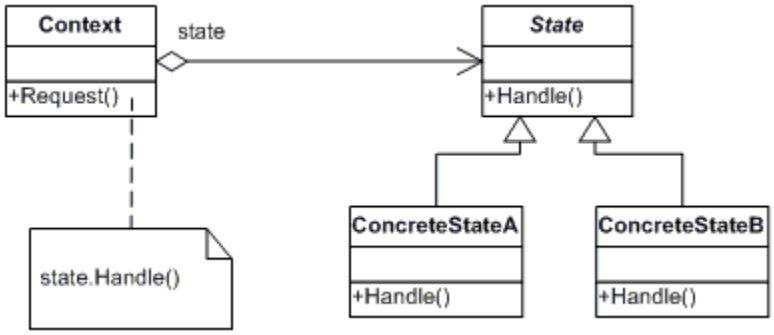
Jack Raney

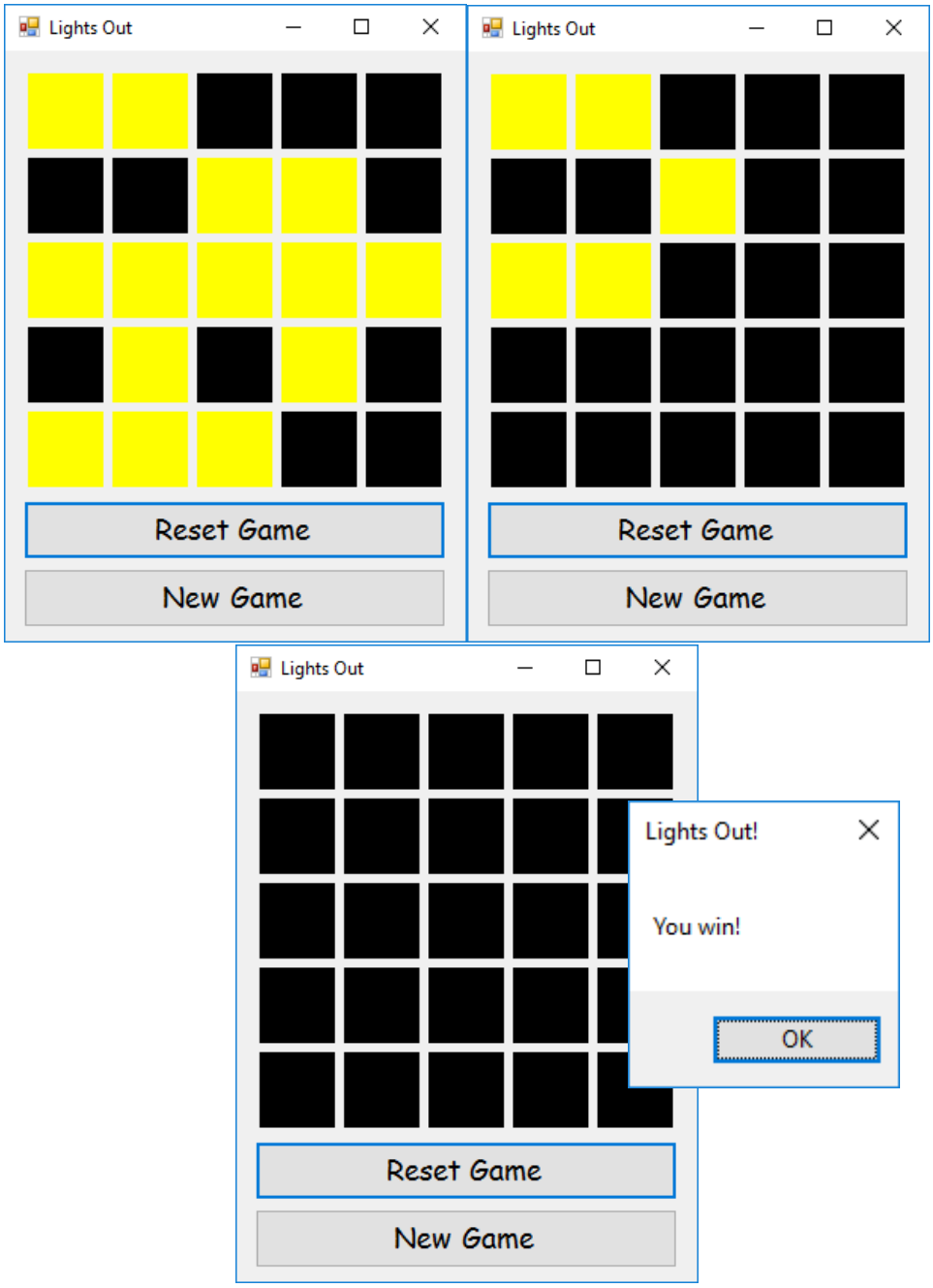
Design Patterns

The State Pattern

This paper was written to describe the State Pattern, a design pattern that's intent, according to dofactory, is to "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class". This means that a State object contains a method or methods that vary depending on the whichever State subclass it is and changes through polymorphism. The UML diagram for the State Pattern is shown below.



I have created the program LightsOut (based on the electronic game Lights Out) to exemplify this pattern. If you don't know, the objective of the game is to turn off all the lights. When a light is pressed, it and the lights directly adjacent to it (no diagonals) switch between being on or off.



The Code

Light.cs

```
public abstract class Light //The State class
{
    protected bool start;

    public bool Start
    {
        get { return start; }
        set { start = value; }
    }

    public abstract Light switchLight();    //The Handle() method
    }
}
```

Light is the parent class to LightOn and LightOff, the two different states the light will be in.

The start Boolean determines whether or not that light started on or off in the puzzle.

LightOn.cs

```
public class LightOn : Light    //A ConcreteState class
{
    public LightOn(bool s)
    {
        start = s;
    }

    public override Light switchLight() //The Handle() method
    {
        return new LightOff(start);
    }
}
```

LightOn and LightOff both set the start property in their constructors. Both switchLight() methods return the opposite state, which inherits (not in the technical way, mind you) the state of the start Boolean.

LightOff.cs

```
public class LightOff : Light   //A ConcreteState class
{
    public LightOff(bool s)
    {
        start = s;
    }

    public override Light switchLight() //The Handle() method
    {
        return new LightOn(start);
    }
}
```

Form1.cs

```
public partial class Form1 : Form   //The Context class
{
    Random rand = new Random();

    private Light[,] grid = new Light[5, 5];

    public Form1()
    {
        InitializeComponent();
        newGame();

        foreach (Panel p in gridPanel.Controls.OfType<Panel>())
        {
            p.Click += clickLight;
        }
    }
```

In the form, each light is represented by a panel. All 25 of them are stored within a larger panel and listen for a click event shown later in the code.

```
private int panelX(Panel p)
{
    string name = p.Name;
    char[] nameArray = name.ToCharArray();
    return (int) nameArray[5] - 48;
}

private int panelY(Panel p)
{
    string name = p.Name;
    char[] nameArray = name.ToCharArray();
    return (int) nameArray[6] - 48;
}

private Panel getPanel(int x, int y)
{
    string name = "panel" + x + y;
    foreach (Panel p in gridPanel.Controls.OfType<Panel>())
    {
        if (p.Name.Equals(name))
            return p;
    }
    return panel00;
}
```

Within my code I represented the locations of the lights with x and y coordinates starting at (0,0) that are represented within the names of the panels (panel00, panel10, panel20, etc). panelX() and panelY() return those coordinates based off the name, and getPanel() returns the Panel based off of coordinates entered.

```
private void changeColor(Panel p)
{
    if (p.BackColor.Equals(Color.Black))
    {
        p.BackColor = Color.Yellow;
        return;
    }
    p.BackColor = Color.Black;
}
```

changeColor() flips the color of an input Panel between black and yellow whenever the light is turns off or on.

```
private void update(Panel p)    //The Request() Method
{
    int x = panelX(p);
    int y = panelY(p);
    grid[x, y] = grid[x, y].switchLight();
    changeColor(p);

    foreach (Panel pan in gridPanel.Controls.OfType<Panel>())
    {
        int row = panelX(pan);
        int col = panelY(pan);

        if ((Math.Abs(row - x) + Math.Abs(col - y)) == 1)
        {
            grid[row, col] = grid[row, col].switchLight();
            changeColor(pan);
        }
    }
}
```

update() switches a selected light panel on or off and changes its color. The same is done to all adjacent light panels.

```
private void newGame()
{
    bool empty = true;

    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            grid[i, j] = new LightOff(false);
        }
    }

    foreach (Panel p in gridPanel.Controls.OfType<Panel>())
    {
        p.BackColor = Color.Black;
    }

    foreach (Panel p in gridPanel.Controls.OfType<Panel>())
    {
        if (rand.Next() % 5 == 0)
        {
            update(p);
        }
    }

    foreach (Panel p in gridPanel.Controls.OfType<Panel>())
    {
        if (p.BackColor.Equals(Color.Yellow))
        {
            grid[panelX(p), panelY(p)].Start = true;
            empty = false;
        }
    }

    if (empty)
        newGame();
}
```

newGame() creates a new puzzle by doing the following:

1. Turning all of the lights off
2. "Pressing" a random set of buttons (each has a 1 in 5 chance to be pressed)
3. Coloring the turned-on lights yellow and setting the start Booleans appropriately.
4. Checking to make sure there are lights on. If there are not, the process is done again

```
private void checkWin()
{
    bool empty = true;

    foreach (Panel p in gridPanel.Controls.OfType<Panel>())
    {
        if (p.BackColor.Equals(Color.Yellow))
        {
            empty = false;
        }
    }

    if (empty)
    {
        MessageBox.Show(this, "You win!", "Lights Out!", MessageBoxButtons.OK);
    }
}
```

When a light is clicked, one of the things it does is check if all the lights are off. If so, the player wins.

```
private void clickLight(object sender, EventArgs e)
{
    update((Panel) sender);
    checkWin();
}
```

clickLight() is the event method called by each panel. It simply calls update() and checkWin().

```
    private void resetButton_Click(object sender, EventArgs e)
    {
        for (int i = 0; i < 5; i++)
        {
            for (int j = 0; j < 5; j++)
            {
                if (grid[i, j].Start)
                {
                    grid[i, j] = new LightOn(true);
                    getPanel(i, j).BackColor = Color.Yellow;
                }
                else
                {
                    grid[i, j] = new LightOff(false);
                    getPanel(i, j).BackColor = Color.Black;
                }
            }
        }
    }

    private void newGameButton_Click(object sender, EventArgs e)
    {
        newGame();
    }
}
```

When the reset button is pressed, it returns the lights to their beginning layout based off of the start Boolean.

The new game button simply calls newGame()

Reflection

The State Pattern itself is very easy to understand, so I decided to challenge myself with the program by recreating a game I used to own as a child. While a bit time-consuming (creating this program involved looking up many of the properties and methods built into C#), I am very pleased with the results. I had a previous idea to this one that would have been even more difficult: creating a simulator for Conway's Game of Life. I will probably do that as a pet project one of these days.

In my opinion, the State Pattern is incredibly useful, particularly from a game development standpoint.