Coding Standards

**Inhoud**

[1. Inleiding 3](#_Toc525745726)

[1.1 1 Algemeen 3](#_Toc525745727)

[1.2 2 Doel 3](#_Toc525745728)

[2. Naamgeving 3](#_Toc525745729)

[2.1 Algemeen 3](#_Toc525745730)

[2.2 Namespaces 4](#_Toc525745731)

[2.3 Classes, Structs & Interfaces 5](#_Toc525745732)

[2.4 Methodes 5](#_Toc525745733)

[2.5 Properties 5](#_Toc525745734)

[2.6 Test classes 6](#_Toc525745735)

[2.7 Test methods 6](#_Toc525745736)

[2.8 Events 6](#_Toc525745737)

[2.9 Fields 6](#_Toc525745738)

[2.10 Parameters 6](#_Toc525745739)

[2.11 Resources 7](#_Toc525745740)

[2.12 Directories 7](#_Toc525745741)

[2.13 Overige files 7](#_Toc525745742)

[3. Type Design 7](#_Toc525745743)

[3.1 Class of struct 7](#_Toc525745744)

[3.2 Abstracte classes 7](#_Toc525745745)

[3.3 Static classes 8](#_Toc525745746)

[3.4 Interfaces 8](#_Toc525745747)

[3.5 Enums 8](#_Toc525745748)

[4. Member Design 8](#_Toc525745749)

[4.1 Properties 8](#_Toc525745750)

[4.2 Constructors 8](#_Toc525745751)

[4.3 Events 8](#_Toc525745752)

[4.4 Fields 9](#_Toc525745753)

[4.5 Parameters 9](#_Toc525745754)

[5. Exceptions 9](#_Toc525745755)

[5.1 Algemeen 9](#_Toc525745756)

[5.2 Custom 404 pagina SharePoint (oud?) 10](#_Toc525745757)

[6. Logging 11](#_Toc525745758)

[6.1 Algemeen 11](#_Toc525745759)

[7. Code Formatting 11](#_Toc525745760)

[7.1 Algemeen 11](#_Toc525745761)

[7.2 Regions 11](#_Toc525745762)

[8. Unittest 12](#_Toc525745763)

[8.1 Naamgeving 12](#_Toc525745764)

[8.2 Project 12](#_Toc525745765)

[8.3 Structuur: Four-Phase Test Pattern 13](#_Toc525745766)

[8.4 Builder Pattern 14](#_Toc525745767)

[8.5 Xunit test Framework 14](#_Toc525745768)

[8.6 Mocking Framework 15](#_Toc525745769)

[8.7 Assertion Framework 15](#_Toc525745770)

[8.8 Testcoverage 15](#_Toc525745771)

[8.9 Try/Catch 15](#_Toc525745772)

[9. Commentaar 16](#_Toc525745773)

[9.1 Algemeen 16](#_Toc525745774)

[9.2 Class 16](#_Toc525745775)

[9.3 Fields & properties 16](#_Toc525745776)

[9.4 Methods 16](#_Toc525745777)

[9.5 EndPoints / Public APIs 16](#_Toc525745778)

[Korte samenvatting XML-commentaar 17](#_Toc525745779)

[10. Coding Conventions 17](#_Toc525745780)

[11. Algemeen 17](#_Toc525745781)

[11.1 Methods 18](#_Toc525745782)

[11.2 Security 18](#_Toc525745783)

[12. Coding Tools 18](#_Toc525745784)

[12.1 StyleCop 18](#_Toc525745785)

[12.2 ReSharper 19](#_Toc525745786)

[12.3 GhostDoc 19](#_Toc525745787)

[12.4 Code Analysis 19](#_Toc525745788)

[12.5 Code Metrics 20](#_Toc525745789)

[12.6 Log4Net 20](#_Toc525745790)

[12.7 Common.Logging 20](#_Toc525745791)

[12.8 Moq 20](#_Toc525745792)

[12.9 Fluent Assertions 20](#_Toc525745793)

[12.10 Ninject 21](#_Toc525745794)

[12.11 Automapper 21](#_Toc525745795)

[12.12 SafeNuGet 21](#_Toc525745796)

[13. Code Review 21](#_Toc525745797)

[14. Branching 21](#_Toc525745798)

# Inleiding

## 1 Algemeen

Deze Coding standards zijn een set van richtlijnen voor .NET & SharePoint die de programmeerstijl, praktijken en methoden aanbevelen voor elk aspect van een stuk software geschreven in deze taal. Deze richtlijnen zijn opgezet voor structurele software kwaliteit. Software programmeurs zijn sterk aanbevolen om deze richtlijnen te volgen om te helpen bij het verbeteren van de leesbaarheid van de broncode en creëren van eenvoudigere software onderhoud.

## 2 Doel

Het doel van dit document is om een totaalbeeld te geven van alle afgesproken ontwikkelrichtlijnen en standaarden voor de ontwikkeling van .NET applicaties. Dit is een werkdocument en kan worden aangevuld met nieuwe inzichten.

# Naamgeving

## Algemeen

* Gebruik verklarende namen, die duidelijk de intentie van de class/het project/de file laten zien.
* Gebruik geen nummers in namen tenzij het echt onderdeel uitmaakt van de naam. Dus geen VerwerkActie2, Wel Cit2Mutation. Namen beginnen altijd met een letter.
* Gebruik uitspreekbare namen, ga niet onnodig lopen afkorten, klinkers weg laten etc., dus daffodil i.p.v. dffdil, en verwerkRapport i.p.v. verwRprt.
* Gebruik namen waar makkelijk naar gezocht kan worden.
* In loops is het gebruik van i, j, k etc. toegestaan, voor de rest zijn dit geen geldige variabele namen.
* Voorkom namen die voorkennis vereisen die niet in het domein terug te vinden is, zoals ERIBES i.p.v. ELINE omdat dit in de geschiedenis ooit zo was.
* Gebruik geen namen die alleen verschillen in hoofd en kleine letters.
* Gebruik geen afkortingen of samentrekkingen als onderdeel van id-namen.
* Gebruik geen underscores of koppeltekens.
* Er zijn 2 manieren van hoofdletter gebruik: PascalCasing of camelCasing

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Keuze** | **Voorbeeld** |
| Namespace | Pascal | namespace   System.Security { ... } |
| Type | Pascal | public class   StreamReader { ... } |
| Interface | Pascal | public interface   IEnumerable { ... } |
| Method | Pascal | public class Object {   public virtual string ToString();   } |
| Property | Pascal | public class String {   public int Length { get; }   } |
| Event | Pascal | public class Process {   public event EventHandler Exited;   } |
| Field | Pascal | public class MessageQueue {   public static readonly TimeSpan   InfiniteTimeout;   }   public struct UInt32 {   public const Min = 0;   } |
| Enum value | Pascal | public enum FileMode   {   Append,   ...   } |
| Parameter | Camel | public class Convert {   public static int ToInt32(string value);   } |

## Namespaces

* Gebruik het volgende sjabloon:
  + <Bedrijf> (<Product> | <Technology>). [. <Feature>] [. <Subnamespace>]
  + UWV.Bravo
* Prefix Namespaces met UWV
* Overweg het gebruik van meervoud namespaces waar toepasselijk
* Gebruik niet dezelfde name voor een namespace en een class in de namespace

## Classes, Structs & Interfaces

* Als de class een interface implementeert laat dit dan ook terug komen in de naam van de class
* Gebruik implementatie specifieke namen als de class een bepaalde implementatie is van een interface: Dus liever HibernatePersoonDao dan DefaultPersoonDao of PersoonDaoImpl.
* Gebruik eerder Default dan Impl als toevoeging voor een implementatie waarbij geen specifieke naam te verzinnen is: Dus liever DefaultPersoonService ipv PersoonServiceImpl. Dit omdat default aangeeft dat het de standaard implementatie is, maar er eventueel andere kunnen zijn, terwijl impl impliciteerd dat het de implementatie van de Interface is.
* Gebruik voor Domein-classes altijd de Nederlands als. Zodat je als je met de business praat het over dezelfde entiteiten hebt.
* Gebruik een ‘I’ voor de naam om aan te geven dat het om een interface gaat.

## Methodes

* Method namen zouden een werkwoord in de naam moeten hebben, zoals get... set.., delete..., verwerk... etc.
* Methods moeten duidelijk aangeven wat ze doen d.m.v. de naam, dus niet verwerk, maar verwerkBankrekeningNummer.
* Als een method een boolean retourneert, dan wordt de naam niet GetJarig(), maar IsJarig().

## Properties

* Properties namen zouden een zelfstandig naamwoord, zelfstandig naamwoord zin of bijvoeglijk naamwoord moeten hebben, zoals public string TextWriter { get; set; }
* Benoem properties niet hetzelfde als een ‘Get’ methode, zoals public string TextWriter { get; set; } en public string GetTextWriter(int value) { … }
* Gebruik meervoud voor het aangeven van een collectie i.p.v. er ‘List’ of ‘Collection’ achter te zetten.
* Gebruik boolean properties met een bevestigende zin (CanSeek ipv CantSeek). Gebruik verder optioneel als prefix bij Boolean properties "Is," "Can," of  "Has," maar alleen als het waarde toevoegt.

## Test classes

* Unit testen hebben de naam van de class gevolgd door Tests, voorbeeld PersoonDaoTests, StringUtilTests
* Integratie testen hebben de naam van de class (indien er duidelijk een entry class voor testen is ) gevolgd door integrationTests, voorbeeld PersoonDaoIntegrationTests, PersoonServiceIntegrationTests.

## Test methods

* Test methods worden beschreven met de naam van de test methode met daarachter het verwachte resultaat. Dus NaamVanDeTeTestenMethode\_WatIsDeInputVoorDeTeTestenMethodeEnVerwachteResultaat, bijv. GetById\_WithNoIdShouldReturnNull
* Is er meer dan 1 test voor een method dan kan er doormiddel van \_omschrijving een omschrijving aan een testCase toegevoegd worden, voorbeeld: testGetById() en testGetById\_notFound();
* IntergrationTests moeten worden voorzien van een apart TestAttribute, zoals TestCategory[“Integration”]

## Events

* Events verwijzen altijd naar een actie (oftwel een gebeurtenis die optreed / is opgetreden). Noem daarom events (hetzelfde als methodes) altijd met het werkwoord van de gebeurtenis, zoals Clicked, Painting, DroppedDown.
* Gebruik het heden en verledentijd in het werkwoord van het event.
* Gebruik geen ‘before’ of  ‘after’ voor of achter het event.

## Fields

* Benoem ook constanten met PascalCasing (gebruik niet alleen hoofdletters of underscores)
* Namen van fields herhalen niet de naam van de class. Dus in class Bedrijfstak geen bedrijfstakId, maar gewoon id.

## Parameters

* Alleen bij setters mag de naam van een parameter dezelfde zijn als een reeds bestaand veld.
* Gebruik beschrijvende namen voor parameters.

## Resources

* Gebruik PascalCasing voor items in resources files
* Gebruik beschrijvende namen (en geen korte identifiers), zoals ArgumentExceptionIllegalCharacters, ArgumentExceptionInvalidName of ArgumentExceptionFileNameIsMalformed

## Directories

* Directory namen zijn met PascalCasing. Gebruik ook geen scheiding of andere speciale tekens.

## Overige files

* Overige files zullen bij voorkeur ook met kleine letters met als scheiding teken de '-' geschreven worden. Denk aan Javascript files, images en stylesheets, zoals UWV-style.css of UWV-logo.png.

# [Type Design](http://confluence.dnbad.nl/display/SD/03.+Type+Design)

## Class of struct

* Gebruik standaard altijd classes.
* Gebruik alleen een struct als het voldoet aan alle volgende voorwaarden:
  + Het vertegenwoordigt een enkele logische waarde, vergelijkbaar met primitieve types (int, double, enz.).
  + Het heeft een grootte onder de 16 bytes.
  + Het is onveranderlijk.
  + Het zal niet vaak worden omkaderd.

## Abstracte classes

* Je kunt geen instantie maken van een abstracte class, Gebruik daarom geen public constructors in een abstracte class, maar protected of internal.

## Static classes

* Maak minimaal gebruik van static classes. In principe alleen voor helper/utility classes.

## Interfaces

* Definieer in ieder geval 1 class met een implementatie van een interface, zodat je zeker weet dat je design van je interface goed is.

## Enums

* Gebruik een enum bij strongly type parameters, properties en return values die sets van waarden vertegenwoordigen.
* Gebruik (waar dat kan)  enums dan static constants

# [Member Design](http://confluence.dnbad.nl/display/SD/04.+Member+Design)

## Properties

* Gebruik get-only properties wanneer men niet de waarde van property mag wijzigen.
* Gebruik geen set-only properties or properties waarmee de setter een breedere accessibility heeft dan de getter (gebruik bijvoorbeeld geen properties met een public setter and een protected getter).
* Gebruik geen excepties in property getters.

## Constructors

* Probeer simpele constructors te maken met niet al te veel parameters.
* Gebruik dezelfde naam voor een constructor parameter wanneer deze gebruikt wordt om een property te zetten.
* Doe zo min mogelijk in een constructor (niet meer dan alleen opvangen van parameters).

## Events

* Gebruik een protected virtual method voor het opgooien van een event.

## Fields

* Maak zoveel mogelijk gebruik van (centrale) constanten (waar uiteraard nodig).
* Maak geen fields aan die public of protected zijn, gebruik hiervoor properties om deze te benaderen.

## Parameters

* Gebruik de minst afgeleide parameter type als dit kan,  zoals bijvoorbeeld IEnumerable als parameter, niet ArrayList of IList.
* Gebruik enums als paramater wanneer je anders 2 of meer boolean parameters zou hebben. Enums geven je meer ruimte in de toekomst om eventueel waardes toe te voegen.

# [Exceptions](http://confluence.dnbad.nl/display/SD/05.+Exceptions)

Error handling en exception handling worden geïmplementeerd in de applicatie zodat potentiële problemen met het systeem snel kunnen worden geconstateerd en hierop actie kan worden ondernomen.

## Algemeen

* Als er een fout situatie optreedt dient er in bijna alle gevallen een exceptie gegooid te worden ipv het returnen van een error code of een null value. Het returnen van een null value brengt naast allerlei gevaren ook nog veel extra conditie afvragingen in de code mee, die niet nodig zijn als er een exceptie gegooid wordt.
* Gooi de exception nooit terug naar de gebruiker (i.v.m. security), maar  laat een ‘sorry voor het ongemak’ zien en log de exception (zie hoofdstuk ‘LOGGING”).
* Gebruik liever geen ‘#if DEBUG’
* Gebruik geen try.. catch statements (o.a. ivm performance). Probeer alles netjes op te vangen met eventuele checks.
* Probeer try/catch blokken zoveel mogelijk in een eigen methode op te nemen zodat je vanuit andere punten deze methode kan aanroepen zonder een try catch blok.
* Alleen een try/catch als er op die plek ook echt iets aan de exception kan worden gedaan (afgehandeld). Anders geen try/catch en eventuele exceptions omhoog laten borrelen in bovenliggende code.
* Probeer een zo specifieke exeption om te gooien waar mogelijk
* Zorg voor een generieke exception handling binnen applicaties. Bijvoorbeeld middels een BaseController in MVC of een ErrorHandler behavior in WCF zodat in ieder geval alle exceptions worden gelogd.
* Gebruik nooit de Exception.Message, maar altijd Exception.ToString(), deze geeft je veel meer informatie om je issue straks op te lossen.
* Stop je ‘clean up code’ altijd in de finally blocks. Denk hierbij aan bijvoorbeeld een dispose (bijvoorkeur gebruik trouwens altijd using).
* Als er een fout optreedt wordt er een altijd een high-level bericht getoond aan de gebruiker: ‘Er is een fout opgetreden, probeer het later nog eens’. Het low-level bericht zal worden opgeslagen in de verschillende log bestanden.
* Een maatwerk error pagina in de huisstijl van de applicatie wordt gecreëerd om de standaard application page van SharePoint of ASP.NET te vervangen (Unexpected Error has occured).
* Alle objecten, waarden en eigenschappen moeten eerst worden gecontroleerd zodat er nooit een ‘Unexpected exception’ optreed, dit betekent onder andere dat:
  + Objecten altijd eerst op een null waarde moeten worden gecontroleerd.
  + String variabelen altijd eerst moeten worden op een lege waarde.
* Try/Catch blokken mogen nooit lege handlers bevatten. Elke fout melding moet worden afgehandeld in de handler zelf of op een hoger niveau.

Voor het loggen van fouten wordt gebruik gemaakt van een logging framework. Voor meer informatie hierover zie hoofdstuk [06. Logging](http://confluence.dnbad.nl/display/SD/06.+Logging)

## Custom 404 pagina SharePoint (oud?)

De 404 pagina betreft een pagina met een foutmelding die getoond wordt als de bezoeker een url aanroept die niet (meer) bestaat. In plaats van een standaard, onduidelijke, SharePoint of IIS melding dat de pagina niet gevonden is, wordt een nette pagina getoond.

De custom 404 pagina functionaliteit bestaat technisch uit de volgende onderdelen:

|  |  |
| --- | --- |
| Onderdeel | Omschrijving |
| Custom 404 page layout | Een 404 pagina pagina layout met het rich text veld “Pagina inhoud” en een webpart placeholder |
| custom sps404.html | Plaats deze custom sps404.html (bijv. dd404.html) in dezelfde map als de sps404.html (15\templates\layouts\1043)  Laat de custom 404 pagina verwijzen naar de pagina aangemaakt in stap 3 |
| fileNotFoundPage property property feature | Maak een feature die de FileNotFoundPage property van een Webapplication omzet naar ‘[custom 404 pagina url]’. Dit is een pagina die gebaseerd is op de custom 404 page layout  Deze feature dient aangeroepen te worden bij het uitrollen van de deployment solution. |

# [Logging](http://confluence.dnbad.nl/display/SD/06.+Logging)

## Algemeen

* Als logging framework wordt Log4Net gebruikt.
* Met Log4Net is het daadwerkelijke loggen volledig configurabel. Hoe (formaat) wat (Information, Warning, Error) en waar (file, e-mail, eventlog, database, etc.) de fouten worden gelogd wordt vastgelegd in de config van de applicatie en is eventueel runtime te wijzigen.
* Voor het loggen wordt gebruik gemaakt van Common.Logging als tussenlaag voor het daadwerkelijke logging framework. Hiermee wordt een extra abstractie laag gecreëerd tussen de applicatie en het gebruikte logging framework .
* De volgende evenementen worden gelogd:
  + gebruikersacties (Info, prioriteit 1)
  + onverwachte gebeurtenissen (Warning, prioriteit 3)
  + fouten (Fatal/Error, prioriteit 5)

Zie hoofdstuk [11. Coding Tools](http://confluence.dnbad.nl/display/SD/11.+Coding+Tools) voor meer informatie over de gebruikte frameworks.

# [Code Formatting](http://confluence.dnbad.nl/display/SD/07.+Code+Formatting)

## Algemeen

* Gebruik StyleCop voor het afdwingen van een eenduidige coding style volgens de standaard van Microsoft. Zie hoofdstuk [11. Coding Tools](http://confluence.dnbad.nl/display/SD/11.+Coding+Tools).
* Alle code formatting regels van StyleCop worden gevolgd (inspring, witregels, volgorde, etc) en worden daarom hier ook niet verder uitgewerkt.
* Maak je regels niet langer dan 132 characters. Als je regel toch te lang wordt kijk er naar of je deze kan breken in meerdere delen

## Regions

* Gebruik geen regions in je code, omdat ze meestal gebruikt worden om lange stukken code te verbergen.
* Als je dit toch gebruikt denk aan het volgende :
  + Gebruik nooit regions in methods
  + Gebruik niet meer dan ongeveer 5 regions
  + Gebruik zeker geen regions om ‘lelijke of lange’ code te hidden.

# [Unittest](http://confluence.dnbad.nl/display/SD/08.+Unittest)

De unittest controleert of de code van de specifieke functie technisch juist is. Alle externe afhankelijkheden die binnen de methode gebruikt zijn, kunnen worden opgevangen door mockobjecten. Hiermee test je alleen wat specifiek in die methode wordt gedaan. Hieronder staan enkele korte richtlijnen voor het opstellen van unittests.

## Naamgeving

De naam van een test class is de naam van de class die hij test plus “Tests”, bijvoorbeeld:  
De AntwoordServiceTests test de AntwoordService.

Uit de naam van de methode moet alles gelijk duidelijk worden:

-           NaamVanDeTeTestenMethode\_WatIsDeInputVoorDeTeTestenMethodeVerwachtteResultaat

Er hoeft niet persé met underscores te worden gewerkt in de naam. Bijvoorbeeld:

-           InitializeWithNullArgumentThrows

Het belangrijkst is dat er [DAMP](http://stackoverflow.com/questions/6453235/what-does-damp-not-dry-mean-when-talking-about-unit-tests) (Descriptive And Meaningful Phrases) wordt gewerkt binnen de naamgeving.

## Project

Elke solution heeft een folder ‘Tests’ met daaronder voor elk project apart een unittest project.

Binnen een project is er een mappenstructuur die de unit-tests scheidt van alle overige (helper) classes.  
Deze kan er bijvoorbeeld zo uitzien:

-          UWV.Bravo.Services.Tests

* Helper [map]
* Fixtures [map]
* UnitTests [map]
  + VraagServiceTests.cs
* IntegrationTests [map]
  + VraagServiceTests.cs

Door de map ‘UnitTests’ ipv ‘Tests’ te noemen voorkomen we een vervelende *namespace* als:

-          UWV.Bravo.Services.Tests.Tests.VraagServiceTests

Het is handig om een project in de ‘Tests’ een project te hebben met alle *helpers* en *builders* die over meerdere projecten gebruikt kunnen worden. Bijvoorbeeld: UWV.Bravo.Tests.Utils.

## Structuur: Four-Phase Test Pattern

Binnen de gedachtegang van xUnit (<http://en.wikipedia.org/wiki/XUnit>) bestaat het idee van vier fasen in een test:

* Setup
* Test
* Verify
* Teardown

Het idee is dat het duidelijk moet zijn binnen een test wat er gebeurt en dat een test niet afhankelijk moet zijn van een TestInitialize of een TearDown methode ergens anders in de class. Een methode is dus zelf verantwoordelijk voor de initialisatie en het opruimen van een class die hij test.

Check de volgende pagina voor meer info: <http://xunitpatterns.com/Four%20Phase%20Test.html>.

Deze fasen worden expliciet gemaakt door comments. Er hoeft niet persé ruimte tussen de afzonderlijke stukken gehouden te worden, de comments bieden het overzicht.

Bijvoorbeeld:

[?](http://confluence.dnbad.nl/display/SD/08.+Unittest)

|  |
| --- |
| [Fact]  Public void BuildModelReturnsModelWithCorrectBehandelaar()  {      // Fixture setup      var mockUserContextInfo = new Mock<IUserContextInfo>();      mockUserContextInfo.Setup(x => x.Inlognaam).Returns("TestUser");      var sut = new BeschouwingBuilder(mockUserContextInfo.Object);        // Exercise system      var result = sut.BuildModel();        // Verify outcome      result.SelectedUser.Should().Be("TestUser");      // Teardown fixture  } |

In de teardown kan een object ge-disposed worden indien nodig, wat in dit voorbeeld niet gebeurt.  
Wanneer je zelf een fixture object die ge-disposed moet worden, kan je deze ook in een using aanmaken. Dan wordt hij sowieso ge-disposed (door de try-finally van de using).

Zie de volgende confluence pagina voor meer info:  
[http://confluence.UWVad.nl/display/MIC/Unit+testing+met+xUnit?src=search](http://confluence.dnbad.nl/display/MIC/Unit+testing+met+xUnit?src=search)

## Builder Pattern

Het builder pattern zorgt ervoor dat je altijd een goed gevuld object terug krijgt op het moment dat je de “Build” methode aanroept. Door middel van properties en methods van de builder kun je bepaalde properties/objecten van een bepaalde waarde voorzien. Builders worden in aparte classes opgenomen zodat deze eenvoudig zijn her te gebruiken in verschillende tests.

Bijvoorbeeld: Beschouwing beschouwing = Beschouwing Builder.Build();

Door methodes aan te maken op de builder die het object zelf weer teruggeeft, kunnen er methodes gechained worden op leesbare wijze in een test. Zie deze confluence pagina voor een voorbeeld:

Icon

NOG TOEVOEGEN

Het is handig wanneer alle builders in een aparte class library staan, in de tests map in de solution.

## Xunit test Framework

We gebruiken naast MSTest ook Xunit als test framework, dit biedt veel voordelen. Lees er hier meer over:

[http://confluence.UWVad.nl/display/MIC/Unit+testing+met+xUnit?src=search](http://confluence.dnbad.nl/display/MIC/Unit+testing+met+xUnit?src=search)

Deze frameworks kunnen door elkaar worden gebruikt, in principe. Een xunit project hoeft echter geen UnitTest project te zijn, maar kan een class library zijn.

Note: Op dit moment (12-03-2015) veroorzaakt een referentie naar xunit 1.9.2 eeb probleem op de build server in een MSTest project (Unit test project).  
Ook een referentie in een class library *zonder* minimaal een xunit test (Fact of Theory) zorgt voor problemen. Er wordt gekeken of dit in de 2.0 versie wordt opgelost (deze is nog niet gereleased).  
Voor meer info: <http://xunit.github.io/>

## Mocking Framework

Een goede unit test draait volledig losstaand en maakt geen gebruik van omgevingsvariabelen zoals een database. Classes die de te testen class gebruikt voor het vervullen van zijn functionaliteit worden gemockt. Op deze manier wordt alleen de echte functionaliteit van de class getest.

Moq wordt gebruikt als mocking framework, zie hoofdstuk [11. Coding Tools](http://confluence.dnbad.nl/display/SD/11.+Coding+Tools).

## Assertion Framework

Met een assertion framework worden unit tests een stuk beter leesbaar. Een assertion framework bestaat vaak uit een aantal extension methods waarmee eenvoudig de verwachte waarde van een object kan worden gecontroleerd.

-          Zonder framework: Assert.AreEqual(“Hallo”, objectX)

-          Met framework: objectX.Should().Be(“Hallo”)

Fluent Assertions wordt gebruikt als assertion framework, zie hoofdstuk [11. Coding Tools](http://confluence.dnbad.nl/display/SD/11.+Coding+Tools).

De assertions in Xunit zijn ook leesbaar.

## Testcoverage

Hou bij testcoverage 80% aan voor je nieuwe code. Uitzondering is het testen van de GUI code (denk aan javascript e.d.).

## Try/Catch

Gebruik voor de code in een unittest geen try/catch blokken. Het idee hierachter is om de testcode zo klein mogelijk te houden en daarmee de leesbaarheid te vergroten. Standaard gedrag is dat een unittest faalt wanneer er een exception gegooid wordt. Overigens kan een exception natuurlijk ook het gewenste gedrag van de test zijn.  
Het testen binnen xunit geeft ook een goede mogelijkheid om te testen of iets een exception geeft, specifieker dan het ExpectedException attribute van MSTest.

# [Commentaar](http://confluence.dnbad.nl/display/SD/09.+Commentaar)

## Algemeen

* Gebruik triple-/ (XML-commentaar) voor **public** methodes
* Beschrijf niet wat er gebeurt (kun je namelijk lezen in de code), maar waarom er iets gebeurt / wordt gedaan.
* Alle commentaar moet in het Engels geschreven worden. Aangezien de naamgeving van domein objecten in het Nederlands wordt gedaan, worden deze ook zo gebruikt in commentaar (en dus niet vertaald naar het Engels).
* Geen inline commentaar. Inline commentaar is een code smell waarbij de methode dus beter kan worden opgedeeld in verschillende methodes waardoor deze weer leesbaar is zonder commentaar.
* Op het moment dat je een comment wilt plaatsen bij code, kijk dan of je via extract method en een logische naamkeuze voor die method de code duidelijk kan maken.
* Er mag gebruikt gemaakt worden van TODO commentaar tags, maar bij voorkeur niet. Het is de bedoeling dat wanneer de software wordt opgeleverd er natuurlijk sowieso geen TODOs meer inzitten.
* Gebruik eventueel GhostDoc voor semi-automatisch commentaargeneratie. Let dan wel op fatsoenlijk commentaar en pas het automatische gegenereerde commentaar aan. Zie hoofdstuk  [11. Coding Tools](http://confluence.dnbad.nl/display/SD/11.+Coding+Tools).

## Class

* Een class kan voorzien worden van commentaar waarin omschreven wordt wat de class representeert, of welke functionaliteit deze toevoegt.

## Fields & properties

* Neem bij velden, constanten & properties in principe nooit commentaar op. Kijk eerst of je door een betere naam te kiezen dit kan voorkomen.

## Methods

* Als een methode een logische naam en signatuur heeft is het niet nodig deze te documenteren. Mocht je bezig zijn met een algemene class probeer dan wel de hele public api, en de extension points goed te documenteren.

## EndPoints / Public APIs

* Gebruik altijd commentaar bij endpoints en publiek benaderbare APIs.

Korte samenvatting XML-commentaar

* Typ drie keer een slash (dus ///) gevolgd door ENTER om een commentaarblok te genreren. Een voorbeeld met uitgebreide voorbeeldcode en verwijzing:

**Commentaar**

[?](http://confluence.dnbad.nl/display/SD/09.+Commentaar)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | /// <summary>  /// Dit is een korte samenvatting van de methode, call, etc.  /// </summary>  /// <param name="datadefinitie"></param>  /// <example>  /// This sample shows how to call the <see cref="Insert"/> method.  /// <code>  /// class TestClass  /// {  ///     static void Main()  ///     {  ///         Insert(datadefinitie);  ///     }  /// }  /// </code>  /// </example>  public override void Insert(Datadefinitie datadefinitie)  { /\* code here \*/ } |

# [Coding Conventions](http://confluence.dnbad.nl/display/SD/10.+Coding+Conventions)

# Algemeen

* In plaats van string concatenatie met "+" gebruik String.format()om teksten te maken uit vaste tekst gecombineerd met variabelen.
* StringBuilder kan gebruikt worden als de string niet in 1 statement gecontacteerd kan worden.
* Gebruik bij het vergelijken van strings: string.Equals(a, b, StringComparison.CurrentCultureIgnoreCase);
* Gebruik ToString niet voor andere doeleinde dan logging en debugging.
* Implementeer ToString bij alle value objecten en ook bij andere objecten als het toegevoegde waarde heeft. Voer vooral geen equals van objecten uit d.m.v. de ToString te vergelijken.

## Methods

* Methods moeten klein zijn, niet groter dan 20 regels.
* Meerdere return zijn toegestaan, maar dit moet wel met mate blijven. Vaak is het duidelijker als er snel een return plaatsvindt indien niet aan een bepaalde voorwaarde wordt voldaan. Maar als er op 10 plekken gereturned wordt is de methode niet overzichtelijk meer te noemen. (Zou niet voor moeten komen als methodes klein zijn).
* Een methode zou maar 1 ding moeten doen, dus niet het lezen en verwerken van een file, maar 1 voor het lezen en 1 voor het verwerken. en 1 om de 2 calls naar de anderen te maken. Om dit te bevorderen kan een methode het beste of code bevatten of calls naar andere methodes bevatten, maar dit niet mixen.
* Gebruik zo min mogelijk argumenten, 3 is de max. Pas specifieke classes toe indien meer argumenten vereist zijn.
* Het re-assignen van parameters is niet toegestaan.
* In het algemeen is het erg lelijk om booleans als argument mee te geven. De lezer wordt gedwongen om te kijken waar de true of false voorstaat.

## Security

Neem de volgende guidelines in acht van “The Open Web Application Security Project”:

<http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf>

# [Coding Tools](http://confluence.dnbad.nl/display/SD/11.+Coding+Tools)

## StyleCop

StyleCop is een open source static code analysis tool van Microsoft dat C# code controleerd volgens de Microsoft's .NET Framework Design Guidelines. De regels van StyleCop zijn onder te verdelen in de volgende categorieën:

* Documentation
* Layout
* Maintainability
* Naming
* Ordering
* Readability
* Spacing

StyleCop zoekt automatisch naar settings file in bovenliggende folders. Het is dus handig om een file te maken en deze in de root folder van de sources te zetten. Verder kun je deze package gebruiken voor het uitvoeren van StyleCop tijdens de build. Dit moet per project worden geconfigueerd.

Install deze package:

**install-package stylecop.msbuild**

Als je wilt dat je build breekt wanneer er een overtreding wordt began (ipv een warning) kan je in je project file (door in Visual Studio je project te unloaden) het volgende element in de eerste *PropertyGroup* toe te voegen:

*<StyleCopTreatErrorsAsWarnings>false</StyleCopTreatErrorsAsWarnings>*

Vervolgens kan je het project weer reloaden.

## ReSharper

Resharper maak het mogelijk om code te controleren terwijl je deze aan het bewerken bent. Er wordt gecontroleerd of alle code die geschreven is ook wel echt gebruikt wordt. Resharper maakt het ook mogelijk om sneller code aan te passen. Zo geeft Resharper door middel van een gekleurde streep onder de aan te passen code aan dat de code ook anders opgesteld kan worden. Er verschijnt ook een lampje met daarbij een voorstel hoe de code anders kan zijn of hoe een fout opgelost kan worden. Verder bevat Resharper veel extra handige functionaliteit waarmee je sneller en beter kunnen ontwikkelen.

Download: <http://www.jetbrains.com/resharper/>

## GhostDoc

GhostDoc is een extentie die automatisch XML comments genereerd for methods en properties gebaseerd op het type, parameters, naam en andere context informatie.

Download: <http://submain.com/products/ghostdoc.aspx>

## Code Analysis

Code analysis analyseert assembly code volgens de design rules van Microsoft .NET Framework Design Guidelines. De checks worden geoutput als warning messages binnen Visual Studio en toont informatie hoe deze te verhelpen.

Download: Zit standaard in Visual Studio

## Code Metrics

Met Code metrics krijg je beter inzicht in de code die je ontwikkelt. Door gebruik te maken van Code metrics kan je begrijpen welke types en / of methoden moeten worden herwerkt. Je kunt potentiële risico’s identificeren en ook zo huidige status / voortgang van een project inzien.

Download: Zit standaard in Visual Studio

## Log4Net

Log4Net  is een logging framework waarmee de logs naar verschillende soorten outputs kan worden weggeschreven.

Download: <http://logging.apache.org/log4net/>

## Common.Logging

Met Common.Logging library wordt een extra abstractie laag gecreëerd tussen de applicatie en het logging framework.

Download: <http://netcommon.sourceforge.net/>

## Moq

Moq is een mockling library waarmee eenvoudig dummy instanties van een class worden gemaakt zodat classes eenvoudig zijn te unittesten.. Moq is type-safe, refactoring-friendly en support mocking interfaces en classes.

Download: <https://github.com/Moq/>

## Fluent Assertions

Fluent Assertions is een extentie waarmee je makkelijk en bevat een set extensions methods die het doen van assertions in unittesten eenvoudiger maken..

Download: <https://github.com/dennisdoomen/FluentAssertions>

## Ninject

Ninject is een dependency injector waarmee je makkelijk losse stukken aan elkaar kunt ‘lijmen’ en een optimale ontkoppeling te creëren tussen de verschillende lagen van de applicatie.

Download: [www.ninject.org](http://www.ninject.org/)

## Automapper

AutoMapper is een simpel en kleine library om objecten automatisch te mappen naar andere objecten. Ook kan er gebruik worden gemaakt van custom mappings, mochten ze niet je objecten niet 1 op 1 overeenkomen.

Download: <https://github.com/AutoMapper/AutoMapper>

## SafeNuGet

SafeNuGet is een package die de gekoppelde andere packages checked aan de hand van een blacklist.

Meer informatie: <https://www.owasp.org/index.php/OWASP_SafeNuGet>

# [Code Review](http://confluence.dnbad.nl/display/SD/13.+Code+Review)

Voor het laten uitvoeren van collegiale reviews code kan gebruik worden gemaakt van standaard TFS functionaliteit:

<http://www.visualstudio.com/en-us/get-started/get-code-reviewed-vs.aspx>

# [Branching](http://confluence.dnbad.nl/display/SD/14.+Branching)