# Übungsblatt 4 (Lösung)

**Aufgabe 11** Eine leichtgewichtige DOM Implementierung

Zunächst wird die Definition eines Knotes benötigt. Der Knoten soll hierbei Operationen wie *ToHtml* oder *AddChild* ermöglichen.

```
abstract class Node
{
    public abstract String ToHtml();

    public abstract void AddChild(Node child);
}
```

Die Klassen *TextNode* und *Comment* sind sehr ähnlich. Beide erlauben keine Kindelemente und besitzen eine Eigenschaft für den Text.

```
class TextNode : Node
{
    public String Text
    {
        get;
        set;
    }

    public override void AddChild(Node child)
    {
        throw new DomException("Text does not support childs.");
    }

    public override String ToHtml()
    {
        return Text;
    }
}
```

Für das Dokument kann man eine ähnliche Struktur konstruieren, wobei die *AddChild* Operation so implementiert wird, dass man einem *Document* ein *Root* Element geben könnte.

```
class Document : Node
{
    public Element Root
    {
        get;
        set;
    }

    public override void AddChild(Node child)
    {
        if (Root == null && child is Element)
```

```
            Root = (Element)child;
        else
            throw new DomException("A document can only occupy one root element.");
    }

    public override String ToHtml()
    {
        return Root.ToHtml();
    }
}
```

Das *Element* ist sicherlich die wichtigste Klasse. Prinzipiell gibt es hier keine großen Überraschungen.

```
class Element : Node
{
    List<Node> children;

    public Element(String tagName)
    {
        children = new List<Node>();
        TagName = tagName;
    }

    public Node Parent
    {
        get;
        set;
    }

    public IEnumerable<Node> Children
    {
        get { return children; }
    }

    public String TagName
    {
        get;
        private set;
    }

    public override void AddChild(Node child)
    {
        if (child is Element)
        {
            var element = (Element)child;

            if (element.Parent != null)
                throw new DomException("Multiple parents are not allowed.");

            element.Parent = this;
        }

        children.Add(child);
    }
```

```
    public override String ToHtml()
    {
        var content = new String[children.Count];

        for (var i = 0; i < children.Count; i++)
            content[i] = children[i].ToHtml();

        return String.Format("<{0}>{1}</{0}>", TagName, String.Concat(content));
    }
}
```

Im Prinzip hätte man jeden *Node* mit einer Eigenschaft *Children* ausstatten können. Knoten wie *TextNode* hätten dann einfach einen leeren Iterator, d.h. keine Kinder, zurückgegeben.

### **Aufgabe 12** Flyweight für Namensräume

Die Definition des Shared Flyweight ist in diesem Fall sehr einfach.

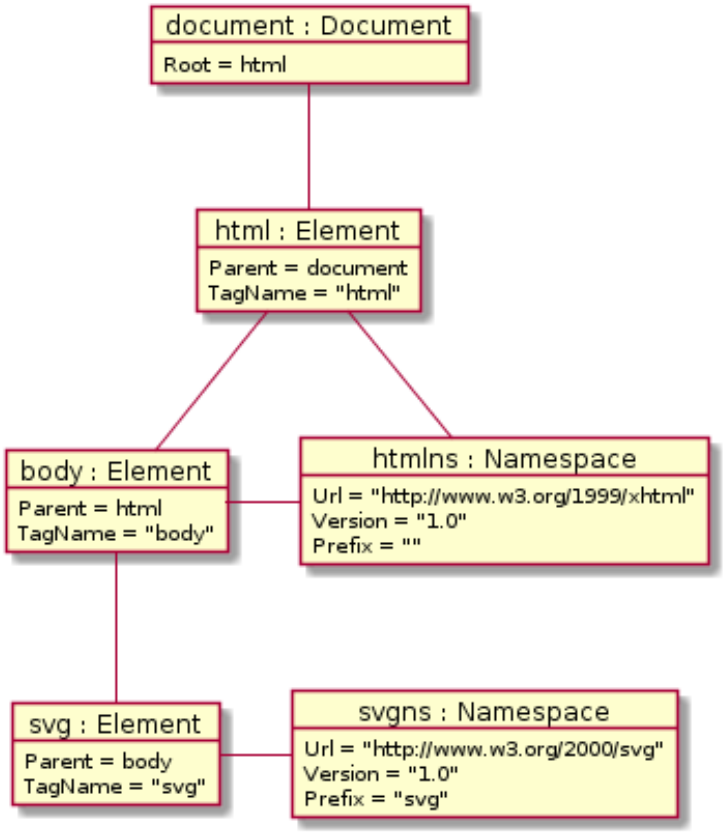
```
class Namespace
{
    public String Prefix
    {
        get;
        set;
    }

    public String Url
    {
        get;
        set;
    }

    public String Version
    {
        get;
        set;
    }
}
```

Dieses Beispiel ist dahingehend gekünstelt, dass hier nur drei Strings enthalten sind. In C# ist jeder String eine Referenz, wobei auch die Technik des String-Poolings angewendet wird. Somit hat man bei zwei identischen Instanzen sowieso nicht 6 Strings, sondern nur 3 Strings gespeichert.
Der Vorteil eines Flyweights ist bei dieser Struktur also nahezu vernachlässigbar. Insgesamt erhält man nur einen Faktor 3 (statt eines neuen Objekts mit 3 neuen Referenzen wird nur eine Referenz auf ein bestehendes Objekt benötigt).
Die Flyweight Factory kann wieder mit der Hilfe eines Singletons gebaut werden.

```
class Namespaces
{
    List<Namespace> namespaces;

    private Namespaces()
    {
```

```
        namespaces = new List<Namespace>();
    }

    static readonly Namespaces factory;

    static Namespaces()
    {
        factory = new Namespaces();
    }

    public static Namespaces Factory
    {
        get { return factory; }
    }

    public Namespace Find(String url, String prefix)
    {
        foreach (var entry in namespaces)
            if (entry.Url == url)
                return entry;

        return AddNamespaceFor(url, prefix);
    }

    public void Add(Namespace entry)
    {
        namespaces.Add(entry);
    }

    private Namespace AddNamespaceFor(String url, String prefix)
    {
        var ns = new Namespace
        {
            Url = url,
            Prefix = prefix,
            Version = "1.0"
        };

        namespaces.Add(ns);
        return ns;
    }
}
```

Im folgenden Objektdiagramm wird angenommen, dass es eine Instanz von *Document* aus Aufgabe 11 gibt, welche zum gegebenen Zeitpunkt den abgebildeten Elementbaum besitzt.




Die Instanz *htmlns* von *Namespace* wird von zwei Elementen gleichzeitig verwendet, d.h. referenziert.