Jackson Spencer

The Facade Pattern

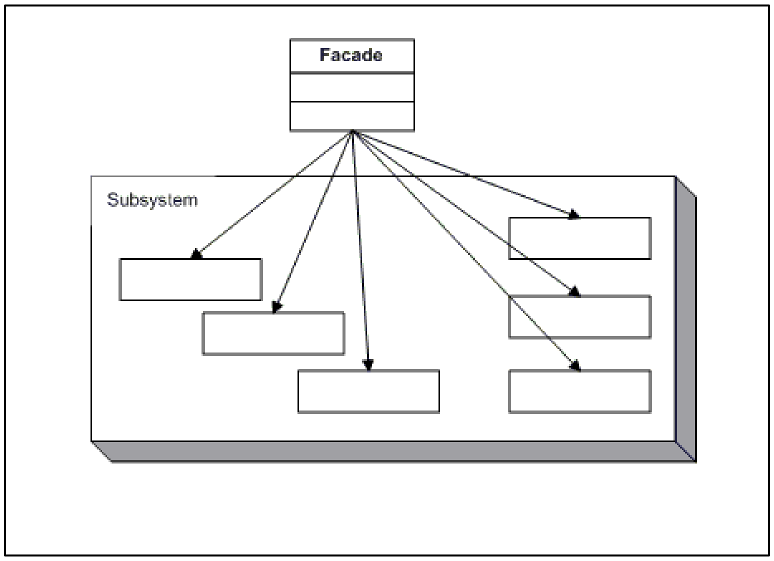
September 14, 2016

ECCS 2411: Design Patterns

Introduction

The assignment required an application to demonstrate the Facade design pattern. We were to implement the pattern using a Facade class, as an umbrella of sorts, and multiple subsystem classes. This can be described more in the UML diagram section. The data I decided to use for this program was to build an entertainment center remote to control a TV, Blu-Ray player, record turntable, CD player, and sound system.

UML Diagram



This diagram shows the general components of the Facade Pattern. The façade class is responsible for knowing what subsystem is responsible for each request. The subsystem classes implement the functionality of the system as a whole, basically doing all the work. The subsystems also have no knowledge of the façade class and keep no references to it (notice no arrows pointing back to the Façade class from the subsystems).

Classes

Each component of the remote has its own class, in order for each to have its own properties. Each component of the remote had two properties. Either it was active, or it was not. The first option to activate a device on the form was to “Watch Cable TV.” There are three methods: tvActive() returns the value of the boolean “isOn” which is determined by the methods tvON() and tvOFF().

```
public class TV   // A subsystem class to control TV
    {
        private bool isOn = false;

        public bool tvActive()      // Determines TV Active or not
        {
            return isOn;
        }

        public void tvON()  // Turn On Method
        {
            isOn = true;
        }

        public void tvOFF()     // Turn Off Method
        {
            isOn = false;
        }
    }
```

The other component classes are very similar to this one. Each have the properties of being “Active” or “Off.”

The façade class took the form of a class called *Univ_Remote*. It takes each class that represents the entertainment center components and makes requests using those classes. In each method, only the components needed for that activity are made active. For example, the method for the Record Player ensures that only the record player and the sound system are active. There is also a method that deactivates every device.

```
public class Univ_Remote        // The Facade Class
    {
        public TV TV = new TV();
        public XBR Xbox = new XBR();
        public Record Record = new Record();
        public CD CD = new CD();
        public SoundSys Sound = new SoundSys();

        public void cableTV()   // Activates the TV only
        {
            TV.tvON();
            Xbox.brOFF();
            Record.recOFF();
            CD.cdOFF();
            Sound.sysOff();
        }

        public void bluRay()    // Activates the TV and the Xbox
        {
            TV.tvON();
            Xbox.brON();
            Record.recOFF();
            CD.cdOFF();
            Sound.sysOff();
        }

        public void recPlay()      // Activates the Turntable and the sound system
        {
            TV.tvOFF();
            Xbox.brOFF();
            Record.recON();
            CD.cdOFF();
            Sound.sysOn();
        }

        public void cdPlay()    // Activates the CD player and the sound system
        {
            TV.tvOFF();
            Xbox.brOFF();
            Record.recOFF();
            CD.cdON();
            Sound.sysOn();
        }

        public void allOff()    // Deactivates all components
        {
            TV.tvOFF();
            Xbox.brOFF();
            Record.recOFF();
            CD.cdOFF();
            Sound.sysOff();
        }
           }
```

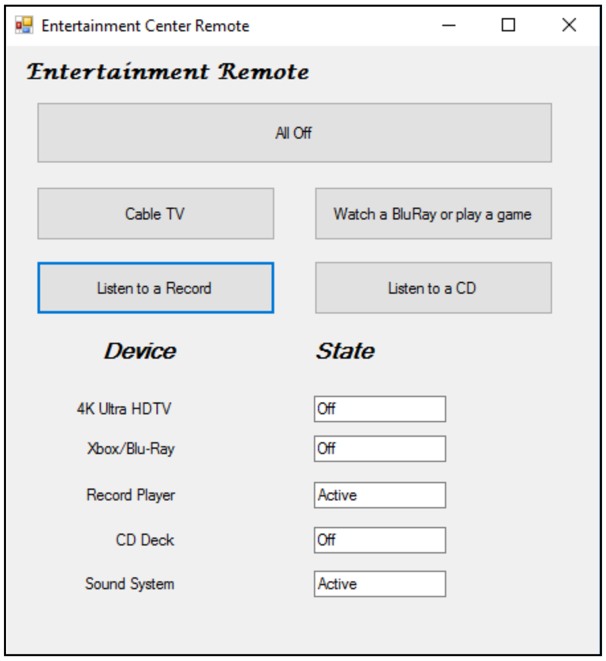
Finally, the form implements the Façade class to handle the controls of the application. The compActive() method says that if a component is active, the textboxes in the form will display it as being active, and that if it is inactive, the form will display that it is off. It does this by reading from the Façade class every time a button is selected.

```
public partial class frmRemote : Form
    {
        Univ_Remote Remote;
        public frmRemote()
        {
            InitializeComponent();
            Remote = new Univ_Remote();
```

```
}

private void compActive()       // Method to determine if components are active
{
    if (Remote.TV.tvActive())
        txt_TV.Text = "Active";
    else
        txt_TV.Text = "Off";

    if (Remote.Xbox.xboxOn())
        txt_BR.Text = "Active";
    else
        txt_BR.Text = "Off";

    if (Remote.Record.turnTblOn())
        txt_Record.Text = "Active";
    else
        txt_Record.Text = "Off";

    if (Remote.CD.playerOn())
        txt_CD.Text = "Active";
    else
        txt_CD.Text = "Off";

    if (Remote.Sound.soundOn())
        txt_Sound.Text = "Active";
    else
        txt_Sound.Text = "Off";
}

private void btn_Off_Click(object sender, EventArgs e)
{
    Remote.allOff();
    compActive();
}

private void btn_TV_Click(object sender, EventArgs e)
{
    Remote.cableTV();
    compActive();
}

private void btn_BR_Click(object sender, EventArgs e)
{
    Remote.bluRay();
    compActive();
}

private void btn_Record_Click(object sender, EventArgs e)
{
    Remote.recPlay();
    compActive();
}

private void btn_CD_Click(object sender, EventArgs e)
{
    Remote.cdPlay();
    compActive();
```

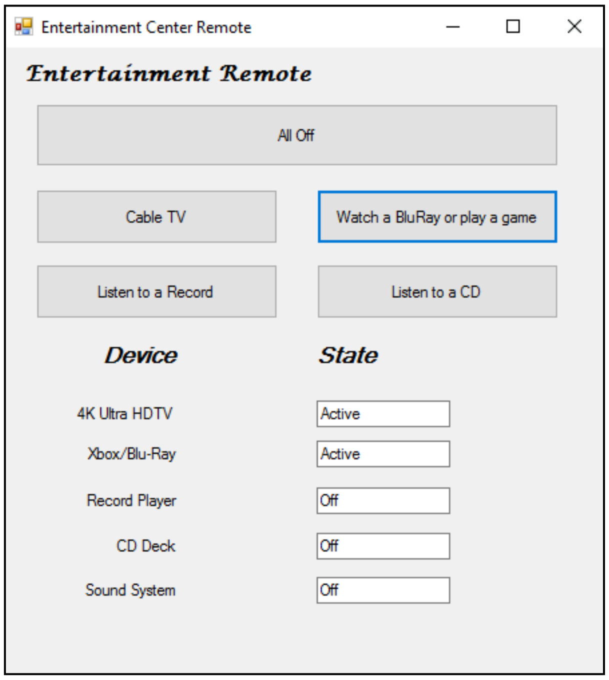
```
        }
    }
```

Listen To Record



Device	State
4K Ultra HDTV	Off
Xbox/Blu-Ray	Off
Record Player	Active
CD Deck	Off
Sound System	Active

Watch a Movie



Device	State
4K Ultra HDTV	Active
Xbox/Blu-Ray	Active
Record Player	Off
CD Deck	Off
Sound System	Off

Reflection

This pattern was simple once I figured out what it was asking for. I was confused as to what the Façade supposed to do. Looking up multiple definitions and seeing different examples made it a lot easier to grasp the concept. The only context I can see this pattern being relevant is in an all-controlling remote or something similar to that. I gained even more of an understanding of how classes depend on others. I think this design pattern taught a useful idea of simplification, having every control under the umbrella of one class.