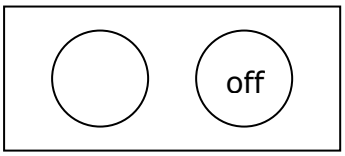
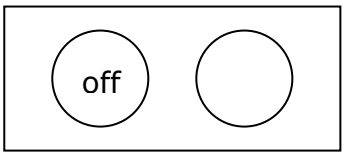
Implementing Aggregation

What is Aggregation?

The parent-child example described different ways of implementing an association, i.e. where one object 'uses' the services of another. Aggregation is a stronger form of association in that one object is made up of other objects. For example, train movement through slow-moving (<50km/hr) sections of a railway line may be controlled by two-aspect signal lights suspended over the track to inform the train driver whether to stop (red) or go (green); i.e.

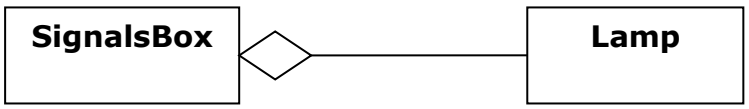


Stop instruction



Go instruction

In this example a single set of signal-lights is a box containing two lamps, one red and one green (here we will refer to it as a 'signals box'). The box is one object, but itself contains two lamp objects. Each lamp is a part of the signals box. A diamond symbol is used to express this aggregation;



The key phrase indicating aggregation is 'is a part of'. However there is a discussion in the programming community as to whether aggregation can be distinguished from a strong (direct) association that has a 'one-to-many' relationship. Basically, if the designer prefers, aggregation as a concept (and diamond symbol) *need not be used at all* and all such relationships could be legitimately expressed as an association. This does not concern us other than to be aware of the debate; even the Unified Modelling Language (UML) is deliberately imprecise in defining aggregation.

The Lamp class

An example of how one might implement (or interpret) in coding a particularly strong form of direct association (i.e. aggregation) with respect to modelling a signals box will be given later. First a

comprehensive explanation of a Lamp class is described. When used in Form this class can be used to display a single coloured circle (a lamp) to screen. The code for the class is given below.

```
class Lamp
    {
        private Color m_lampColour;         // Lamp colour
        private bool m_onStatus;            // true = ON, false = OFF
        private Rectangle m_pos;            // Position

        public Lamp(Color col, int atx, int aty)
        {
            m_onStatus = false;             // Initially off
            m_lampColour = col;             // Sets lamp's colour
            m_pos = new Rectangle(atx, aty, 50, 50);
        }

        public bool lampOn { get { return m_onStatus; }
                             set { m_onStatus = value; } }

        public void display(Graphics g)
        {
            SolidBrush myBrush;

            if (m_onStatus)
                myBrush = new SolidBrush(m_lampColour);
            else
                myBrush = new SolidBrush(Color.Gray);

            g.FillEllipse(myBrush, m_pos);
        }
    }
```

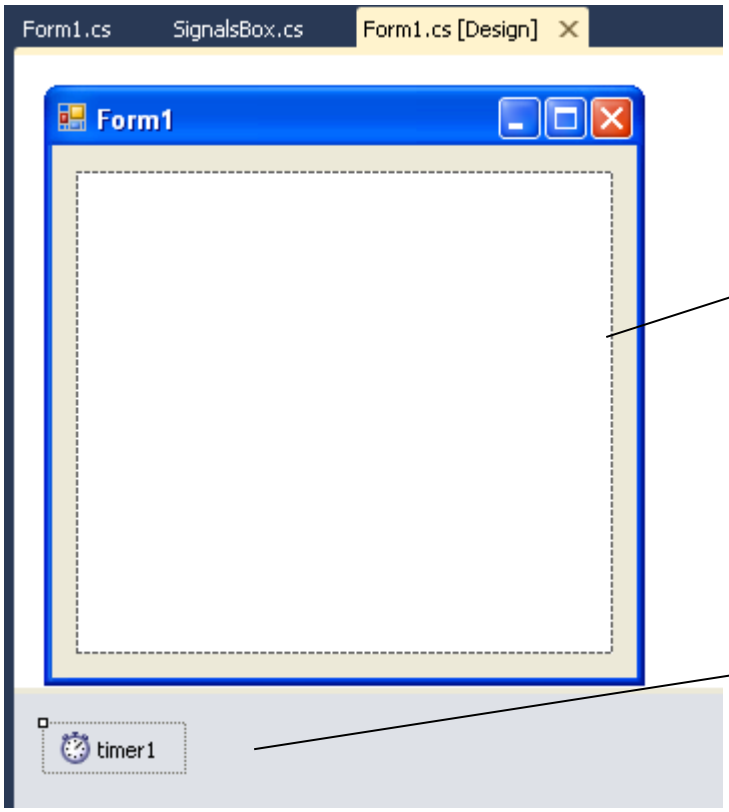
The Lamp class contains three attributes. The **m_lampColor** variable is of class **Color** which can be used to set the background or foreground colour of various graphical components, and is part of the System.Drawing namespace. Colours are represented by symbolic constants such as **Color.Gray** in method **display()**. Attributes **onStatus** and **m_x**, **m_y** determine whether the lamp is turned on or not and its position in the current drawing window (typically a PictureBox).

The Lamp class has a constructor, one read write property and one method. The constructor initialises the three attributes on instantiation of an object of the class. The property and method are self-explanatory; note in principle that the lamp can be turned on or off and displayed to screen. For example if the m_lampColour was red but turned off, then the **display()** call would actually display a grey circle. The lamp has to be turned on in order for the colour equating to m_lampColor to display.

The fact that the lamp can be turned on and off has an important consequence since in order to 'see' the effects of such an action there has to be a way of pausing program execution. For example, if a program was written which instantiated an object of class Lamp, set the colour to red,

then turned it off and on again each time displaying the lamp, the user would not see the blink because program execution would be too fast. There needs to be a time delay between changes of lamp state in order for it to be visible to the observer. More than one strategy could be adopted, but the one chosen here is to use a Timer attached to the form and toggle the Lamp on and off in its tick event handler.

Applications can now be written to instantiate an object of class Lamp and can use a Timer class so that changes of state are visible during runtime. The following program draws a 'red' lamp and immediately turns it on. After 1 second the program turns it off, then waits another 1 second before turning it back on again.



A simple pictureBox add to a windows form, with background

A timer object added to the frame with:

Enabled = true;

Interval = 1000

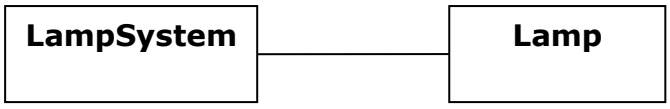
```
public partial class Form1 : Form
{
  private Lamp m_redLamp;
  public Form1()
  {
        InitializeComponent();

        m_redLamp = new Lamp(Color.Red, 10, 10);
        m_redLamp.lampOn = true;

        // Connect the Paint event of the PictureBox to the event handler
        // method.
        pictureBox1.Paint += new
              System.Windows.Forms.PaintEventHandler(this.pictureBox1_Paint);
  }
```

```
  private void pictureBox1_Paint(object sender,
                                 System.Windows.Forms.PaintEventArgs pe)
  {
       // Create a local version of the graphics object for the PictureBox.
       Graphics g = pe.Graphics;
       m_redLamp.display(g);
  }

  private void timer1_Tick(object sender, EventArgs e)
  {
       // Toggle lamp on and off
       m_redLamp.lampOn = !m_redLamp.lampOn;
       // Force picture box to be redrawn
       pictureBox1.Invalidate();
  }
}
```

Notes:

In order to draw our Lamp we need to add a method to respond to the paint event of the picture box, this is performed in the Form constructor. In order to force the repainting of the picture box and thus the calling of our event handler function we invalidate the picture box in the timer tick event handler.

A simple class diagram of this program would be;



The SignalsBox class

Now we are in position to demonstrate aggregation through modelling a two-aspect (i.e. two lamp) railway signal box. A new class to define the signals box is needed, the code listed below;

```
    public class SignalsBox
    {
        private Lamp m_redLamp;
        private Lamp m_greenLamp;

        public SignalsBox(int x, int y)
        {
            // Create lamps
            m_redLamp = new Lamp(Color.Red, x, y);
            m_greenLamp = new Lamp(Color.LightGreen, x, y+50);
        }

        public int changeLights(int flag)
        {
```

```
            switch (flag)
            {
                case 0:
                    m_redLamp.lampOn = true;
                    m_greenLamp.lampOn = false;
                    flag = 1;
                    break;

                case 1:
                    m_redLamp.lampOn = false;
                    m_greenLamp.lampOn = true;
                    flag = 0;
                    break;
            }
            return flag;
        }

        public void displayLights(Graphics g)
        {
            m_redLamp.display(g);
            m_greenLamp.display(g);
        }
    }
```

The **SignalsBox** class has two attributes, one for each lamp. When the **SignalsBox()** constructor is called, note that *the lamps themselves are actually created by the constructor*. This is arguably the strongest form of association (i.e. aggregation) and in terms of implementation is different from a direct association of types 3 or 4 (see material on associations) in that in those cases the constituent objects are *first* created and *then* passed to the container object's set method or constructor.

The SignalsBox has two defined states, either RED, OFF or OFF, GREEN. In the LampSystem example the single lamp was switched on and off from within **timer1_Tick(),** but in this aggregated example individual control of each light is implemented in a method of SignalsBox called **changeLights()**. Each of the two conditions (ie, RED, OFF or OFF, GREEN) are represented by an integer **flag** variable taking the value of 1 or 0 (a Boolean variable could be used, but would need to be changed if the system was ever to subsequently implement more than two conditions). If the case flag variable is 0, this means the red lamp must then be turned on and the green lamp turned off, and the flag condition changed to 1. A flag variable of 1 would result in the turning off the red lamp and turning on the green lamp. Thus multiple calls to the changeLights() method will alternate the signals box condition and so alternate colour visibility of the red and green lamps when method **displayLights()** is called.

A second form, identical to the first, can now be created to demonstrate the conditions of signal box, displaying a stop – go – stop cycle.

```
public partial class Form2 : Form
{
```

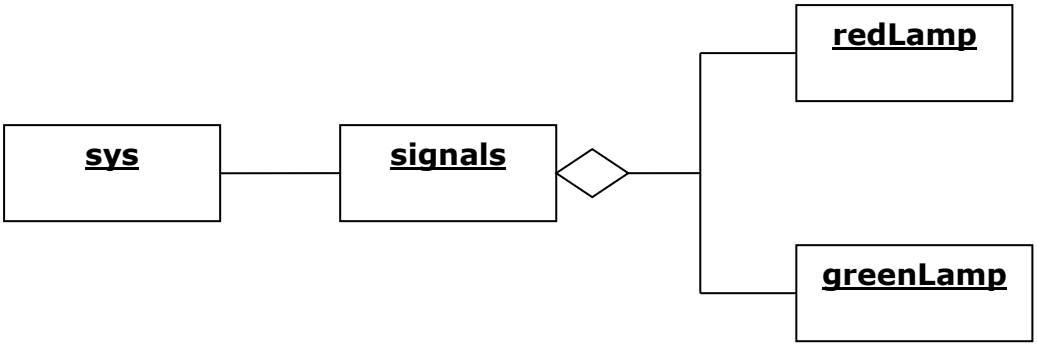
```
  private SignalsBox m_sigBox;
  private int m_lightsState;
  public Form2()
  {
       InitializeComponent();

       m_sigBox = new SignalsBox(10, 10);
       m_lightsState = m_sigBox.changeLights(0);

       // Connect the Paint event of the PictureBox to the event handler.
       pictureBox1.Paint += new
              System.Windows.Forms.PaintEventHandler(this.pictureBox1_Paint);
  }

  private void pictureBox1_Paint(object sender,
                                 System.Windows.Forms.PaintEventArgs pe)
  {
       // Create a local version of the graphics object for the PictureBox.
       Graphics g = pe.Graphics;
       m_sigBox.displayLights(g);
  }

  private void timer1_Tick(object sender, EventArgs e)
  {
       m_lightsState = m_sigBox.changeLights(m_lightsState);
       pictureBox1.Invalidate();
  }
}
```

Notes:

The **SignalBox** object (which itself will create the two lamps) is instantiated from within the forms constructor. Integer variable **m_lightsState** is then initialised to 0 and this variable determines the condition the signal box will display, tested by repeated calls to **changeLights()** which in turn modifies the condition and returns the state variable for the next iteration around the loop, after a second delay.

An updated class diagram of the program might be;

Programs can also be designed through use of Object diagrams, and in this case helps appreciate the aggregation relationship;



Exercises

1. A Traffic light contains 3 lamps; red, yellow (for amber) and green. By utilising the Lamp class write a class called TrafficLight that contains three lamps and can instigate the following 4 conditions (hint: modify the SignalsBox class);

RED	RED	OFF	OFF
OFF	YELLOW	OFF	YELLOW
OFF	OFF	GREEN	OFF

2. Create a Form that draws the traffic light and cycles through the full sequence of light changes from red through to red in the normal UK fashion, with 4 second delays between the light changes.

3. Using the TrafficLight class, set up a pair of lights to simulate the traffic flow over a narrow bridge.