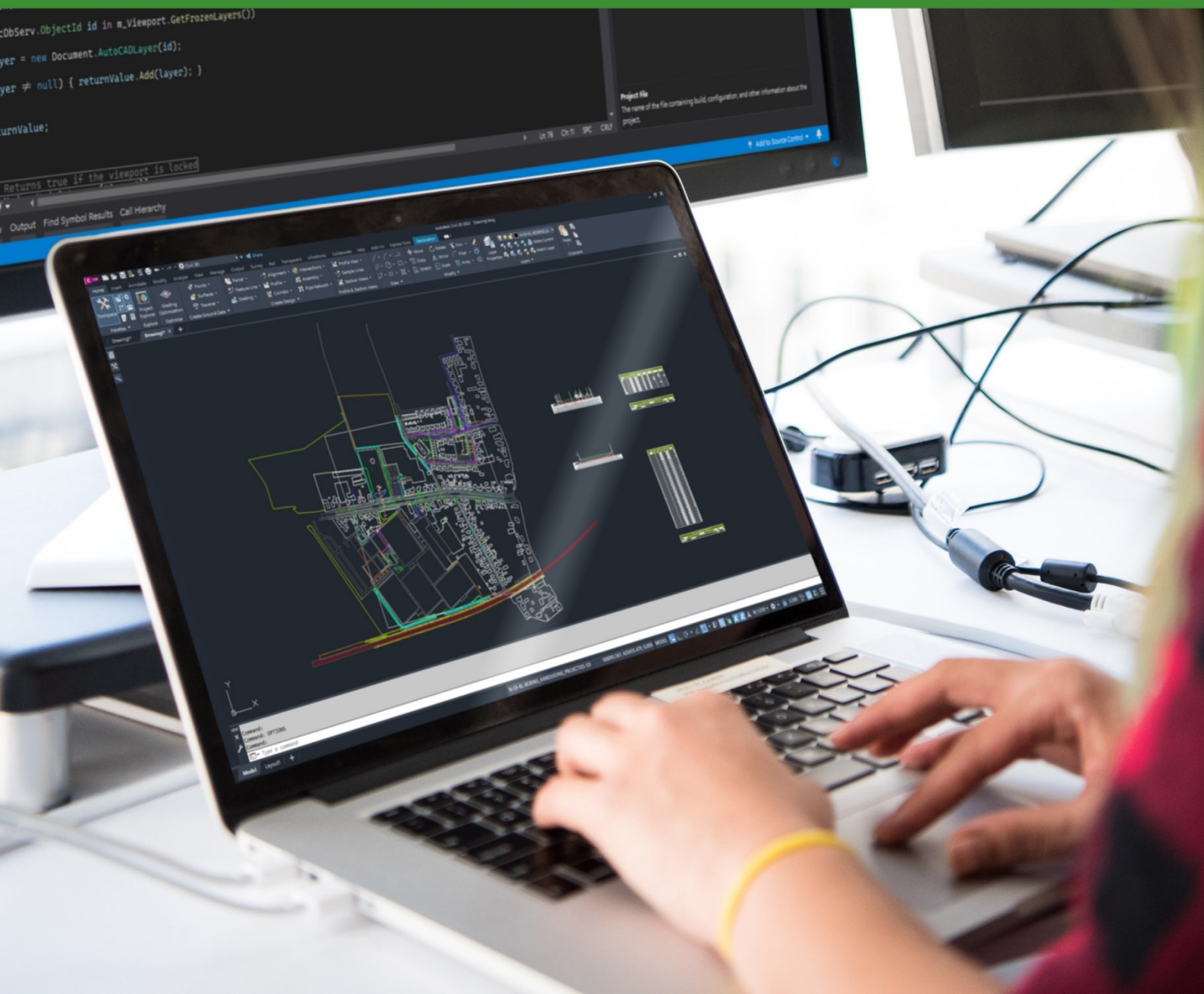




Anton Huizinga

Coderingsafspraken en richtlijnen



Afspraken en richtlijnen voor het schrijven van broncode

Anton Huizinga

Versie 2.0 - laatst gewijzigd 01-05-2022

© Anton Huizinga

www.huiz.net

www.linkedin.com/in/antonhuizinga/

Autodesk®, AutoCAD®, DWG® and the DWG logo are registered trademarks or trademarks of Autodesk, Inc., and/or its subsidiaries and/or affiliates in the USA and other countries.

AutoCAD screen shots reprinted courtesy of Autodesk, Inc.

Microsoft® and Visual Studio® are trademarks of the Microsoft group of companies.

Visual Studio screen shots reprinted courtesy of Microsoft.

All other trademarks are the property of their respective owners

VOORWOORD

Het hanteren van afspraken en richtlijnen is niet alleen onmisbaar voor samenwerkende programmeurs, ook voor zelfstandige programmeurs is het essentieel om gestructureerd te gaan werken.

Code die je geschreven hebt en pas maanden of jaren later nog eens doorneemt, kan je soms achter de oren doen krabben vanwege de onduidelijkheden. Waarom heb je het toen zo geschreven? Sliep je soms, en wat verwachtte je eigenlijk van die spaghettibrij?

Daarom is het verstandig om consequent te gaan programmeren, dezelfde structuren te blijven hanteren om andere programmeurs en uiteraard jezelf te voorzien van leesbare en duidelijke code.

Dit boekwerk bevat *mijn* afspraken-set. Hoe die er van jou uit komt te zien weet ik niet. Je mag deze set gebruiken, je mag deze set aanpassen of je mag voor jezelf een volledig nieuwe set schrijven. Je mag de helft van de richtlijnen schrappen of verdubbelen.

Als je het maar gaat gebruiken... ☺

Anton Huizinga

*"Real programmers don't comment their code.
If it was hard to write, it should be hard to read."*

INHOUDSOPGAVE

1	Algemeen	1
	Principes	1
	Terminologie	1
2	Algemene standaarden	2
	Duidelijk en consistent	2
	Opmaak en stijl	2
	Libraries	2
	Variabelen	3
	Initialisatie van variabelen	3
	Parameters	3
	Statements	4
	Enums	4
	Witruimte	5
	Accolades	5
	Commentaar	6
3	.NET Coderingstandaarden	8
	Bestanden en structuur	8
	Assembly Properties	8
	Versies	8
	Naming convensions	8
	Constants	10
	Strings	10
	Arrays, Lists en Collections	10
	Structures	10
	Classes: Fields/Properties	11
	Classes: Constructors	11
	Classes: Methods	11
	Classes: Events	11
	Classes: Member overloading	11
	Classes: Static Classes	11
	Namespaces	12
	Exceptions	12
	Resource Cleanup: Garbage Collector	12
	Resource Cleanup: Try-Catch-Finally	12
	Resource Cleanup: Dispose	12
	Interop	12
4	AutoCAD API	13
	Aliases	13
	Doubles	13
	Drawing Lock	13
	Transactions	13

Database14

5 Nawoord..... 15

1 ALGEMEEN

Dit document beschrijft een uitwerking van afspraken en richtlijnen voor het schrijven van broncode.

Geen enkele set van afspraken of richtlijnen zal afdoende zijn voor alle programmeurs in de wereld. Het doel van een standaard is om efficiëntie te stimuleren voor één of samenwerkende programmeurs. Het zal resulteren in minder bugs en beter onderhoudbare code. Het doel van deze standaard is om jou er van bewust te maken hoe je beter code gaat schrijven. Het triggert je hopelijk om na te denken over een set afspraken en richtlijnen, of je nu deze set gebruikt of een hele nieuwe set schrijft.

In het begin zal het lastig en vervelend zijn om een/jouw standaard toe te gaan passen, maar na enige tijd zal het door gewenning juist plezierig worden om consequent te programmeren.

PRINCIPES

De principes van het toepassen van een standaard zijn:

- ✓ **Begrijpbaar:** dat wil zeggen dat de code leesbaar is door bijvoorbeeld het gebruik van volledige benaming in plaats van afkortingen, en het toepassen van documentatie waar het verhelderend werkt.
- ✓ **Correct:** dat inhoudt dat de code geen onverwachte fouten veroorzaakt door bijvoorbeeld vooraf alle waarden te checken.
- ✓ **Consistent:** waarmee je laat zien dat je telkens dezelfde structuren toepast, dit toont vakmanschap aan en bewijst dat je oog voor detail hebt.
- ✓ **Modern:** het achterwege laten van oude coderingstechnieken die vaak omslachtig en onduidelijk zijn.
- ✓ **Legaal:** geen hacks of code die permanente wijzigingen in de computer aanbrengt. Alle veroorzaakte wijzigingen moeten omkeerbaar zijn. Hiermee wordt niet bedoeld het met toestemming van de gebruiker verwijderen of aanmaken van bestanden.
- ✓ **Zeker:** waarmee bedoeld wordt dat versienummers kloppen, duidelijkheid is over de gekoppelde bibliotheken en geen verwarring kan ontstaan welke functionaliteit en/of bekende bugs nu wel of niet aanwezig zijn in welke versie.

TERMINOLOGIE

In dit boekwerk wordt regelmatig aangegeven wat je wel of niet moet doen, wat je wel of niet zou moeten doen, en waarom je iets zou doen. Dit wordt weergegeven als volgt:

Wel doen: doe het volgende ...

Niet doen: doe het niet op deze manier ...

Waarom: omdat het ...

2 ALGEMENE STANDAARDEN

Dit hoofdstuk geldt voor algemene zaken rondom het programmeren en niet met de specifieke taalstructuur van .NET talen.

DUIDELIJK EN CONSISTENT

Wees duidelijk en consistent.

Wel doen: Deze set afspraken streeft ernaar dat de geschreven code duidelijk leesbaar, gemakkelijk te begrijpen en eenvoudig te onderhouden is.

Wel doen: Deze set afspraken consequent toepassen op alle code die je schrijft.

Het is misschien een open deur intrappen maar je schrijft een set standaarden (of je neemt deze over) om je broncode te verbeteren. Pas het dan ook toe, en neem de tijd om het toe te passen. Elke keer dat je even iets op een andere manier doet omdat het dan tijdbesparend lijkt, zal je bezig zijn om je code te verslechteren en wordt de code minder goed onderhoudbaar.

Vergelijk het met door rood te rijden. Zodra je de regel negeert om niet door rood te rijden, resulteert dit onmiddellijk in een stukje onveilig weggedrag. Zowel voor jou als voor een medeweggebruiker, ook al heb je er op dat moment zelf weinig last van. Je kunt zelf een ongeluk krijgen. Je kunt iemand doodrijden omdat die persoon denkt veilig over te kunnen steken. Of iemand moet heel hard voor jou afremmen en veroorzaakt een kettingbotsing. Of andere mensen worden ook nonchalant en gaan net als jij afgesproken verkeersregels negeren. Totdat de samenleving een anarchie geworden is.

Denk daar nog maar eens aan als je je niet aan je eigen coderingsafspraken houdt.

OPMAAK EN STIJL

Niet doen: Tab-tekens gebruiken om code in te laten springen. Als je de bestanden eens bekijkt in een andere editor, dan kan het afwisselend gebruik van tabs en spaties voor een onleesbare brij zorgen.

Visual Studio kan zodanig worden ingesteld dat tabs automatisch worden omgezet in spaties. Hanteer een korte afstand van bijvoorbeeld 2 tot 4 spaties. Dat is meer dan genoeg om netjes in te springen.

Breek coderegels in leesbare blokken van een maximum lengte. Meestal zijn tekstregels tot zo'n 80 karakters leesbaar op een beeldscherm. Door de regels af te breken voorkom je horizontaal scrollen.

Gebruik een lettertype met vaste breedte, zoals 'Consolas' of 'Courier New'. Microsoft heeft ook het font 'Cascadia Code' beschikbaar gesteld, speciaal voor IDE's. Het lettertype maakt binnen Visual Studio gebruik van ligaturen (samengestelde tekens).

LIBRARIES

Niet doen: Linken naar bibliotheken die niet gebruikt worden. Dit veroorzaakt onnodige ballast bij het compileren en laden van je applicatie. Bovendien kan het verwarring veroorzaken bij andere programmeurs, waar wordt een gekoppelde bibliotheek nou eigenlijk aangeroepen in je code?

VARIABELEN

Wel doen: Declareer variabelen zo laat mogelijk in de code, pas als je ze nodig hebt of aan het begin van een blok code waarin je ze nodig hebt.

Beperk het gebruik van variabelen die je in de hele applicatie gebruikt. Indien nodig, groepeer deze dan in bijvoorbeeld een instellingen-class.

INITIALISATIE VAN VARIABELEN

Wel doen: Initialiseer (voorzie van een waarde) variabelen zodra je ze declareert. Ga er niet vanuit dat .NET wel met een correcte beginwaarde komt.

Goed:

```
string Tekst = "Dit is tekst.";
string Leeg = string.Empty;
```

Fout:

```
string Tekst;
double Getal;
```

Niet doen: Declareer geen meerdere variabelen tegelijk met dezelfde waarde. Het kan wel maar het zal ook fouten kunnen veroorzaken.

Goed:

```
string Tekst1 = "Dit is tekst.";
string Tekst2 = "Dit is tekst.";
```

Fout:

```
string Tekst1 = Tekst2 = "Dit is tekst.";
```

Waarom: Bij variabelen die rechtstreeks een waarde bevatten zal bovenstaande wel werken, maar het is slechter leesbaar. Variabelen die geen waarde bevatten maar die verwijzen naar een geheugenlocatie waar de waarde staat, gaan hiermee de mist in. Alle variabelen verwijzen dan naar dezelfde geheugenlocatie. Wijzig je vervolgens de tweede variabele, dan verwijst de eerste er plotseling ook naar. Ook al is het in .NET geen codeerfout, doe het nooit.

PARAMETERS

Wel doen: Sorteer parameters op een logische manier. Als je parameters "Links" en "Rechts" in deze volgorde verwacht, gebruik deze dan ook als zodanig en gebruik een gelijksoortige benaming.

Goed:

```
public void GetSpace(double left, double right, string name, int doors)
```

Fout:

```
public void GetSpace(double Right, int Doors, string Name, double Linkerkant)
```

Als je Out Parameters gebruikt, zet deze dan altijd achteraan. Eerst In Parameters, dan Out Parameters.

Niet doen: Voorkom zoveel mogelijk het gebruik van parameters met een default waarde. Denk

maar goed na over de parameters die je moet invullen. Bij default waarden zie je snel over het hoofd wat er gebeurt.

Goed:

```
public void FormateerSchijf(string schijfletter)
```

Fout (minder goed):

```
public void FormateerSchijf(string schijfletter = "C")
```

STATEMENTS

Niet doen: Plaats niet meer dan één Statement op een regel. Het maakt het lezen moeilijker en debuggen kan niet meer op die ene Statement.

Goed:

```
string Tekst = "Dit is tekst.";
string Leeg = string.Empty;
```

Fout:

```
string Tekst1 = "Dit is tekst."; string Leeg = string.Empty;
```

ENUMS

Wel doen: Maak gebruik van Enums als je vaste waarden wilt gebruiken in parameters. Het is daarmee duidelijker welke waarden je kunt toepassen en je krijgt hulp van de IDE.

Niet doen: Gebruik geen constanten of variabelen als een Enum beter voldoet.

Goed:

```
enum Kleur
{
    Rood,
    Groen,
    Blauw
}
```

Fout:

```
const int Rood = 0;
const int Groen = 1;
const int Blauw = 2;
```

Niet doen: Gebruik niet de functie Enum.IsDefined() omdat dit op de achtergrond nogal voor extra load zorgt.

Wel doen: Stel als het kan een 0-waarde in met een duidelijke naam. Gebruik ook consequent deze naam, bijvoorbeeld "None".

Voorbeeld:

```
enum Richting
{
    None = 0,
    Links,
```

```
    Rechts
}
```

Dit past uiteraard niet bij Enums die geen nul-waarden zouden moeten retourneren. Bijvoorbeeld bij kleuren is het niet logisch om de optie "None" te retourneren:

```
enum Kleur
{
    None = 0,
    Rood,
    Groen,
    Blauw
}
```

Wel doen: Stel bij Enums die Bitwise moeten ondersteunen, de Flags Attribute in. Stel default waarden in met telkens een macht-tot-de-twee.

Niet doen: Stel geen defaultwaarde van 0 in bij Enums die Bitwise moeten ondersteunen.

Goed:

```
[Flags]
enum Toegangsrecht
{
    Lezen = 1,
    Schrijven = 2,
    Verwijderen = 4,
    Kopieren = 8
}
```

Fout:

```
[Flags]
enum Toegangsrecht
{
    Onbekend = 0,
    Lezen = 1,
    Schrijven = 2,
    Verwijderen = 4,
    Kopieren = 8
}
```

Waarom: Bij Bitwise Enums zal in het foute voorbeeld "Onbekend" altijd geldig zijn.

WITRUIMTE

Wel doen: Het wordt aanbevolen om witruimte toe te passen. Bovenaan elk blokje dat los staat van de vorige statement, zou een witregel moeten komen. Dit leest veel gemakkelijker. Beperk het tot één witregel, behalve boven elke functie. Scheidt elke functie met minimaal 2 en liever 3 witregels zodat je sneller door de code kan scrollen en kan zien waar een nieuwe functie begint.

ACCOLADES

Accolades worden gebruikt om blokken code samen te voegen tot bij elkaar horende code. Bij if-

constructies hoef je van .NET geen accolades te gebruiken als er maar één statement wordt uitgevoerd, maar leer jezelf aan om het wel te doen.

Wel doen: Altijd accolades toepassen bij een conditie.

Goed:

```
if (a == 5)
{
    b = a * a;
}
```

Fout:

```
if (a == 5)
    b = a * a;
```

Waarom: Met accolades leest het veel beter, je weet nu wat allemaal wordt uitgevoerd als de conditie waar is. Bovendien is het in bovenstaande nog redelijk in te schatten wat wordt uitgevoerd bij een geldende conditie maar bij een using-constructie wordt het ingewikkelder.

Fout:

```
using (Database db = Application.DocumentManager.MdiActiveDocument.Database)
using (Transaction tr = db.TransactionManager.StartTransaction())
{
    BlockReference brP = null;
    Point3d ptLocation = new Point3d();
    double x = 0.0;
    double y = 0.0;
    ...
}
```

Wat wordt in bovenstaand voorbeeld uitgevoerd binnen de eerste using? De code zal prima werken maar is niet duidelijk te lezen. Gebruik dus altijd accolades.

COMMENTAAR

Wel doen: Schrijf commentaar dat samenvat wat een stuk code gaat doen. Schrijf dit als volledige tekst met beginhoofdletters en punctuatie.

Niet doen: Herhaal niet de hele code in het commentaar.

Wel doen: Inline commentaar plaats je of boven een statement of achter een statement. Bovendien doe je dit alleen waar het verhelderend werkt. Bijvoorbeeld bij variabelen die je bovenaan declareert, om uit te leggen waarom je deze gaat gebruiken in de code. Of bij een lastige constructie om uit te leggen bij welke stap je nu bent.

Niet doen: Schrijf niet achter elke regel een stuk commentaar. Je hoeft echt niet meer commentaar dan code te schrijven. Als je meer kwijt wilt dan kun je dat ook samenvatten boven dat stuk code of boven de functie zelf.

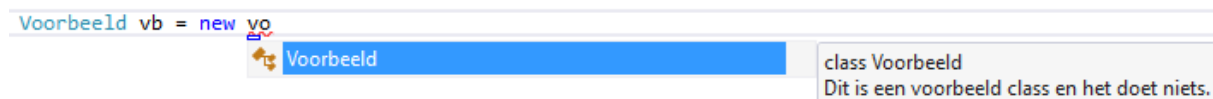
Wel doen: Voeg een File Header toe aan elk bestand met daarin de naam van het bestand, een korte omschrijving van de functionaliteit, de auteur en eventueel de licentie. Maak het ook niet te complex, volledig versiebeheer van het bestand is vaak over de top.

Wel doen: Voeg commentaar toe boven de declaratie van een Class en boven elke functie. Doe dit met een XML Documentation Comment. IntelliSense zal dit gebruiken om je van de juiste informatie te voorzien als je een Class of een functie aanroept.

Voorbeeld:

```
/// <summary>
/// Dit is een voorbeeld class en het doet niets.
/// </summary>
public class Voorbeeld {
```

In Visual Studio kun je beginnen met typen van drie slashes en dan zal de IDE zelf zo'n commentaarblok genereren. Zodra je dit hebt toegevoegd, zal de IDE je van hulp voorzien.

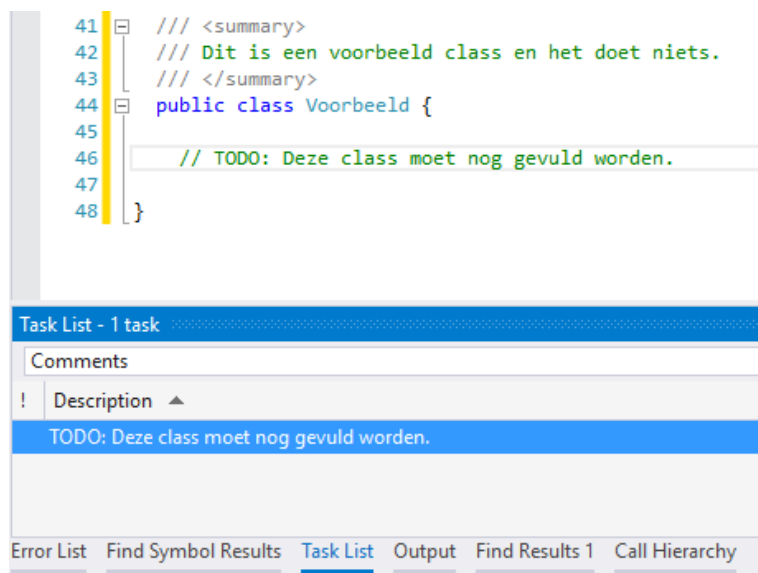


Wel doen: Voeg TODO commentaar toe waar je nog bezig bent of verwacht iets uit te breiden. Voordat je de applicatie definitief maakt, kun je alle TODO commentaren nog langslopen en bijwerken met de juiste code. Het helpt je bepaalde zaken niet te vergeten en stuurt je direct naar de juiste plaats.

TODO commentaar schrijf je als:

```
// TODO: Deze class moet nog gevuld worden.
```

Dit zal resulteren in een lijst in de Task List. Door dubbel te klikken op zo'n melding kom je gelijk op de juiste plaats.



3 .NET CODERINGSTANDAARDEN

Dit hoofdstuk richt zich inhoudelijk op afspraken en richtlijnen met betrekking tot de .NET structuur.

BESTANDEN EN STRUCTUUR

Wel doen: Noem de bestanden naar de inhoud. Bijvoorbeeld een Class met de naam *MainForm* hoort in een bestand met de naam 'Mainform.cs', of eventueel met een voorloper zoals een Namespace: 'MijnApp.MainForm.cs'.

Niet doen: Vermijd om meerdere Classes in één bestand te plaatsen. Je vindt het minder snel terug als een Class met de naam *Kleur* ergens onder een Class met de naam *Lijntypes* in een bestand met de naam 'MijnApp.Lijntypes.cs' staat. Die Class hoort in een eigen bestand met de naam 'MijnApp.Kleur.cs'.

ASSEMBLY PROPERTIES

Vul de Properties van de uiteindelijke Assembly altijd in met duidelijke beschrijvingen zoals bedrijfsnaam, applicatienaam, omschrijving en versie.

VERSIES

Wel doen: Bedenk hoe je versienummering gaat toepassen. Gebruik je Major en Minor? Of ook Builds? Kleine wijzigingen in een Build of in een Minor?

Werk de versie altijd bij als je een build uitgebracht hebt. Hiermee voorkom je dat er verschillende varianten van dezelfde versie op de markt zijn. Bovendien is het lastiger te installeren als er voor de computer al een gelijke versie aanwezig is.

NAMING CONVENTIONS

Wel doen: Gebruik altijd betekenisvolle namen voor alles. Gebruik ook altijd consequent dezelfde stijl. Alle functies die iets ophalen laat je beginnen met bijvoorbeeld 'Get...' en alle functies die iets instellen laat je dan beginnen met 'Set...'.

Niet doen: Er is geen ruimtegebrek op een computer. Gebruik daarom nooit afkortingen, tenzij het een algemeen gebruikte afkorting is. 'Proc' is een algemeen bekend woord voor 'Procedure', maar als het kan verwarren met 'Process' dan kun je het beter voluit schrijven.

Een prachtige opmerking over het verkeerd gebruik van afkortingen: "*If you say 'plz' because it is shorter than 'please', then I will say 'No' because it is shorter than 'Yes'.*"

Wel doen: Gebruik altijd Engels om iets te duiden, tenzij het een typisch product is voor een markt in een bepaald land. Een functie die voor een Nederlandse gemeente een huursubsidietoeslagpercentage uitrekent, hoeft niet krom vertaald te worden naar *GetRentSubsidySurchargePercentage()* of *GetHousingBenefitAllowanceRate()*.

Niet doen: Hanteer naamgeving zoals Visual Studio ze verwacht, dus geen functies laten beginnen met een getal, gebruik liever helemaal geen numerieke waarden in een naamgeving. Ook speciale

letters kunnen beter niet gebruikt worden, ook als de IDE het ze zou accepteren.

Wel doen: Sorteer de delen van een naam altijd logisch, woorden als Get of Set altijd vooraan. De rest van belangrijk naar minder-belangrijk.

Wel doen: Gebruik *PascalCase* of *camelCase* voor naamgeving van objecten.

Object:	Naamgeving:
Class of Structure	public class NaamVanClass
Namespace	namespace MijnBedrijf.Functies
Flag Enum	public enum MijnWaarden
Enum	public enum MijnWaarde
Methode	public void PrintNaarPapier()
Public Field	public string NaamBedrijf
Private Field	private string m_NaamBedrijf
Variable	string naamBedrijf

Over het algemeen gebruik je *PascalCase*, dat houdt in dat alle afzonderlijke woorden met een hoofdletter beginnen. Alleen variabelen binnen een functie worden geschreven met *camelCase*, waarbij de eerste letter klein wordt geschreven. Ook private Fields die in een Class worden gebruikt, worden met een kleine letter geschreven. Aan te bevelen is om hiervoor een "m_" als prefix te gebruiken zodat er verschil is met de variabelen binnen een functie.

Wel doen: Voorzie de naam van een Interface met een hoofdletter I als prefix. Voorzie Generic Type Parameters van een hoofdletter T als prefix.

Wel doen: parameters worden met *lowercase* of *camelCase* geschreven.

```
public void GetCalculatedValue(double length, double width)
```

Niet doen: Gebruik niet de ouderwetse prefix-naamgeving om variabelen een letter te geven volgens hun type.

Fout:

```
double dAngle = 0.0;
int iCounter = 1;
string sText = string.Empty;
```

Dit kan overigens wel handig zijn bij het benamen van Controls op een Form. Zodoende kan snel in de code bijvoorbeeld alle "Btn..." Controls worden gevonden. Veelgebruikte zijn:

Object:	Naamgeving:
Button	Btn...
CheckBox	Chk...
ComboBox	Cmb...
DataGrid	Dg...
Label	Lbl...
ListBox	Lb...
Panel	Pnl...
RadioButton	Rb...
TextBox	Txt...

CONSTANTS

Niet doen: Gebruik geen constanten als variabelen binnen codeblokken. Als je gebruik maakt van constanten, plaats deze dan bovenaan in de Class.

Je kunt wel gebruik maken van *read-only* Fields. Hiermee kun je prima een object maken van een Class en vervolgens *read-only* aanbieden binnen je applicatie.

STRINGS

Wel doen: Als je veel tekst wilt samenvoegen, gebruik dan een `StringBuilder`-object. Dit werkt beter dan teksten samenvoegen met de '+'-operator. Deze laatste kun je wel gebruiken als je korte teksten moet samenvoegen.

Gebruik de `Format`-methode om waarden in een string toe te passen, in plaats van losse teksten achter elkaar te plakken.

Goed:

```
string message = string.Format("Er zijn {0} items gevonden", items.ToString());
```

Fout:

```
string message = "Er zijn " + items.ToString() + " items gevonden";
```

Wel doen: Gebruik bij vergelijken van strings altijd een waarde voor het vergelijkingstype.

Goed:

```
if (message1.Equals(message2, StringComparison.OrdinalIgnoreCase))
```

Fout:

```
if (message1.Equals(message2))
```

ARRAYS, LISTS EN COLLECTIONS

Wel doen: Gebruik zo vaak mogelijk *Collections* of *Lists*. Deze types zijn zeer flexibel en prettig werkbaar, in tegenstelling tot *Arrays*.

Wel doen: Als je zelf een Collection type maakt, implementeer dan *IEnumerable* zodat je ook LINQ kunt toepassen op je object.

Niet doen: Retourneer geen *null* in je functies die normaliter een List, Array of Collection retourneren. Geef liever een leeg object terug.

STRUCTURES

Niet doen: Gebruik geen Struct als een Class beter functioneert. Structures kun je gebruiken voor eenvoudige objecten met een vluchtig bestaan, zonder uitgebreidere functionaliteit. Maak liever een Class.

Een AutoCAD `Point2d` of `Point3d` zijn Structures en bevatten twee of drie velden voor de X, Y, Z waarden. Dat zijn logische Structures. Als je zelf een Point type aanmaakt met meerdere Z-waarden en allerlei functies om daar iets mee te doen, gebruik dan een Class.

CLASSES: FIELDS/PROPERTIES

Wel doen: Maak gebruik van Properties in plaats van Public Fields. Properties kunnen ook gebruikt worden om Private Fields te tonen. Deze Properties kunnen eventueel *read-only* zijn terwijl de Private Field dat niet is.

Wel doen: Maak Properties *read-only* als gebruikers deze niet hoeven te wijzigen. Dit voorkomt een hoop ellende.

Niet doen: Gebruik geen *set-only* Properties. Je kunt hier niet zoveel mee. In plaats daarvan kun je ook een functie schrijven die de waarde in een Private Field plaatst.

Wel doen: Geef alle Properties een waarde in de Constructor.

CLASSES: CONSTRUCTORS

Wel doen: Stel alleen de Properties in en voeg eventuele parameterwaarden toe aan de Properties. Doe niets meer dan dat in een Constructor.

Wel doen: Als je een Constructor hebt met parameters, maak dan ook een default Constructor aan waarin je alle Properties een default waarde geeft. In de Overload Constructor hoef je vervolgens alleen de parameters te verwerken.

CLASSES: METHODS

Wel doen: Als je Out Parameters gebruikt, zet die dan aan het eind.

Bedenk goed hoe je je functie op zet. Wil je in de functie een waarschuwing doen? Wil je een leeg object retourneren bij een fout? Of retourneer je een Exception en handel je die af in de code die de functie aanroept?

CLASSES: EVENTS

Denk er om dat Events foutgevoelig kunnen zijn. Gebruik altijd Try {} Catch {} functionaliteit en probeer te voorkomen dat je in Events nieuwe Events veroorzaakt. Je kunt dit bijvoorbeeld oplossen door een *static bool* die bijhoudt of je een functie uitvoert. In de Event afvangende code controleer je of je een functie uitvoert of niet.

CLASSES: MEMBER OVERLOADING

Overloading is beter dan functies met default parameters.

Niet doen: Wissel niet van naamgeving in de parameters bij Overloaded functies. Wees consequent en consistent.

CLASSES: STATIC CLASSES

Niet doen: Maak zo weinig mogelijk gebruik van Static Classes. In Object Oriented Programming is het niet wenselijk. Als je het gebruikt, is het alleen voor hulpfuncties.

NAMESPACES

In sommige boeken wordt voorgesteld om Namespaces op te zetten als: *Bedrijfsnaam.Applicatie-naam.Functionaliteit*. Dit hoeft niet, je mag ook je eigen systeem verzinnen. Maar wees duidelijk.

EXCEPTIONS

Wel doen: Gebruik de meest logische Exception. Bijvoorbeeld een `ArgumentNullException` als een waarde `null` is.

Niet doen: Gebruik geen Exceptions om uit een functie te gaan omdat het nou eenmaal makkelijk is. Als een lege parameter wordt verstuurd (dus geen `null`), geef dan een bericht, retourneer `false` of doe iets anders, maar vuur geen Exception af.

Niet doen: Geen Exceptions afvuren vanuit een `Finally` block.

RESOURCE CLEANUP: GARBAGE COLLECTOR

Niet doen: Forceer het opruimen van objecten niet. Het heeft geen zin en het kan de werking van je applicatie verstoren als de verkeerde objecten zijn opgeruimd. De computer is prima in staat om zelf te bepalen wanneer een object opgeruimd moet worden. En computers hebben tegenwoordig ruimschoots geheugen.

RESOURCE CLEANUP: TRY-CATCH-FINALLY

Gebruik een `Finally` block om je objecten te sluiten, op te ruimen of te legen. Doe dit niet in een `Catch` block. Deze wordt namelijk niet altijd uitgevoerd, een `Finally` wel.

RESOURCE CLEANUP: DISPOSE

Ruim Disposable objecten op door de `Dispose` methode aan te roepen. Wat nog beter is, gebruik deze objecten in een `using`, dan worden ze automatisch opgeruimd, ook als een Exception optreedt.

Voorbeeld:

```
using (Transaction tr = db.TransactionManager.StartTransaction())
{
    ...
}
```

De `Transaction` wordt nu automatisch opgeruimd. Bij het compileren wordt deze structuur omgezet naar een `Try {} Catch {} Finally {}` structuur omgezet met in het `Finally` block een `Dispose` aanroep.

INTEROP

Wel doen: Verstuur een `Win32Exception` als dat nodig is na een P/Invoke fout.

4 AUTOCAD API

Dit deel beschrijft de afspraken die specifiek gelden voor de AutoCAD API.

ALIASES

Wel doen: Gebruik altijd aliases voor *usings* naar Autodesk Namespaces. Gebruik hiervoor altijd dezelfde benaming.

```
using acAppServ = Autodesk.AutoCAD.ApplicationServices;
using acDbServ = Autodesk.AutoCAD.DatabaseServices;
```

Vervolgens kun je in de code verwijzen naar:

```
acDbServ.Polyline poly = new acDbServ.Polyline();
```

Autodesk Classes willen nog weleens conflicteren met System Classes. Op deze manier zal je er geen last meer van hebben. Bovendien kun je nu code makkelijker schrijven voor AutoCAD look-alikes zoals BricsCAD.

DOUBLES

Doubles zijn van nature foutgevoelig. Als je in je functie een rekensom doet, dan kan dit een fout antwoord opleveren:

86.5 - 86.4 = 0.09999999999999943

Na zo'n 12 decimalen gaan doubles afwijken. Dus coördinaten vergelijken zal altijd een fout opleveren. Gebruik daarom altijd een afgerond getal in een vergelijking. Gebruik eventueel een functie daarvoor om getallen af te ronden voor het vergelijken.

Je kunt ook het resultaat vergelijken of dit kleiner is dan een klein getal. Als je wilt weten of het verschil 0 is (en dus gelijk), kun je ook kijken of het verschil kleiner is dan 0.0001 wat misschien wiskundig niet correct is maar wel het gewenste resultaat. De kans dat je iets nog nauwkeuriger wilt weten is erg klein, bovendien kun je dan beter met decimals rekenen in plaats van doubles.

DRAWING LOCK

Lock de tekening altijd als je een functie aanroept vanuit een Palette. Anders levert dit een `eLockViolation` Exception op.

```
using (acAppServ.DocumentLock acLckDoc = acAppServ.Application.DocumentManager.Mdi-
ActiveDocument.LockDocument())
{
    ...
}
```

TRANSACTIONS

Commit altijd een transactie, ook als je alleen een object leest. `Commit()` sluit de transactie netjes en snel af.

Gebruik altijd een *using* om een Transaction. Deze wordt altijd opgeruimd met een `Dispose` zonder dat je daar allerlei constructies voor hoeft te schrijven.

DATABASE

Er zijn meerdere Databases waarnaar verwezen kan worden: `ActiveDatabase` en `WorkingDatabase`. Deze hoeven niet gelijk te zijn. Normaal gesproken werk je altijd in de `ActiveDatabase` (`MdiActiveDocument`) en meestal is dit ook de `WorkingDatabase`.

Gebruik nooit een *using* om een Database. De actieve tekening zal niet worden opgeruimd met een `Dispose()`, ook al wordt deze aangeroepen in een *using*, maar om problemen te voorkomen dat een Database ineens uit het geheugen verdwijnt kun je dat beter niet doen.

Goed:

```
Database db = Application.DocumentManager.MdiActiveDocument.Database;
using (Transaction tr = db.TransactionManager.StartTransaction())
{
    Point3d ptLocation = new Point3d();
    ...
}
```

Fout:

```
using (Database db = Application.DocumentManager.MdiActiveDocument.Database)
{
    using (Transaction tr = db.TransactionManager.StartTransaction())
    {
        Point3d ptLocation = new Point3d();
        ...
    }
}
```

5 NAWOORD

Het belangrijkste van richtlijnen is dat je een structuur hebt om consequent en consistent te programmeren, wat de duidelijkheid en overzichtelijkheid ten goede komt en zaken niet voor meerdere interpretaties vatbaar zijn.

Het moet je niet beperken. Het moet je juist helpen door te sturen.

Hopelijk heb je aan deze voorbeelden een idee hoe je je eigen coderingsafspraken wilt opstellen.

Laat het me weten wat je ermee doet, en of je het succesvol of niet hebt kunnen toepassen!

En mocht je mijn boek 'Starten met programmeren in .NET voor AutoCAD' goed vinden, dan waardeer ik het enorm als je een positieve review achterlaat bij [Bol.com](https://www.bol.com).

Alvast bedankt!