02.11.2013

Coding Styleguide und Projektstandards

Autor: David Asmuth

Inhalt

C# Standards

Im folgenden sind Programmierrichtlinien aufgeführt die im offiziellen .NET Framework von Microsoft Anwendung finden.

1. Namenskonventionen:

- Klassen **und Methoden** werden im PascalCasing geschrieben. Also beginnend mit einem Großbuchstaben und jedes Teilwort erhält wieder einen großen Buchstaben.

Beispiel:
Richtig: ClientConnection
Falsch: clientConnection

- Methoden Argumente und lokale Variablen werden im camelCasing schrieben. Wie PascalCasing, jedoch ohne vorangehenden Buchstaben.

Beispiel:
Richtig: itemCount
Falsch: itemcount / ItemCount

- Keine ungarische Notation. Bedeutet: keine vorangehenden Typ-Identifizierer an den Variablen. Heutzutage zeigen einem IDEs via ToolTip den Typ und Lokalität an.

Beispiel:
Richtig: counter / name
Falsch: iCounter / strName / _counter

- Keine SCHREIENDEN KONSTANTEN NAMEN. Vermeidet, auch wenn es Konstanten oder readonly Variablen sind, diese komplett Groß zu schreiben. Stattdessen ist hier PascalCasing zu verwenden. Der Grund ist das diese Schreibweise zu viel Aufmerksamkeit zieht, und somit den Lesefluss stört.

Beispiel:

Richtig: SuperEvilNpcBitName
Falsch: SUPEREVILNPCBITNAME

- Vermeidet Abkürzungen. Abkürzungen sind schlecht lesbar und können zu Inkonsistenz führen. In einem Dokument steht “towerHlp” im nächsten “towerHlpr” beide stehen für “towerHelper”. Einzige Ausnahme sind Abkürzungen die bereits im normalen (Informatiker-)Sprachgebrauch zu finden sind. (z.B. ID, XML, NPC, FTP usw.)

Beispiel:

Richtig: towerGroup
Falsch: towerGrp

- Natürliche Abkürzungen mit mehr als 2 Buchstaben werden im PascalCasing geschrieben. Grund: Erhöht die Lesbarkeit.

Beispiel:

Richtig: XmlParser / NpcController / TowerID
Falsch: XMLParser / NPCController

- Keine Unter_Striche in Namen. Erhöht die Lesbarkeit. Wir sind hier nicht in Ruby.

Beispiel:

Richtig: TowerWeapon
Falsch: Tower_Weapon

- Benutzung von vordefinierten Typ Namen anstatt System-Typ Namen. Im Gegensatz zu Java gibt es keine echten primitiven Daten-Typen. Alle Typen sind Objekte. Somit ist “string” und “String” dasselbe genau wie “int” und “Int32” oder “long” und “Int64”. Die vordefinierten Typen sind immer zu bevorzugen.

Beispiel:

Richtig: string name / double offset
Falsch: String name / Double offset

- Verwendung von “var” in lokalen Variablen. Ausnahme sind vordefinierte Datentypen (int, long usw.). Dies vermeidet Durcheinander insbesondere bei Generics.

Beispiel:

Richtig: var towers = new Dictionary<string, ConcurrentStack<Tower>>()
Vermeiden: Dictionary<string, ConcurrentStack<Tower>> towers = new Dictionary<string, ConcurrentStack<Tower>>();

- Klassennamen sind (zusammengesetzte) Nomen.

Beispiel:

Richtig: Shooter / DeadTower
Falsch: Shoot / Die

- Interfaces beginnen mit einem großen “I”. Da es kein “implements” und kein “extends” für Abhängigkeiten gibt, sondern nur den Doppelpunkt, weiss man sonst nicht ob dies nun ein Interface oder eine Vaterklasse ist.

Beispiel:

Richtig: class BasicTower : ITower, GameObject // Implementiert: ITower & Erweitert: GameObject
Falsch: class BasicTower : Tower, GameObject // Erweitert oder implementiert Tower?

- Der Klassenname ist gleich dem Dateinamen, sowie der Namespace ist gleich der Project-Ordnerstruktur

Beispiel:

```
namespace Unity.GameObjects.Towers // Pfad: Unity\GameObjects\Towers\
{
        class BasicTower // Dateiname  BasicTower.cs
        {

        }
}
```

- Enums werden im Singular geschrieben und ohne typen ausgezeichnet (Letzteres nur wenn man mit Bitfeldern arbeitet). Zudem sollte kein Suffix “Enum” an den Namen gehängt werden.

Beispeil:

Richtig: public enum Direction
{
North,
East,
South,
West,
}

Falsch: public enum DirectionsEnum : int
{
North = 1,
East = 2,
South = 3,
West = 5
}

2. Ordungen

- Geschwungene Klammern starten in einer neuen Zeile

Beispiel:

```
class BasicTower
{

}
```

- Statische Felder kommen vor Feldern, die vor Gettern und Settern kommen.

Beispiel:

```
class BasicTower
{
    private static string TowerName;

    private int number;
    private double offset;

    public Number { get { return number; } set { number = value; } }

    // Konstruktoren hier

    // Methoden hier
}
```

3. Zugriffe

- Zugriffe auf Felder anderer Klassen finden über Getter und Setter Methoden statt. Kein Direktzugriff.

Beispiel:

Richtig:

```
private int length;
public int Length { get { return length;  } set { length = value;} }
```

Falsch:

```
public int Length;
```

Unity und Projekt Anpassungen

Da Unity den C#-Code schon während der Design-Zeit interpretiert (Serialisiert), um z.B. Öffentlich Felder im Unity-Editor anzeigen und bearbeiten zu können, müssen Anpassungen an den Standards vorgenommen werden.

C# Code anpassungen

- Da der Unity-Editor nur öffentliche Felder serialisiert (also keine Getter/Setter), wir aber unsere Zugriffe von Außen sauber Kapseln wollen, müssen wir unsere privaten Felder als “SerializeField” markieren, damit diese dennoch von Unity serialisiert werden und im Unity-Editor erscheinen. Dies geschieht natürlich nur mit Feldern die wir auch im Editor benötigen.

Beispiel:

```
[SerializeField]
private float health;
```

- Namespaces für Komponenten entfallen, da Unity nicht damit arbeitet. Die OrdnerStruktur des Projects ergibt sich somit aus der Assit-Struktur in Unity, da der Programm-Code ebenfalls als Asset gewertet wird.

- Die geschwungene Klammerung beginnt NICHT in einer neuen Zeile. Da dies Platzsparender ist.

- Keine Variablen die nur aus einem Zeichen bestehen. Ausnahme sind Lauf-Variablen (z.B. in schleifen)

Hinzugefügt 10.11.2013

- Assets für die Benutzeroberfläche werden mit dem Prefix Gui gekennzeichnet.

Beispiel:

class GuiMenu

Struktur

Geändert 10.11.2013 Anpassung an das Resource System.

- Die Asset-Struktur wird folgendermaßen zusammengesetzt: Da die C# Componenten immer einer bestimmten Gruppe Innerhalb des Projekts zuzuordnen sind (z.B. Komponente TowerWeapon zu der Gruppe der Türme) wird für diese Gruppe ein Asset-Verzeichnis angelegt. In diesem Verzeichnis liegen die C#-Komponenten. Alle weiteren sind sog. Resourcen, die vom Spiel ge- oder entladen werden können. Resourcen die zu dieser Gruppe gehören sind in einzelnen Sub-Verzeichnissen die dessen Typ beschreiben unterteilt.

Beispiel: ([Verzeichnis] {File})

```
[Assets]  (Root)
    |__ [Tower]
            |__[Resources]
            |         |__ [Models]
            |         |         |__ {Base.mdl}
            |         |         |__ {Cannon1.mdl}
            |         |__ [Textures]
            |         |         |__ {BaseTex.tga}
            |         |__ [Materials]
            |         |         |__ {BaseMat}
            |         |__ [Sounds]
            |                   |__ {shoot.mp3}
            |                   |__ {reload.mp3}
            |__ {TowerAttack.cs}
            |__ {TowerShootController.cs}
```

Gültigkeit

Die Regeln sind spätestens beim Zuführen ins öffentliche Repository eingehalten.