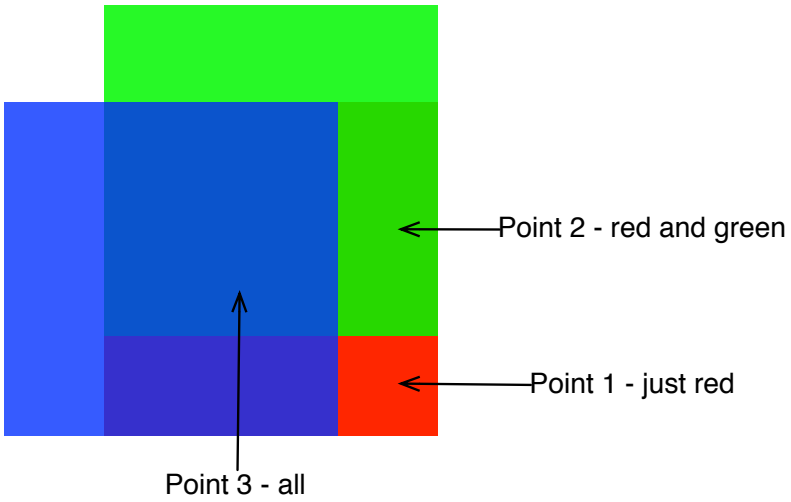
    25.2. Change the background color to a new random color when the user presses the space bar.

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.

26. Start with the test, so create a **Select Shape Test** in the Drawing Tests. This will work as follows:

```
public void TestSelectShape ( )
{
   Drawing myDrawing = new Drawing();
   Shape[] testShapes = {
             new Shape(Color.Red, 25, 25, 50, 50),
             new Shape(Color.Green, 25, 10, 50, 50),
             new Shape(Color.Blue, 10, 25, 50, 50) };

   foreach(Shape s in testShapes) myDrawing.AddShape( s );

   List<Shape> selected;
   Point2D point;

   point = SwinGame.PointAt( 70, 70 );
   myDrawing.SelectShapesAt( point );
   selected = myDrawing.SelectedShapes;
   CollectionAssert.Contains( selected, testShapes[0] );
   Assert.AreEqual( 1, selected.Count );

   point = SwinGame.PointAt( 70, 50 );
   myDrawing.SelectShapesAt( point );
   selected = myDrawing.SelectedShapes;
   CollectionAssert.Contains( selected, testShapes[0] );
   CollectionAssert.Contains( selected, testShapes[1] );
   Assert.AreEqual( 2, selected.Count );
   …
```

The test creates 3 rectangles, and currently checks that the correct shapes are selected when the SelectShapesAt method is called.



At this point there is a missing *constructor* in the *Shape* class, and the missing *SelectShapesAt* method in the *Drawing* class.

27. Get the code to compile first - add stubs for the following:

    27.1. A new **constructor** to the **Shape** class, it should take in a color and then parameters for x, y, width and height.

> **Note**: Make sure that the x and y parameters use the float type.

    27.2. A new **SelectShapesAt** method in the **Drawing** class. This will take a **Point2D** value you can name *pt*.

    27.3. A new read-only **Selected Shapes** property in the **Drawing** class. This will return a List < Shape >.

28. Add a new test to Shape, that tests your new constructor. It should create a Shape object using the constructor and then validate that the Shape has the right color, is in the right location, and has the right size.

29. Run the tests and watch them fail.

> **Tip**: It is important to see that the tests can fail when the code isn't there. You then know your changes have worked

> **Tip**: Grab a **screenshot** for your portfolio — you can use it as evidence of use of the IDE and test driven development approach!

30. Switch to the Shape's constructor and implement the constructor.

> **Hint**: Also change the default constructor to call this new constructor to avoid code duplication. That would be following recommended good practice.

31. Run your tests and ensure that the Shape's constructor now works.

32. Next get Drawing's **SelectShapesAt** method to work. 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?

33. 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
```

34. Re-run your tests and watch them succeed.

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.

35. 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.
36. Also change the Shape's **Draw** method add some code to call **Draw Outline** if the Shape is selected.

Now update the main instructions.

37. In the event loop:
    37.1. Tell the drawing to **Select Shapes At** the mouse's position when the **right mouse button** is **clicked**.
38. 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 will 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 to remove.

39. Switch to your **Drawing Tests** code.
40. Add a new **Remove Shape Test**. Have it create a Drawing, and add some Shapes to it. Check the Shape Count, and then tell the Drawing to Remove one of the shapes. Double check the Shape Count. Also check that the shape is no longer there by using Select Shapes At and checking that the shape is no longer in the selected shapes.

> **Hint**: Keep a local variable to refers to the shape you want to remove.

41. Use the refactoring tools to add a stub for **Remove Shape**. It should accept a single **Shape** parameter, and will remove this from the list when we implement it (after seeing the test fail).
42. Run the unit tests and check that it fails.
43. Implement **Remove Shape**.

> **Hint**: Have a look at the methods in **List**, you should be able to call a single method and have the List remove the Shape for you!

44. Run the unit tests and check that it fails.

45. Update the program's main instructions so that pressing delete will delete all of the selected shapes from the drawing.

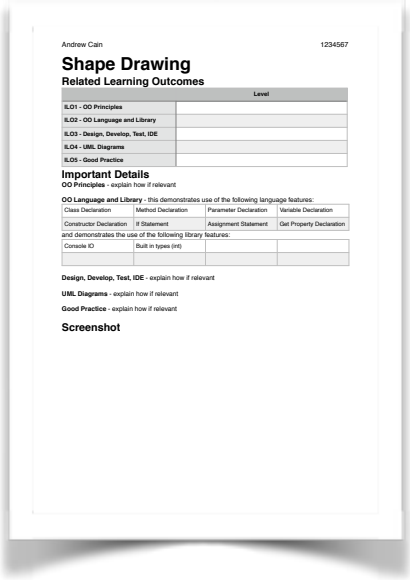
> **Note**: Check if the user has typed the vk_DELETE **or** the vk_BACKSPACE keys.

46. Run the program and test that you can now add, select, and delete shapes.

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

Do not create a new cover page, instead add to your existing Shape Drawing program cover page. Add a few sentences for the learning outcomes you feel that you have demonstrated some aspect of in this work. Include screenshots of the program in action.

> **Tip**: Think about what you are demonstrating with this. Showing understanding and thought at this step will make it easier to summarise when you get to the end of the unit. Reflect on any aspects you found interesting or challenging in this process.



Andrew Cain 1234567

**Shape Drawing**

**Related Learning Outcomes**

| | Level |
|---|---|
| **ILO1 - OO Principles** | |
| **ILO2 - OO Language and Library** | |
| **ILO3 - Design, Develop, Test, IDE** | |
| **ILO4 - UML Diagrams** | |
| **ILO5 - Good Practice** | |

**Important Details**

**OO Principles** - explain how if relevant

**OO Language and Library** - this demonstrates use of the following language features:

| | | | |
|---|---|---|---|
| Class Declaration | Method Declaration | Parameter Declaration | Variable Declaration |
| Constructor Declaration | If Statement | Assignment Statement | Get Property Declaration |

and demonstrates the use of the following library features:

| | | | |
|---|---|---|---|
| Console IO | Built in types (int) | | |
| | | | |

**Design, Develop, Test, IDE** - explain how if relevant

**UML Diagrams** - explain how if relevant

**Good Practice** - explain how if relevant

**Screenshot**

### *Pass Task 9 - Assessment Criteria*

Make sure that your task has the following in your submission:

- The program can add, select, remove, and draw Shapes as indicated.
- Code must follow the C# coding convention used in the unit (layout, and use of case).
- The code must compile and the screenshot show it outputting the correct details.
- Must also include screenshots of the unit tests passing (and possibly others).

# Pass Task 10 - The Spell Book

Return to your School of Magic program. This week you will add a **Spell Book** to this program.

1. Use the following description to **draw** your own **UML class diagram** of the Spell Book and its relationship with your Spell class.

A Spell Book contains zero of more spells, and spells can be added and removed by the Wizard at any time. The wizard can then access the spells by fetching the spell at a given index in the book.

**Hint**: The ability to access a Spell at a given index can be implemented as an **indexer** in C#. In the UML show this as a property named this[int i], i.e.:

```
this[int i] :: Shape <<readonly property>>
```

**Tip**: Draw this on a whiteboard, take a photo, we will love it. This is a great way to think through your designs quickly, and record the results. Dont spend ages with a drawing program, we want the content not a pretty picture!

2. Create a Spell Book Tests file and add unit tests for adding, removing, and fetching Spells from a Spell Book.
3. Implement the new Spell Book class, and add the methods, properties, and fields you indicated in your UML. The following code demonstrates how to implement an indexer in C#.

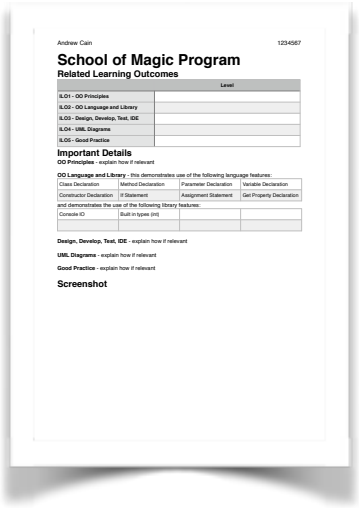
```
public class SpellBook
{
    private List<Spell> _spells;
    public Spell this[int i]
    {
        get
        {
            return _spells [i];
        }
    }
}
SpellBook myBook; …
Spell s = myBook[0];
```

4. Run your tests and make sure they pass.

5. Add XML documentation to your Spell Book class.

> **Note**: An **indexer** is a special kind of property that allows the caller to access your object using an index like an array. So by adding an indexer to your Spell Book callers can ask for **myBook[0]**. This will call the getter, passing in 0 as the value i.
>
> You can also have setters, so **myBook[0] = new Spell(…);** can also work but does require a **set** inside the indexer.

Once your tests are working correctly you can prepare this piece for your portfolio. Do not create a new cover page, instead add to the existing School of Magic page.


Andrew Cain 1234567

**School of Magic Program**

**Related Learning Outcomes**

| | Level |
|---|---|
| ILO1 - OO Principles | |
| ILO2 - OO Language and Library | |
| ILO3 - Design, Develop, Test, IDE | |
| ILO4 - UML Diagrams | |
| ILO5 - Good Practice | |

**Important Details**

**OO Principles** - explain how if relevant

**OO Language and Library** - this demonstrates use of the following language features:

| | | | |
|---|---|---|---|
| Class Declaration | Method Declaration | Parameter Declaration | Variable Declaration |
| Constructor Declaration | If Statement | Assignment Statement | Get Property Declaration |

and demonstrates the use of the following library features:

| | | | |
|---|---|---|---|
| Console IO | Built in types (int) | | |
| | | | |

**Design, Develop, Test, IDE** - explain how if relevant

**UML Diagrams** - explain how if relevant

**Good Practice** - explain how if relevant

**Screenshot**

### *Pass Task 10 - Assessment Criteria*

Make sure that your task has the following in your submission:

- UML class diagram shows the Spell Book and Spell classes and their relationships. A photo of a rough sketch will be great, as long as it is readable.
- The unit tests correctly check the features of the Spell Book.
- The Spell Book has XML documentation for all public features.
- Code must follow the C# coding convention used in the unit (layout, and use of case).
- The code must compile and the screenshot show the tests passing.