Übungsblatt 3 (Lösung)

Aufgabe 8 Ein Eventsystem erstellen

Zunächst gilt es einen Container für die Ereignisargumente zu erstellen:

```
class EventArgs
{
    public EventArgs(String oldContent, String newContent)
    {
        OldContent = oldContent;
        NewContent = newContent;
    }
    public String OldContent
    {
        get;
        private set;
    }
    public String NewContent
    {
        get;
        private set;
    }
}
```

Anschließend wird die Definition für einen Observer erstellt:

```
abstract class Observer
{
    public abstract void Notify(Object sender, EventArgs e);
}
```

Die Collection, welche die einzelnen Observer verwalten soll, könnte folgendermaßen aussehen:

```
class ObserverCollection
{
    List<Observer> observers;

    public ObserverCollection()
    {
        observers = new List<Observer>();
    }

    public void Register(Observer observer)
    {
        if (!observers.Contains(observer))
            observers.Add(observer);
    }
    public void Unregister(Observer observer)
    {
        if (observers.Contains(observer))
```

```
            observers.Remove(observer);
    }
    public void NotifyAll(Object sender, EventArgs e)
    {
        foreach (var observer in observers)
            observer.Notify(sender, e);
    }
}
```

Jetzt noch eine beispielhafte *TextBox* erstellen:

```
class TextBox
{
    String text;
    ObserverCollection textChanged;

    public TextBox()
    {
        text = String.Empty;
        textChanged = new ObserverCollection();
    }

    public ObserverCollection TextChanged
    {
        get { return textChanged; }
    }
    public String Text
    {
        get { return text; }
        set
        {
            var oldValue = text;
            text = value;
            textChanged.NotifyAll(this, new EventArgs(oldValue, value));
        }
    }
    public void Clear()
    {
        Text = String.Empty;
    }
    public void AppendText(String text)
    {
        Text += text;
    }
}
```

Wichtig ist hierbei, dass jede Änderung entsprechend mit den Observern kommuniziert. In diesem Fall wird dies dadurch erreicht, dass jede Änderung über den Setzer der Eigenschaft *Text* läuft. Diese Methode ruft schließlich die *NotifyAll* Methode unserer Klasse zum Verwalten der Observer auf.
Abschließend noch eine Implementierung eines Observers, der Ereignisse direkt an die Konsole ausgibt:

```
class ConsoleTextObserver : Observer
{
    public override void Notify(Object sender, EventArgs e)
```

```
    {
        Console.WriteLine("Alter Text: {0}", e.OldContent);
        Console.WriteLine("Neuer Text: {0}", e.NewContent);
    }
}
```

Aufgabe 9 Zahlenketten

Am Anfang steht die Definition unseres Iterators. In dieser Aufgabe wird nur ein Integer-Enumerator verwendet, dennoch ist es wichtig dem UML Diagramm zu folgen. Daher muss ein generisches Interface erstellt werden:

```
interface Enumerator<T>
{
    T Current { get; }
    Boolean Next();
    void Reset();
}
```

Die wichtigste Implementierung dieses Interfaces findet in der *Range* Klasse statt. Diese Klasse symbolisiert einen Bereichsvektor. Anfangs sollte dieser auf einen ungültigen Wert initialisiert sein.

```
class Range : Enumerator<Int32>
{
    Int32 minimum;
    Int32 maximum;
    Int32 current;

    public Range(Int32 minimum, Int32 maximum)
    {
        this.minimum = minimum;
        this.maximum = maximum;
        this.current = minimum - 1;
    }

    public Int32 Current
    {
        get { return current; }
    }

    public Boolean Next()
    {
        if (current < maximum)
        {
            ++current;
            return true;
        }

        return false;
    }

    public void Reset()
    {
        this.current = minimum - 1;
```

```
    }
}
```

Eine weitere Implementierung findet sich in der Klasse *SquareRange* wieder. Diese Klasse nimmt einen bestehenden Iterator an und quadriert das aktuelle Element. Durch die Verwendung dieses Iterators, soll auch der dahinterstehende Iterator verändert werden.

```
class SquareRange : Enumerator<Int32>
{
    Enumerator<Int32> it;
    Int32 current;

    public SquareRange(Enumerator<Int32> it)
    {
        this.it = it;
        this.current = -1;
    }

    public Int32 Current
    {
        get { return current; }
    }

    public Boolean Next()
    {
        if (it.Next())
        {
            current = it.Current * it.Current;
            return true;
        }

        return false;
    }

    public void Reset()
    {
        this.current = -1;
        this.it.Reset();
    }
}
```

Die Klasse *EvenRange* sieht in etwa genauso aus, nur die Implementierung der *Next* Methode hat sich geändert. In diesem Fall gilt folgendes:

```
public Boolean Next()
{
    while (it.Next())
    {
        if (it.Current % 2 == 0)
        {
            current = it.Current;
            return true;
        }
    }
```

```
    return false;
}
```

Das grundsätzliche Feature von Iteratoren besteht darin, dass jegliche Liste (unabhängig von der dahinterstehenden Implementierung) ansprechen kann. Damit ist man viel flexibler als mit Arrays und kann eben auch Verkettung, wie mit dieser Aufgabe demonstriert, erstellen.

Aufgabe 10 4-3-3 und 4-4-2

In der Aufgabe sollte das Strategy Pattern anhand eines beispielhaften Fußballspiels angewendet werden. Zunächst sind einige Dummy-Klassen wie *SoccerMatch* oder *Team* zu erstellen.
Anschließend gilt es die *FormationStrategy* Basisklasse zu definieren.

```
abstract class FormationStrategy
{
    protected SoccerMatch match;
    protected Team team;

    public FormationStrategy(SoccerMatch match, Team team)
    {
        this.match = match;
        this.team = team;
    }

    public abstract void PlaceDefense();

    public abstract void PlaceMidfield();

    public abstract void PlaceStrikers();
}
```

Eine beispielhafte Implementierung dieser Basisklasse ist durch die Klasse *Formation433Strategy* gegeben. Zur Demonstration sollen einige Ausgaben beim Aufrufen der einzelnen Teilaufgaben erzeugt werden.

```
class Formation433Strategy : FormationStrategy
{
    public Formation433Strategy(SoccerMatch match, Team team)
        : base(match, team)
    {
    }

    public override void PlaceDefense()
    {
        Console.WriteLine("Defense has been placed according to 4-3-3.");
    }

    public override void PlaceMidfield()
    {
        Console.WriteLine("Midfield has been placed according to 4-3-3.");
    }

    public override void PlaceStrikers()
```

```
    {
        Console.WriteLine("Offense has been placed according to 4-3-3.");
    }
}
```

Dieses Fußballspiel könnte durchaus noch weiter ausgebaut werden. Der folgende objektorientierte Entwurf bildet eine solide Basis für dieses Unterfangen.