

Delegates in Visual Basic.NET

door Alex Thissen

Met de komst van Visual Basic.NET wordt een VB programmeur geconfronteerd met een aantal nieuwe concepten. Naast de vernieuwing in de syntax van de taal heeft VB.NET er een tweetal primitieven erbij gekregen: de delegate en de class. In dit artikel zult u kennismaken met de delegate.

Visual Basic.NET introduceert de delegate als nieuw type voor de Visual Basic programmeur. Voor de meesten van u zal het de eerste kennismaking zijn met delegates. Aan de gedachte achter delegates zult u even moeten wennen. In dit artikel zult u leren wat delegates zijn en waarvoor ze gebruikt kunnen worden.

Delegates zijn objecten met daarin een verwijzing naar een methode. Een dergelijke methode kan deel uitmaken van een Visual Basic module of klasse. C en C++ kennen al langer begrippen als functiepunters die erg overeenkomen met delegates. Echter, in de Common Language Runtime kunnen functiepunters (wat in feite de geheugenadressen van de betreffende methoden zijn) op zich niet bestaan omdat alles een type moet zijn. Vandaar dat de delegate geïntroduceerd is: een CLR type waarvan de objecten pointers naar methoden bevatten. Met een dergelijk delegate object is het mogelijk om de aanroep naar de methode waar de delegate naar verwijst te maken, zonder dat we verder iets hoeven te weten van die methode. Bovendien kunnen we delegate objecten doorgeven binnen onze programmacode.

De rest van dit artikel zal proberen om bovenstaande beter uit te leggen en u het nut van delegates te laten inzien. Dit zal gebeuren aan de hand van een voorbeeld waarin op verschillende manieren gebruik gemaakt wordt van delegates. Het scenario betreft een persoon die geld wil pinnen bij een pinautomaat. De persoon wordt vertegenwoordigd door een klasse Pinner die gebruik zal gaan maken van een klasse ChipperAutomaat. In eerste instantie bekijken we een oplossing zonder daarbij gebruik te maken van delegates. Vervolgens zult u zien hoeveel eleganter het is om in deze situatie delegates te gebruiken. Tenslotte krijgt u te zien hoe delegates te gebruiken zijn om methoden behalve synchroon ook asynchroon aan te roepen.

De klasse Pinner (de persoon die gaat pinnen) ziet er in eerste instantie als volgt uit:

```
Public Class Pinner
    Public Sub DoePinVerzoek(ByVal objChipper As _
        ChipperAutomaat, ByVal Bedrag As Single) _
        Dim blnGelukt As Boolean
        Dim sngMaximum As Single
        blnGelukt = objChipper.Chippen(Bedrag, sngMaximum)
        Dim strGelukt = IIf(blnGelukt, "", "niet ")
        Console.WriteLine("Uw pinverzoek is {0}gelukt.", _
            strGelukt)
        If Not blnGelukt Then
            Console.WriteLine("Maximum bedrag is : {0} Euro", _
                sngMaximum)
        End If
    End Function
End Class
```

Zoals u ziet kan met behulp van een Pinner object een pinverzoek gedaan worden door de methode DoePinVerzoek aan te roepen en daarbij een ChipperAutomaat object en het gewenste bedrag mee te geven. De implementatie van de ChipperAutomaat klasse ziet er als volgt uit:

```
Public Class ChipperAutomaat
    Public Function Chippen(ByVal Bedrag As Single, ByRef _
        Maximum As Single) As Boolean
        Console.WriteLine("Verwerken van uw opdracht. Dit kan even duren")
        'Wacht 6 seconden om lange bewerking na te bootsen
        System.Threading.Thread.Sleep(6000)
        If Bedrag > 200 Then
            Maximum = 200
            Return False
        End If
        Return True
    End Function
End Class
```

De Chippen methode ontvangt een te pinnen bedrag en zal controleren of dit bedrag niet groter is dan de limiet van 200 euro en geeft dan de waarde True terug. Is het te pinnen bedrag te groot, dan zal de returnwaarde False zijn. Het pinverzoek is dan afgewezen en zal de waarde van Maximum het maximum toegestane pinbedrag bevatten. Om de communicatietijd na te bootsen is er een pauze van 6 seconden ingebouwd.

Met het volgende stuk code kunt u een pinner 100 euro laten pinnen:

```
Dim objPinner As New Pinner()
Dim objChipperAutomaat As New ChipperAutomaat()
objPinner.DoePinVerzoek(objChipperAutomaat, 100)
```

De bovenstaande aanpak gaat goed zolang een pinner altijd gebruik zal maken van een Chipper automaat. Het wordt lastiger indien u de pinner ook gebruik wilt laten maken van bijvoorbeeld een Postbank pinautomaat, zoals in de volgende klasse PostbankPinAutomaat.

```
Public Class PostbankPinAutomaat
    Public Shared Function GeldOpname(ByVal Bedrag As Single, _
        ByRef Maximum As Single) As Boolean
        Console.WriteLine("Verwerken van uw opdracht bij Postbank. Dit kan even duren")
        'Wacht 6 seconden om lange bewerking na te bootsen
        System.Threading.Thread.Sleep(6000)
        Dim ran As New System.Random()
        Maximum = ran.Next(1000)
        Return Bedrag <= Maximum
    End Function
End Class
```

Zoals u ziet bevat de PostbankPinAutomaat klasse een soortgelijke methode als ChipperAutomaat om een pinverzoek te doen. Het verschil is de implementatie, de naam en het type methode (shared in plaats van instantie methode). De signature van de methode, dat wil zeggen de argumenten en de returnwaarde, zijn wel hetzelfde. Toch kunt aan de methode DoePinVerzoek niet een PostbankPinAutomaat variabele meegeven omdat er expliciet een ChipperAutomaat object verwacht wordt, waarop de Chippen methode moet kunnen worden aangeroepen. In het eerste codefragment is dit dikgedrukt weergegeven. De gekozen oplossing is dus niet zo flexibel.

U heeft nu een aantal mogelijkheden om de code van de Pinner klasse zo aan te passen dat u ook van een PostbankPinAutomaat gebruik kunt maken:

- U schrijft een aparte methode in de Pinner klasse voor ieder type pinautomaat, zoals bijvoorbeeld DoeChipperPinVerzoek en DoePostbankPinVerzoek enzovoorts. Ieder van deze methoden heeft een argument met het corresponderende type pinautomaat. U zult dan iedere keer als er een nieuwe pinautomaat klasse gemaakt wordt de code van de klasse Pinner uit moeten breiden.

Bovendien zullen er net zoveel methoden in Pinner zijn als er pinautomaat klassen zijn. Dit is niet een ideale oplossing.

- Visual Basic is nu volledig object-geïntendeerd. U zou dus een abstracte basisklasse PinAutomaat kunnen schrijven met daarin een Overridable methode Pinnen en vervolgens de concrete klassen ChipperAutomaat en PostbankPinAutomaat hiervan af kunnen leiden met een implementatie voor de Pinnen methode. De methode PinVerzoek uit de Pinner klasse zal dan een argument ByVal objPinAutomaat As PinAutomaat bevatten in plaats van een argument met een concreet type pinautomaat, zoals ChipperAutomaat of PostbankPinAutomaat. Het mechanisme van polymorfisme zorgt ervoor dat de juiste implementatie code in de van PinAutomaat afgeleide klasse wordt aangeroepen. Zie ook listing ***. Op zich is dit een hele nette oplossing, maar u gaat er daarbij wel vanuit dat het mogelijk is om de klassen ChipperAutomaat af te kunnen leiden van een basisklasse. Dit is niet altijd het geval.
- Een andere mogelijkheid is het gebruik van delegates. Dit zal hieronder in detail besproken worden.

In essentie wilt u een methode van de vorm

```
Public Function Functienaam(ByVal Bedrag As Single, ByVal _
    Maximum As Single) As Boolean
```

aanroepen, waarbij Functienaam er in principe niet toe doet. Het zou voldoende zijn om aan de methode DoePinVerzoek een verwijzing naar de juiste methode mee te geven, zonder de details van het object waarin de methode staat en de exacte naam van de methode mee te geven. Dat is nu precies waar delegate objecten voor bedoeld zijn. Een delegate object bevat een verwijzing naar een methode zonder specificatie van type object en methodenaam. Wel is van belang hoe de vorm van de methode is, want u kunt in een delegate object alleen methoden stoppen waarvoor het betreffende delegate type ontworpen is. Wat u nodig heeft in het pinner scenario is een delegate type waarin methodeverwijzingen van bovengenoemde vorm gestopt kunnen worden.

Als een delegate object eenmaal een methodeverwijzing bevat kan dit object doorgegeven worden binnen uw code als iedere andere variabele. Op ieder moment is het mogelijk om met behulp van het delegate object de methode waarnaar verwezen wordt aan te roepen. Hoe dat in zijn werk gaat zult u later zien.

Declaratie van delegate type

Om de code uit het voorbeeld te verbeteren gaat u gebruik maken van delegates. Allereerst dient er een delegate type gedeclareerd te worden waarvan de objecten in staat zijn om verwijzingen naar methoden van de vorm

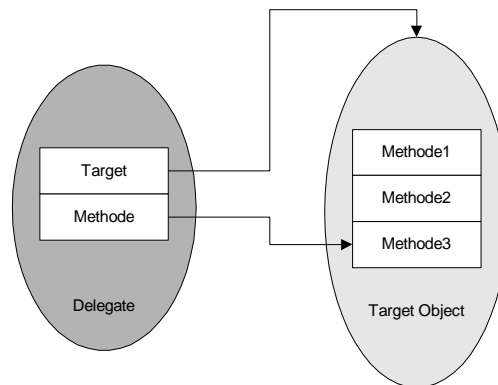
```
Public Function Functienaam(ByVal Bedrag As Single, ByVal _
    Maximum As Single) As Boolean
```

te bevatten. Een dergelijke declaratie ziet er als volgt uit

```
Public Delegate Function DoePinVerzoekDelegate(ByVal Bedrag As
    Single, _
    ByVal Maximum As Single) As Boolean
```

Het toevoegen van het woord delegate is voldoende om van een methode signature een delegate type te maken. De naam van het nieuwe type is wat de naam van de methode lijkt te zijn. In dit geval heet het delegate type dus PinVerzoekDelegate. Zoals u ziet zijn de naam van het type object en de naam van de aan te roepen methode niet vastgelegd bij de declaratie van het delegate type.

Van het type PinVerzoekDelegate kunnen we instanties c.q. objecten maken. Intern ziet een delegate object er uit zoals in onderstaande figuur. Een delegate object bevat een referentie naar het object waarin een methode wordt aangeroepen en een verwijzing naar de juiste methode binnen het object. Uiteraard wordt er verwezen naar een methode die exact de vorm moet hebben zoals die tijdens de declaratie van het delegate type is opgesteld.



Als u een object van een bepaald delegate type wilt maken, hoeft u in de constructor van het delegate type alleen de verwijzing naar de methode binnen een object mee te geven. U gebruikt daarbij de AddressOf operator. Het maken van een instantie van een delegate type ziet er dan als volgt uit:

```
Dim objDelegate As PinVerzoekDelegate
objDelegate = New PinVerzoekDelegate(AddressOf _
    objChipperAutomaat.GewensteMethode)
```

waarbij objChipperAutomaat een object uit de klasse ChipperAutomaat is. Daarnaast is het mogelijk om de methode pointer van een delegate naar één van de statische methoden van een klasse of een methode in een Visual Basic module te laten verwijzen. Uiteraard moet ook deze methoden de juiste signature hebben. Voor statische en module methoden wordt intern de Target van een delegate niet gebruikt. In de constructor gebruikt u in plaats van een object referentie de naam van de klasse of module. De constructie van een delegate object ziet er dan uit als:

```
objDelegate = New PinVerzoekDelegate(AddressOf _
    PostBankPinAutomaat.GeldOpname)
```

of

```
objDelegate = New PinVerzoekDelegate(AddressOf _
    ChipKnipModule.Chipknippen)
```

Aanroepen van delegates

Als u door middel van het delegate object de methode waarnaar verwezen wordt daadwerkelijk wilt aanroepen, dan wordt de Invoke methode van een delegate object gebruikt, in de plaats van de daadwerkelijke methodenaam.

```
blnGelukt = objDelegate.Invoke(100, sngMaximum)
```

De Invoke methode heeft precies dezelfde signature als de target methode. De Invoke methode is de default methode van het delegate object en mag ook weggelaten worden.

```
blnGelukt = objDelegate(100, sngMaximum)
```

Als we het oorspronkelijke codevoorbeeld aanpassen, dan zal de Pinner klasse gebruik gaan maken van een delegate argument in plaats van de ChipperAutomaat argument.

```
Public Class Pinner
    Public Sub DoePinVerzoek(ByVal pvd As _
        PinVerzoekDelegate, ByVal Bedrag As Single)
        Dim blnGelukt As Boolean
        Dim sngMaximum As Single
        blnGelukt = pvd.Invoke(Bedrag, sngMaximum)
        Dim strGelukt = IIf(blnGelukt, "", "niet ")
    End Sub
End Class
```

```

Console.WriteLine("Uw pinverzoek is {0}gelukt.", _
    strGelukt)
If Not blnGelukt Then
    Console.WriteLine("Maximum bedrag is : {0} Euro", _
        sngMaximum)
End If
End Function
End Class

```

De aanroep van de DoePinVerzoek methode zal ook gewijzigd moeten worden tot

```

Dim objPinner As New Pinner()
Dim objAutomaat As New ChipperAutomaat()
Dim objDelegate As New PinVerzoekDelegate(AddressOf _
    objAutomaat.Chippen)
objAutomaat = Nothing
objPinner.DoePinVerzoek(objDelegate, sngBedrag)

```

Misschien maakt u zich zorgen dat het object objAutomaat niet meer zal bestaan op het moment dat de delegate de methode moet gaan uitvoeren. Immers, de referentie objAutomaat wordt op Nothing gesteld. Bij statische methoden van een klasse of een methode in een module loopt u dat risico niet.

Uw eventuele zorgen zijn gelukkig ongegrond. Het feit dat een delegate een referentie naar het object bevat betekent dat het object nooit door de garbage collector opgeruimd zal worden zolang de delegate niet opgeruimd hoeft te worden. De interne referentie van het delegate object houdt het object in leven.

Als u nu in plaats van de ChipperAutomaat de PostbankPinAutomaat klasse wilt gebruiken zal de aanroep er praktisch hetzelfde uitzien (volgend op het vorige codefragment).

```

objDelegate As New PinVerzoekDelegate(AddressOf _
    PostbankPinAutomaat.GeldOpname)
objPinner.DoePinVerzoek(objDelegate, sngBedrag)

```

Omdat de GeldOpname methode een statische methode van de klasse PostbankPinAutomaat is, hoeft u geen object voor de pinautomaat aan te maken. Merk op dat er nu niets is veranderd aan de code van DoePinVerzoek in de klasse Pinner. Dat was waar het onder andere om was te doen.

Interne details van een delegate

Wanneer u een delegate declareert

```
Public Delegate Sub EenDelegate(ByVal Bedrag As Single)
```

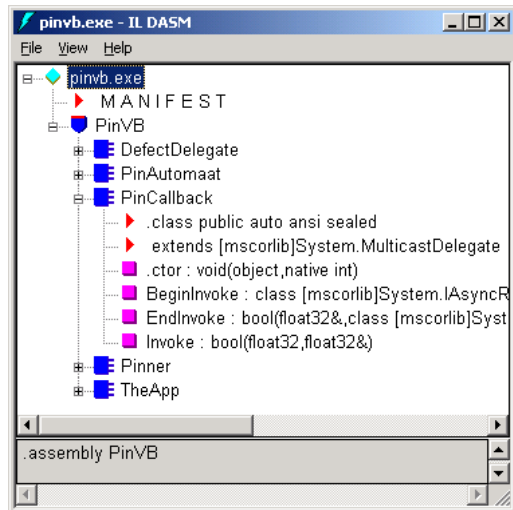
zal de compiler hiervoor code genereren die grofweg hiermee overeenkomt (in pseudo-code):

```

Public Class SomeDelegate
    Inherits System.MulticastDelegate
    Public Sub New(ByVal obj As Object, ByVal mref As MethodRef)
        Base(obj, mref)
    ...
    End Sub
    Public Sub Overridable Invoke(ByVal val As Single)
    ...
    End Sub
    ...
End Class

```

Wanneer u een assembly opent met behulp van de IL Disassembler tool (ildasm.exe) zult u zien dat de gedeclareerde delegate types in principe geen primitieve types van de CLR zijn. Door de VB compiler zal voor ieder delegate type een class gecompileerd worden die is afgeleid van MulticastDelegate. Deze class bevat een non-default constructor en een drietal methoden, te weten Invoke, BeginInvoke en EndInvoke. Zie ook de volgende figuur.



De methode Invoke heeft u al voorbij zien komen. De Begin- en EndInvoke komen verderop aan de orde.

Wanneer u een delegate aanroept door middel van Invoke doet zult u merken dat dergelijke calls synchroon verlopen. De thread waarop de code van de aanroep (in dit geval de code binnen DoePinVerzoek) draait zal ook de code binnen de delegate methode (Chippen c.q. GeldOpname) uitvoeren op een synchrone manier. Dit betekent dat in ons voorbeeld bij een pinopdracht naar de chipperautomaat er een zestal seconden overheen zal gaan voordat we de controle terugkrijgen en het tweede pinverzoek kunnen doen. U kunt dus in tussentijd geen andere zaken doen, zoals het informeren van de gebruiker over de voortgang van de pinopdracht.

Multicast delegates

In het voorbeeld heeft u nu gezien hoe een tweetal delegates in aparte calls naar Pinner.DoePinVerzoek werden gebruikt. Het is ook mogelijk om delegates te combineren tot een gelinkte lijst van delegates. Een dergelijke lijst van delegates is afgeleid van het type System.MulticastDelegate, dat weer van System.Delegate is afgeleid. Een MulticastDelegate object heeft naast de eerder genoemde interne Target en Method member variabelen ook een verwijzing naar het vorige delegate object in de gelinkte lijst van delegates. De laatste delegate in de lijst verwijst niet meer naar een delegate object.

Overigens blijkt dat de Visual Basic.NET compiler eigenlijk altijd delegate types implementeert die afgeleid zijn van System.MulticastDelegate en niet van System.Delegate, of u nu gebruik maakt van multicasting of niet.

De System.Delegate class bevat zowel twee statische methoden als instantie methoden Combine en Remove om een nieuw delegate object aan de lijst toe te voegen respectievelijk te verwijderen. Bij het gebruik van de statische methoden moet u er rekening mee houden dat deze een referentie van het type Delegate teruggeven. Als u deze wilt opslaan in een variabele van uw specifieke delegate type, zoals PinVerzoekDelegate, zult u nog een cast met behulp van CType van de Delegate referentie naar PinVerzoekDelegate moeten gebruiken.

U maakt als volgt gebruik van multicast delegates

```

Dim objPinDelegateMC As PinVerzoekDelegate
objPinDelegateMC = CType(Delegate.Combine(objPinDelegate1, _
    objPinDelegate2), PinVerzoekDelegate)
blnGelukt = objPinner.DoePinVerzoek(objPinDelegateMC, 200)

```

waarbij objPinDelegate1 en objPinDelegate2 twee verschillende delegates van het type PinVerzoekDelegate zijn.

Binnen de methode DoePinVerzoek zal bij de aanroep van Invoke

```
blnGelukt = pvd.Invoke(Bedrag, sngMaximum)
```

de PinVerzoekDelegates in de lijst van de multicast delegate pvd achtereenvolgens aangeroepen worden. Effectief levert deze code dus hetzelfde resultaat als bij het gebruik van twee afzonderlijke aanroepen

```
blnGelukt = objPinner.DoePinVerzoek(objPinDelegat1, 200)
blnGelukt = objPinner.DoePinVerzoek(objPinDelegat2, 200)
```

Zoals u wellicht ziet zitten er aan het gebruik van een multicast delegate minstens twee nadelen. Er wordt slechts één methode aanroep DoePinVerzoek gedaan en daarom is het niet mogelijk om per delegate in de lijst verschillende argumenten mee te geven. U bent genoodzaakt om bij alle pinautomaten voor hetzelfde bedrag te pinnen. Daarnaast verliest u alle return waarden, behalve die van de laatste delegate aanroep in de gelinkte lijst.

De CLR zal de multicast delegate gebruiken om gewone synchrone aanroepen van de delegates in de lijst te maken. Deze worden één voor één uitgevoerd en niet parallel zoals wellicht vermoed werd. U zult aan het einde van de volgende paragraaf zien hoe u multicast delegates asynchroon kunt laten uitvoeren.

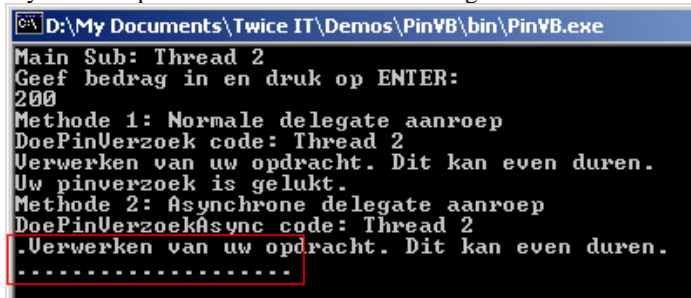
Asynchrone methode aanroepen met delegates

Delegates worden extra interessant in een situatie waar we asynchrone uitvoer van methoden willen introduceren. Normaliter zal voor asynchrone invocatie van methoden extra programmacode moeten worden geschreven in de methode die we asynchroon willen aanroepen.

Delegates maken het asynchroon aanroepen van methoden aanzienlijk eenvoudiger. Eerder heeft u de Delegate.Invoke methode gebruikt om (synchrone) aanroepen van een methode te doen. In de disassembly zag u dat er ook nog methoden BeginInvoke en EndInvoke gegenereerd worden door de compiler voor ieder delegate type. Deze laatste twee methoden bieden de mogelijkheid om de gewenste asynchrone aanroepen te doen.

In plaats van de aanroep van Invoke zult u voor asynchrone aanroepen de BeginInvoke methode gebruiken. Hiermee wordt de delegate methode gestart, waarna de controle direct terugkomt en niet pas nadat de code binnen de delegate methode is uitgevoerd. De code volgend op BeginInvoke zal direct na de aanroep van BeginInvoke uitgevoerd worden, terwijl tegelijkertijd de delegate methode wordt uitgevoerd. Dit wordt mogelijk gemaakt doordat de code van de delegate methode op een andere thread dan die van de aanroepende code wordt uitgevoerd. Er wordt dus multithreading binnen uw applicatie gebruikt, zonder dat u daar eigen code voor heeft hoeven schrijven.

In het te downloaden voorbeeld project wordt er direct na het aanroepen van BeginInvoke een "nuttige" statusupdate naar de gebruiker gestuurd terwijl het pinverzoek asynchroon verloopt: er worden punten naar het console window geschreven. Echter, zoals u hieronder ziet is de eerste punt reeds geplaatst voordat de melding "Verwerken van uw opdracht..." van de Chippen methode verschijnt. Dit komt doordat na de aanroep van BeginInvoke het afdrukken van de punten direct plaatsvindt, nog voordat de asynchrone operatie door de tweede thread is gestart.



```
D:\My Documents\Twice IT\Demos\PinVB\bin\PinVB.exe
Main Sub: Thread 2
Geef bedrag in en druk op ENTER:
200
Methode 1: Normale delegate aanroep
DoePinVerzoek code: Thread 2
Verwerken van uw opdracht. Dit kan even duren.
Uw pinverzoek is gelukt.
Methode 2: Asynchrone delegate aanroep
DoePinVerzoekAsync code: Thread 2
Verwerken van uw opdracht. Dit kan even duren.
.....
```

Verder is er ook code opgenomen die laat zien op welke thread de code van de diverse methoden worden uitgevoerd.

Bij de aanroep van BeginInvoke krijgt u een object terug in de vorm van een referentie naar een IAsyncResult interface en niet de returnwaarde van de delegate methode. Immers, de methode moet nog uitgevoerd worden en dus zijn er nog geen returnwaarden. Deze IAsyncResult interface heeft een viertal properties: CompletedSynchronously, IsCompleted, AsyncState en AsyncWaitHandle.

De IsCompleted property biedt de mogelijkheid om te pollen of de asynchrone operatie afgerond is. Voor een dergelijke asynchrone aanpak met polling, kan de implementatie van de PinVerzoek methode vervangen worden door

```
Dim ar As IAsyncResult
ar = pcb.BeginInvoke(Bedrag, sngMaximum, Nothing, Nothing)
While Not ar.IsCompleted
    Console.WriteLine(".")
    System.Threading.Thread.Sleep(300)
End While
Console.WriteLine()
blnGelukt = pcb.EndInvoke(sngMaximum, ar)
```

Let u voorlopig nog even niet op de twee Nothing argumenten in de BeginInvoke aanroep.

Tijdens het pollen geven we een eenvoudige terugmelding naar de gebruiker dat de opdracht nog in behandeling is. Omdat er normaliter nog een returnwaarde en ByRef waarden uit de aanroep van de methode komen, moet er nog een afrondende aanroep met behulp van EndInvoke gedaan worden. Deze methode bevat niet meer alle argumenten, zoals bij BeginInvoke, maar alleen de output parameters (ByRef argumenten) en de IAsyncResult referentie (de variabele ar) als laatste parameter. De return waarde is natuurlijk dezelfde als die van de methode die normaal synchroon zou worden aangeroepen, in dit geval een boolean.

Callback methoden

In niet alle situaties is het wenselijk of mogelijk om in een lus te blijven pollen en te wachten op het voltooiën van de asynchrone operatie. In die situaties is het ook mogelijk om een callback te laten uitvoeren door de asynchrone methode als deze klaar is.

Callbacks worden gebruikt om bepaalde data en/of gebeurtenissen terug te melden via een aanroep van de betreffende callback methode. Uiteraard kunnen we delegates heel goed gebruiken om callbackmethoden mee door te geven. De base classes van het .NET Framework maken op meerdere plaatsen gebruik van delegates om callbacks te laten uitvoeren.

Zelf kunt u ook callback structuren in uw programma opnemen. U definieert een callback delegate type dat de signature heeft van de gewenste callback methoden. U zult dan een delegate object mee gaan geven aan een routine die de callback moet maken. Deze routine zal de delegate gebruiken om op het juiste moment de callback uit te voeren.

Wanneer u in plaats van pollen naar het afronden van de asynchrone operatie een callback wilt ontvangen dan zult u een delegate mee moeten geven die de callbackmethode bevat. Dergelijke callback delegates moeten van het type System.AsyncCallback zijn. De methode waarop de callback plaatsvindt moet van de vorm

```
Public Sub CallbackMethode(ByVal ar As IAsyncResult)
```

zijn.

Als u een asynchrone aanroep van een methode wilt doen met een callback als deze operatie klaar is, dan zult u een tweetal delegate objecten moeten aanmaken: één delegate voor de asynchroon aan te roepen methode en één delegate voor de callback methode. Deze laatste delegate dient als de voorlaatste parameter van de

BeginInvoke methode mee te geven, waar eerst de Nothing stond die u nog moest vergeten.

```
Dim ar As IAsyncResult
Dim callback As AsyncCallback
callback = New AsyncCallback(AddressOf Me.OnPinCallback)
ar = pcb.BeginInvoke(Bedrag, Maximum, callback, Nothing)
'Hierna volgt nuttige processing
```

De klasse Pinner is uitgebreid met een statische methode OnPinCallback.

```
Public Class Pinner
    ...
    Public Shared Sub OnPinCallback
        Dim pcb As PinVerzoekDelegate
        pcb = CType(CType(ar, _
System.Runtime.Remoting.Messaging.AsyncResult).AsyncDelegate,
PinVerzoekDelegate)
        blnGelukt = pcb.EndInvoke(Maximum, ar)
    ...
End Sub
End Class
```

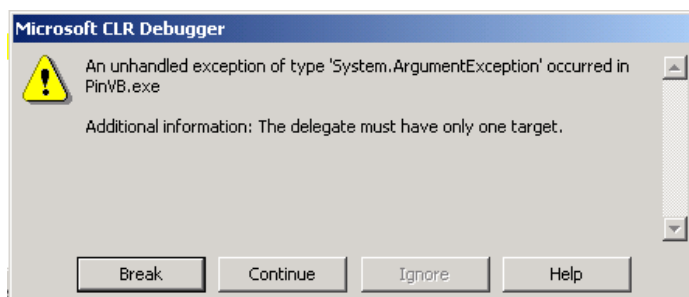
De callback wordt automatisch voor u gemaakt bij het afronden van de asynchrone operatie. Deze aanroep wordt gedaan door de thread waarop ook de asynchrone methode heeft gedraaid. De callback methode heeft als taak om de EndInvoke methode van de delegate aan te roepen, daarmee de returnwaarden uit te lezen en de verdere afhandeling van deze resultaten te doen.

Niet altijd zult u de delegate waarmee de asynchrone operatie gestart werd in een klasse variabele op hebben geslagen. U heeft dan ook niet (meer) rechtstreeks beschikking hebben over dit delegate object. Het is gelukkig mogelijk om uit de IAsyncResult ar variabele die u bij de aanroep van in de callback methode ontvangt het delegate object weer tevoorschijn te halen, zoals ook in het bovenstaande codevoorbeeld gedaan wordt. Er wordt een tweetal typecasts gedaan, van System.IAsyncResult naar System.Runtime.Remoting.Messaging.AsyncResult en vervolgens naar uw delegate type PinVerzoekDelegate. Daarna kan zoals voorheen de aanroep van EndInvoke gemaakt worden met behulp van het zojuist verkregen delegate object.

U heeft tot nu toe steeds Nothing meegegeven als laatste parameter van de BeginInvoke methode van een delegate. Deze laatste parameter is de DelegateAsyncState variabele van het type System.Object. Hierin kunt u een willekeurig object meegeven dat u gebruikt als de state van uw asynchrone aanroep. Zo zou u een uniek handtekening (bijvoorbeeld een integer) mee kunnen geven, zodat als een de asynchrone aanroep voltooid is u tenminste weet welke dat is. Immers, bij meerdere asynchrone aanroepen kan het makkelijk gebeuren dat methoden niet in dezelfde volgorde klaar zijn als dat ze zijn aangeroepen. U kunt hier verder zelf invulling aan geven. Wat u ook kunt doen als u dit niet nodig heeft is het meegeven van het delegate object waarmee de aanroep juist gedaan wordt. Iedere keer dat u de IAsyncResult variabele ar in handen krijgt kunt u het delegate object terughalen via ar.AsyncState. Dit is net iets eenvoudiger dan de eerder beschreven dubbelvoudige cast.

Asynchrone aanroepen en multicast delegates

Wellicht bent u al op het idee gekomen om multicast delegates asynchrone uit te laten voeren. U zult merken dat dit niet mogelijk is door op een multicast delegate variabele (waarin meerdere delegates zitten) de BeginInvoke methode aan te roepen. U zult dan een ArgumentException krijgen.



Zoals de bovenstaande dialoog al aangeeft, mag een delegate maar één target hebben als u de BeginInvoke methode wilt uitvoeren.

Toch is het mogelijk om multicast delegates asynchrone te laten uitvoeren. U zult dit met de hand moeten opstarten. Daarbij maakt u gebruik van de GetInvocationList methode van een Multicast Delegate object. Deze methode geeft een array terug van Delegate referenties waarop u achtereenvolgens de BeginInvoke methode kunt aanroepen.

```
Dim ar As IAsyncResult
Dim arrDelegates As System.Delegate()
Dim del As PinVerzoekDelegate
Dim callback As AsyncCallback = New _
    AsyncCallback(AddressOf Me.OnPinCallback)
arrDelegates = pvd.GetInvocationList()
For Each del In arrDelegates
    CType(del, PinVerzoekDelegate).BeginInvoke(Bedrag, _
Maximum, callback, Nothing)
Next
```

Zoals u ziet wordt de returnwaarde IAsyncResult in dit geval genegeerd. In dit scenario is het lastiger om te pollen naar de eindes van de asynchrone aanroepen, omdat er meerdere gelijktijdige aanroepen gedaan werden. In plaats daarvan is gekozen voor het maken van callbacks als alternatief voor het pollen.

Samenvatting

Delegates zijn CLR managed typen die vergelijkbaar zijn met functiepunters in de vorm van objecten. Met delegates is het mogelijk om een stuk vrijheid in het werken met (object)methoden te introduceren. Delegates bieden bovendien de mogelijkheid om callback methoden te gebruiken en om methoden asynchrone (multithreaded) uit te laten voeren zonder noemenswaardig extra programmeerwerk. De events in de CLR en dus VB.NET zijn gebaseerd op delegates, dus een goed begrip van delegates maakt het werken met events een stuk inzichtelijker.

Over de auteur

Alex Thissen is werkzaam bij Twice IT als senior docent. Hij geeft trainingen over Internet applicaties, XML technologieën en het .NET platform. Verder heeft hij een eigen bedrijf Killer-Apps dat gespecialiseerd is in het geven van advies over en het leveren van maatwerk oplossingen op basis van Microsoft technologieën. Alex is te benaderen op athissen@twice.nl of athissen@killer-apps.nl.