MEF Compositie Framework

Uitbreidbare applicaties bouwen

Nico De Greef en Tim Cools

Microsoft heeft een nieuw compositie framework in de steigers staan: Microsoft Extensibility Framework. MEF wordt gebruikt bij de bouw van uitbreidbare applicaties. Een uitbreidbare applicatie koppelt zich tijdens de uitvoer aan de beschikbare compatible componenten. Deze componenten worden op hun beurt gekoppeld aan hun dependencies. MEF heeft status preview 5, maar er zijn al veel aanhangers. Het belooft zich te ontpoppen tot een simpele en eenvoudige manier om applicaties te ondersteunen die een plug-in model vereisen. Attributen plaatsen, de Compose methode aanroepen en klaar, de koppelingen liggen vast.

Vraag en aanbod is het basisprincipe van MEF. De container is de verzameling van alle componenten die gebruikt worden tijdens de compositie, ook wel parts genoemd. Deze zijn onder te verdelen in twee groepen: import en export. Een export plaatst een object in de container. Een import haalt een object uit de container. Interpreteer de term object in de ruimste zin van het woord. MEF kan niet alleen types uitwisselen, maar ook properties, fields en methods (delegates).

Eenvoudig voorbeeld

Code voorbeeld 1 toont een koppeling tussen een applicatie en een plug-in via de MEF compositie container. Het plug-in member field van de applicatie zal via MEF een instantie van de plug-in toegewezen krijgen. De toewijzing gebeurt bij aanroep van de compose methode op de container.

```
internal class MainApplication
{
    public void GetPlug-in()
    {
        // Start MEF stuff
        TypeCatalog catalog =
            new TypeCatalog(typeof(Plug-
in));
        CompositionContainer container =
            new
CompositionContainer(catalog);
        container.ComposeParts(this);

        Console.WriteLine("Value via MEF:
{0}",
            _myImportedPlug-in.CoolValue);
        Console.ReadLine();
    }

    [Import("MyCoolPlug-in")]
    private Plug-in _myImportedPlug-in;
}

[Export("MyCoolPlug-in")]
internal class Plug-in
{
    public int CoolValue
    {
        get { return 245; }
    }
}
```

CODEVOORBEELD 1: EENVOUDIG MEF PROGRAMMA.

Het export attribuut op de klasse bepaalt dat deze in de container gekend is en beschikbaar voor imports. Het import attribuut op het member field bepaalt dat de inhoud uit de container komt.
De contractnaam, gedefinieerd in de attributen, is een string die Plug-in en _myImportedPlug-in aan elkaar koppelt. Een string is één van de vele mogelijkheden om het koppelingscontract te bepalen. Een string koppeling dwingt echter geen verificatie af tijdens het ontwikkelen, een tikfout is snel gemaakt. De best practices tonen een betere manier om koppelingscontracten te bepalen.
Dit eenvoudige voorbeeld maakt gebruik van het 'Attributed Programming Model'. Tabel 1 geeft een overzicht van de mogelijke attributen.

ImportAttribute	Importeert één object uit de container
ImportManyAttribute	Importeert nul of meerdere objecten uit de container, gebruikt op collecties
ExportAttribute	Exporteert een object in de container

TABEL 1: OVERZICHT VAN DE BASIS VAN HET ATTRIBUTED PROGRAMMING MODEL.

Compositie

Compositie is het sleutelwoord in MEF. De compositie fase bouwt de container op. Er zijn verschillende mogelijkheden om een compositie te realiseren.

Dynamische compositie

Bij de start van de applicatie alle parts één voor één toevoegen aan de compositie container is onbegonnen werk. Daarom is het mogelijk met catalogs te werken. Een catalog verzamelt parts en voegt deze toe aan

de compositie container. Geef aan de constructor van de compositie container een catalog mee en deze distilleert alle parts uit de catalog die importers en exporters gedefinieerd hebben.

```
AssemblyCatalog catalog =
    new AssemblyCatalog(Assembly.GetEn-
tryAssembly());
CompositionContainer container =
    new CompositionContainer(catalog);
```

CODEVOORBEELD 2: GEBRUIK VAN EEN ASSEMBLY-CATALOG.

Code voorbeeld 2 bouwt een catalog op gebaseerd op de AssemblyCatalog, deze wordt meegegeven aan de constructor van de container. De AssemblyCatalog gaat op zoek gaat naar import en export attributen binnen de gekozen assembly.

AggregateCatalog	Collectie van andere catalogs
AssemblyCatalog	Alle parts binnen een assembly
DirectoryCatalog	Alle parts van alle assemblies binnen eenzelfde folder (niet recursief)
TypeCatalog	Alle parts binnen de meegegeven type lijst

TABEL 2: DE MEEGELEVERDE CATALOGS.

Standaard levert MEF een aantal catalogs mee. Tabel 2 somt de ingebouwde catalog types op. Het is uiteraard mogelijk om een custom catalog te bouwen door over te erven van ComposablePartCatalog. Het is niet nodig om onmiddellijk deze stap te nemen wanneer de meegeleverde part catalogs niet voldoen. De AggregateCatalog kan ook gebruikt worden om bestaande catalogs aan elkaar te koppelen tot één grote catalog. Om een recursieve DirectoryCatalog te bouwen, is het voldoende een aantal DirectoryCatalogs aan elkaar te koppelen met de hulp van deze AggregateCatalog. Voor elke subfolder één DirectoryCatalog. Een link naar een RecursiveDirectoryCatalog staat in de referenties.

Batch compositie

De container in MEF is een dynamische container, er kunnen altijd compositie parts en export definities aan toegevoegd en uit verwijderd worden. Na het uitvoeren van de Compose methode van de container, worden alle imports opnieuw geëvalueerd. Deze wijzigingen worden gegroepeerd in een CompositionBatch. Deze houdt bij wat er in de container wijzigt. Het volgende voorbeeld toont de samenstelling van een batch. De AddPart methode voegt objecten aan de batch toe. De Compose methode koppelt de imports van deze objecten aan de overeenkomstige exports. De AddExportedObject methode voegt object instanties toe die als imports worden gebruikt. Let op dat het toevoegen van de objecten met de AddExportedObject methode de CreationPolicy parameters van de imports overschrijven. Al de imports zullen naar dit object wijzen, ongeacht ze op Shared of NonShared staan. CreationPolicy wordt in detail uitgelegd in het topic object lifetime.

```
CompositionBatch batch = new Composition-
Batch();
batch.AddPart(aPart);
ComposablePart controllerPart = batch.AddE
xportedObject(anObjectInstance);

container.Compose(batch);
```

CODEVOORBEELD 3: GEBRUIK VAN EEN COMPOSITION-BATCH.

Indien een bepaalde part niet meer gewenst is, kan deze verwijderd worden met de RemovePart methode van de batch. Dit zal er voor zorgen dat de imports die daarvoor het object gebruikten niet meer naar het object wijzen, maar deze zullen opnieuw gekoppeld worden aan de hand van overgebleven parts in de catalog.

```
CompositionBatch batch = new Composition-
Batch();
batch.RemovePart(controllerPart);

container.Compose(batch);
```

CODEVOORBEELD 4: EEN PART VERWIJDEREN UIT DE CONTAINER.

Indien het gewenst is dat de imports opnieuw geëvalueerd worden bij achtereenvolgende Compose aanroepen, dan moet de AllowRecomposition property van het ImportAttribute op true gezet worden. Het is ook aangewezen de AllowDefault property op true te zetten. Hierdoor zal de Compose niet falen als er voor een bepaalde import geen export gevonden wordt.

```
[Import(typeof(Controller),

RequiredCreationPolicy = CreationPolicy.
NonShared,

AllowRecomposition = true, AllowDefault =
true )]
public Controller Controller
{
    get; set;
}
```

CODEVOORBEELD 5: HERCOMPOSEERBARE IMPORT PROPERTY.

Manuele compositie

Sinds preview 5 kan men rechtstreeks parts aan de container toevoegen met de ComposeParts methode. Deze methode koppelt zowel imports als exports. Een CompositionBatch aanmaken is niet meer noodzakelijk om parts toe te voegen.
De SatisfyImports methode van de container vult de imports van een bestaande object instantie aan. De object instantie wordt meegegeven als parameter aan deze methode. De instantie wordt dan niet aan de container toegevoegd, de container wordt enkel gebruikt om de aangeleverde instantie op te vullen.
MEF zet intern de import en export attributen om naar ImportDefinitions en ExportDefinitions. Deze vormen de basis van MEF. Een ExportDefinition is de definitie van een object dat ter beschikking gesteld wordt aan de container. ContractName en Metadata bepalen hier de zoekcriteria waarmee deze export kan opgevraagd worden. Een ImportDefinition is de definitie van een query aan de container. Deze bevat de cardinaliteit en de constraint. De cardinaliteit bepaalt hoeveel instanties de importer kan ontvangen, de constraint is een lambda expressie die een export definitie ondervraagt. Code voorbeeld 6 toont hoe de MEF container zonder attributen ondervraagd wordt.

```
ImportDefinition importDefinition =
    new ImportDefinition(
        (expdef) => (expdef.ContractName.
StartsWith("MyCoolValue")),

ImportCardinality.ZeroOrMore, true, fal-
se);
IList<Export> exports = new
List<Export>(container.
GetExports(importDefinition));
```

CODEVOORBEELD 6: MEF ONDERVRAGEN ZONDER ATTRIBUTEN.

Een importdefinitie vervangt een import attribuut. De lambda expressie neemt alle exports uit de container waarvan de contractnaam begint met 'MyCoolValue'. Een export is nog geen instantie, het is een proxy. De methode Export.GetExportedObject maakt, afhankelijk van de object lifetime, een instantie. Een werkend voorbeeld, ConsoleApplicationManual, staat in de referenties.

Object lifetime

De lifetime van de parts binnen de container is bepaald door de CreationPolicy enumeratie. Deze policy kan gezet worden op Imports en Exports. De Shared optie zorgt ervoor dat, conform het singleton pattern,

er slechts één instantie gecreëerd wordt per container. De NonShared optie zorgt ervoor dat de container bij elke aanvraag of import, een nieuwe instantie creëert, analoog het factory pattern.
Om de creation policy van een export te definiëren dient de Export met het CompositionOptions attribuut gedecoreerd te worden.

```
[Export(typeof(IService))]
[PartCreationPolicy(CreationPolicy.NonShared)]
internal class Service
{
    // ...
}
```

CODEVOORBEELD 7: CREATE POLICY BEPALEN VAN EEN EXPORT.

Om de creation policy van een import te definiëren volstaat het om het RequiredCreationPolicy property van het Import attribuut te specifiëren.

```
[Import(typeof(IController), RequiredCreationPolicy = CreationPolicy.NonShared)]
public Controller Controller
{
    get; set;
}
```

CODEVOORBEELD 8: CREATE POLICY BEPALEN VAN EEN IMPORT.

Standaard staat de CreationPolicy op Any. Dit komt er op neer dat de parts geshared zijn, uitgezonderd als een Import of Export expliciet bepaald dat het niet geshared moet worden. In tabel 3 kan je zien wat het resultaat is van de mogelijke combinaties.

Best practices

Contract by interface
Zoals in codevoorbeeld 1 al aangehaald zijn er naast strings betere manieren om een contract te bepalen.
Indien het Export en Import attribuut geen contract parameter heeft, baseert MEF zich op het datatype van de te koppelen componenten. Voor zelf geschreven types is dit geen probleem, deze zijn uniek genoeg. Echter een CLR data type zoals Int32 stelt wel een probleem, immers alle componenten van het type Int32 aan elkaar koppelen is doorgaans niet de bedoeling.
De meest gebruikte benadering is om aan het Import en Export attribuut het contract mee te geven onder de vorm van een eigen type. Dit zorgt voor design-time verificatie van de contracten. Het contract type is een parameter van het import of export attribuut, onder de vorm [Import(typeof(IMyCoolType))]. Intern in MEF wordt dit terug naar een string omgezet. Deze string is behoudens uitzondering (ContractTypeAttribute), de na-

	Part.Any	Part.Shared	Part.NonShared
Import.Any	Shared	Shared	NonShared
Import.Shared	Shared	Shared	N/A
Import.NonShared	NonShared	N/A	NonShared

TABEL 3: CREATE POLICY COMBINATIES.

mespace en de naam van het type.
Dit is niet het enige voordeel. In code voorbeeld 9 wordt elke concrete implementatie van IMetricConvertor geëxporteerd als typeof(IMetricConvertor). Een importer van IMetricConvertor krijgt zo al deze concrete implementaties in een collectie van IMetricConvertor. IMetricConvertor is de basis voor de plug-ins. De interface schrijft de Convert methode voor. Elke concrete MetricConvertor implementeert deze interface. MEF zorgt voor de eventuele type conversies. Een volledige demo applicatie is beschikbaar bij de referenties.

Metadata biedt de mogelijkheid om specificaties over een object bij te houden

Typed metadata
Metadata biedt de mogelijkheid om specificaties over een object bij te houden. Deze specificaties kunnen opgevraagd worden zonder het object te instantiëren. Metadata kan in een ImportDefinition ook gebruikt worden om te filteren. Objecten uit de container halen gebeurt dan op basis van contractnaam en metadata specificaties.
De eenvoudigste benadering is string-based metadata, een klassieke key-value collectie. Nadeel van deze benadering is dat Visual Studio de string metadata niet controleert tijdens het ontwikkelen. Fouten komen pas naar boven tijdens het runnen.
Een alternatief, eerder een must, is typed metadata. Deze vorm van metadata wordt gecontroleerd tijdens het ontwikkelen. Code voorbeeld 9 toont het gebruik van typed metadata.

```
[MetricConvertorMetadata(SourceMetric =
"M", DestinationMetric = "CM")]
[Export(typeof(IMetricConvertor))]
internal class MetricConvertorMeterToCen-
timer: IMetricConvertor
{
    #region IMetricConvertor Members

    public decimal Convert(decimal source-
Value)
    {
        return sourceValue * 100;
    }

    #endregion
 }
```

CODEVOORBEELD 9: IMPLEMENTATIE MET METADATA.

ConsoleApplicationMetaData is een uitgewerkt voorbeeld (zie referenties). Deze applicatie gebruikt een plug-in model om lengtematen te converteren. Interface IMetricConvertorMetadata bepaalt de properties voor de metadata. MetricConvertorMetadataAttribute is de concrete implementatie van het custom metadata attribuut en implementeert de interface. Dit definieert de typed metadata voor de MetricConvertor.

Elke convertor wordt gedecoreerd met het MetricConvertorMetadataAttribute, dat aangeeft tussen welke lengtematen de conversie loopt. Bijvoorbeeld [MetricConvertorMetadataAttribute(SourceMetric = "M", DestinationMetric = "CM")] duidt aan dat de MetricConvertor van meter naar centimeter converteert.

```
// Proxy objects, no instances created yet
// Note: We use IMetricConvertorMetadata
instead of the attribute itself. This in-
terface is required and has to be public
ExportCollection<IMetricConvertor, IMe-
tricConvertorMetadata> convertors = con-
tainer.GetExports<IMetricConvertor, IMe-
tricConvertorMetadata>();

// Note: We removed the meta data attri-
bute from MetricConvertorMeterToMilimeter,
and it does not show up.
// The use of ExportCollection<T, TMeta-
Data> makes the MetaData attribute re-
quired, exports without this attribute are
ignored

// Retrieve the proxy object based on the
TYPED metadata (could have done this in
one step while calling GetExports<>() and
passing a lambda expression)
Export<IMetricConvertor, IMetricConvertor-
Metadata> neededConvertor =
    (from convertor
     in convertors
     where convertor.MetadataView.Source-
Metric == sourceMetric.ToUpper()
     && convertor.MetadataView.Destinati-
onMetric ==
        destinationMetric.ToUpper()
    select convertor)
    .SingleOrDefault<Export<IMetricConvert
or, IMetricConvertorMetadata>>();
```

CODEVOORBEELD 10: TYPED METADATA VOORBEELD.

De specificatie van elke MetricConvertor is het contract IMetricConvertor en de metadata MetricConvertorMetadataAttribute. Aan de import zijde, het eigenlijke conversie programma (code voorbeeld 10), is dit de basis om de juiste convertor te kiezen.
De ExportCollection<IMetricConvertor, IMetricConvertorMetaData> geeft aan dat er enkel interesse is in de convertors die van de metadata specificaties voorzien zijn. Impliciet wordt de aanwezigheid van de juiste metadata een requirement.

Lazy loading

Het standaard gedrag bij een compose is dat het gevraagde object geïnstantieerd wordt en aan de import toegekend. Bij objecten die te intensief zijn om te instantiëren heeft dit tot gevolg dat het composen van de container een zware operatie is. Soms is het gewenst om lazy loading toe te passen op de geïmporteerde objecten. Dit houdt in dat een object pas geïnstantieerd wordt bij het eerste gebruik. Hiervoor stelt MEF Export<T> objecten ter beschikking waarbij T het type is van het geïmporteerde object.

```
[Import(typeof(IService))]
public Export<Service> ServiceExport
{
    get; set;
}
```

CODEVOORBEELD 11: LAZY LOADED PROPERTY.

Om het object dan op te vragen wordt de GetExportedObject methode aangeroepen. De eerste aanroep instantieert het object. Daaropvolgende aanroepen geven het zelfde object terug.

```
Service service = ServiceExport.GetExpor-
tedObject();
```

CODEVOORBEELD 12: GET EXPORT OBJECT INSTANTIE.

Het is aangewezen om de Export te encapsuleren en enkel het gewenste object publiek beschikbaar te stellen. Hiervoor dien je de export te importeren via een private variabele en een publieke property te maken die het object zelf aanbiedt.

```
[Import(typeof(IService))]
private Export<Service> serviceExport;

public Service Service
{
    get
    {
        return serviceExport.GetExported-
Object();
    }
}
```

CODEVOORBEELD 13: GEËNCAPSULEERDE LAZY LOADED PROPERTY.

Lazy loading is niet enkel mogelijk bij object instanties maar ook bij collecties. Deze collectie zal dan opgevuld worden met Export objecten. Elk object kan dan gebruikt worden om een instantie aan te maken. Elk object in de collectie wordt afzonderlijk geïnstantieerd op het moment dat het voor de eerste keer gebruikt wordt.

```
[ImportMany(typeof(IService))]
public ExportCollection<Service> Childs
```

```
{
    get; private set;
}
```

CODEVOORBEELD 14: LAZY LOADED COLLECTION PROPERTY.

Delegate import

Buiten het importeren van waardes en objecten biedt MEF de mogelijkheid om methodes te importeren. Hiervoor dienen de import en export attributen boven een delegate field of property geplaatst te worden. Deze worden dan automatisch gekoppeld bij een compose. Gebruik dit principe om events los te koppelen van hun subscriber. Voorbeeld 15 toont hoe een event gedeeld wordt door publisher en subscriber.

```
public class Publisher
{
    [Import("Events.Plug-inReset")]
    private EventHandler _plug-inResetSubscriber;

    internal void Reset()
    {
        // do some logic
        // ....

        // notify subscribers that this plug-in is reset
        _plug-inResetSubscriber(this, EventArgs.Empty);
    }
}

public class Subscriber
{
    [Export("Events.Plug-inReset")]
    private EventHandler _plug-inReset = Plug-inReset;


static void Plug-inReset(object sender, EventArgs e)
    {

Log.Write("Plug-in '{0}' is reset", sender.GetType());
    }
}
```

CODEVOORBEELD 15: DELEGATE PUBLISHER MET ÉÉN SUBSCRIBER.

Het nadeel van voorgaande implementatie is dat elk event slechts één subcriber kan hebben. Een collectie van delegates lost dit op. Elke delegate in de collectie wordt dan individueel aangeroepen.

```
public class MultiPublisher
{
    [Import("Events.Plug-inReset")]
    private IList<EventHandler> _plug-inResetSubscribers;

    internal void Reset()
    {
        // do some logic
        // ....

        // notify subscribers that this plug-in is reset
        foreach (var eventHandler in _plug-inResetSubscribers)
        {
            eventHandler(this, EventArgs.Empty);
        }
    }
}
```

CODEVOORBEELD 16: DELEGATE PUBLISHER MET MEERDERE SUBSCRIBERS.

Pitfalls

Bad plug-in handling

Een groot voordeel van MEF is de mogelijkheid om gebruikers van een applicatie de mogelijkheid te geven zelf geschreven plug-ins te gebruiken. Hier dient men wel voorzichtig mee om te gaan, we kunnen er immers nooit zeker van zijn dat deze plug-ins volledig correct werken in alle omstandigheden. Wanneer de MEF container bijvoorbeeld geen part kan vinden voor een

bepaalde import van een plug-in, dan throwt hij standaard een ImportCardinalityMismatchException bij de compose. Bij verkeerd gebruik van MEF kan dit ervoor zorgen dat geen enkele plug-in geladen wordt. Om dit te vermijden kunnen enkele maatregelen genomen worden. Ten eerste, handel de import van de plug-in collectie af met een Export<T> collectie. Dit geeft de mogelijkheid om elke plug-in afzonderlijk te instantiëren. Als het instantiëren van een enkele plug-in faalt, zal de rest van de plug-in's nog laden. Vervolgens is het aangewezen om imports te definiëren met de AllowDefault property op true. Dit zorgt ervoor dat deze import null is, in plaats van het throwen van een exceptie als de plug-in niet linkt.

De import van een property is een publieke setter. Het maakt de objecten dus wijzigbaar

Een schoentje dat altijd past is een ExportCollection<YourType>. Een compose lukt dan altijd, ongeacht het aantal: geen, één of meerdere. Dit vermijdt zelfs instantiatie fouten.

```
public class Host
{
    private readonly IList<IPlug-in> _loa-
dedPlug-ins;

    [ImportMany(typeof(IPlug-in))]
    public ExportCollection<IPlug-in>
Plug-ins
    {
        get;
        private set;
    }

    [Import(AllowDefault = true)]
    public IConnectionService Connection-
Service
    {
        get;
        private set;
    }

    public Host()
    {
        _loadedPlug-ins = new List<IPlug-
in>();
    }

    public void LoadPlug-ins()
    {
        foreach (var plug-in in Plug-ins)
        {
            try
            {

_loadedPlug-ins.Add(plug-in.GetExportedOb-
ject());
            }
            catch (CompositionException
ex)
            {

Log.Write(ex, "Failed to load plug-in
{0}",
                      plug-in.Definition.
ContractName);
            }
        }
    }
}
```

CODEVOORBEELD 17: BAD PLUG-IN HANDLING.

ImportDefinition	Query definitie om een object uit de container te halen
ExportDefinition	Contractnaam en metadata van een object in de container
Part	Collectie van ImportDefinitions en ExportDefinitions van een type. Bijvoorbeeld een klasse kan meerdere ImportDefinitions en ExportDefinitions bevatten indien op verscheidene properties attributen staan.
Catalog	Collectie van parts en discovery routines voor parts
Export	Proxy voor een object instantie in de container
ExportCollection	Collectie van Export proxies.

TABLE 4: BELANGRIJKSTE MEF TERMEN.

Private import

Logischerwijze is de import van een property een publieke setter. Voorzichtigheid is geboden, het maakt de objecten wijzigbaar door de consumers. MEF ondersteunt standaard het importeren via private property setters, de constructor of private fields. Private setters zorgen ervoor dat enkel MEF de imports kan wijzigen.

```
[Export]
class Plug-in
{
    [Import(typeof(IService))]
    private Service service;

    public Controller Controller
    {
        get;
        private set;
    }

    [Import]
    public Exporter Exporter
    {
        get;
        private set;
    }

    [ImportingConstructor]
    public Plug-in(Controller controller)
    {
        Controller = controller;
    }
}
```

CODEVOORBEELD 18: PRIVATE IMPORTS.

Het importeren via de constructor heeft als bijkomend voordeel dat de imports al beschikbaar zijn binnen de constructor. Dit in tegenstelling tot een import via een property of een veld. Sinds preview 5 is er een IPartImportsSatisfiedNotification interface beschikbaar die een OnImportsSatisfied methode voorziet. De container roept deze methode aan zodra de compositie afgerond is. Dit is het moment om veilig de imports te gebruiken.

Conclusie

MEF biedt de mogelijkheid om op een heel eenvoudige manier uitbreidbare applicaties of frameworks te ontwikkelen. In deze eenvoud schuilt de kracht van dit compositie framework. Initieel krijg je het gevoel dat je een gebrek aan controle hebt, maar eenmaal deze drempel overwonnen, lacht de wereld van MEF je toe.
Om af te sluiten toont tabel 4 een overzicht van de belangrijkste MEF termen.

Links

Article Source Code: http://www.nicodegreef.net/downloads/2009ArticleMef.zip of http://www.timcools.net/downloads/2009ArticleMef.zip

Creating a custom catalog: http://codebetter.com/blogs/glenn.block/archive/2009/04/03/creating-a-functional-programming-model-for-mef.aspx

Tim Cools en Nico De Greef, werken als consultant in het .NET kernel team van B-Holding ICTRA. Zij zijn bereikbaar via nico.degreef@b-holding.be en timotheus.cools@b-holding.be. Hun blogs, over topics die aan bod komen in het kernel team, zijn beschikbaar via timcools.net en nicodegreef.net.