Donald Hessing en Loek Duys zijn werkzaam bij VX Company

Migratie in de praktijk

DE OVERSTAP NAAR .NET FRAMEWORK 2.0

De overstap van Visual Studio 2003 naar Visual Studio 2005 is voor veel bedrijven een logische stap. Met deze stap wordt ook de overgang van .NET Framework 1.1 naar 2.0 gemaakt. De in Visual Studio 2005 ingebouwde wizard helpt bestaande applicaties te migreren. Voor de meeste applicaties zijn deze stappen afdoende. Maar er zit een addertje onder het gras.

Er zit in .NET Framework 2.0 een aantal fundamentele wijzigingen. Wanneer zo'n wijziging invloed heeft op code, geeft de compiler soms een waarschuwing. Maar waarschuwingen kunnen worden genegeerd. De code kan worden gecompileerd en uitgevoerd. Toch kan de applicatie zich heel anders gedragen. Wat veroorzaakt dit probleem? En wat lost het op? In dit artikel worden drie belangrijke probleemgevallen beschreven die we in de praktijk zijn tegengekomen. Voor elk probleem wordt een oplossing gegeven. Om de verschillen tussen beide versies van .NET Framework het beste te kunnen illustreren, zijn bij dit artikel twee solutions met voorbeeldcode te downloaden. De twee solutions wijzen naar één verzameling codebestanden. Wanneer een codebestand wordt aangepast, werkt dat door in beide solutions. Door Visual Studio 2003 en Visual Studio 2005 naast elkaar te gebruiken, wordt duidelijk wat de verschillen in gedrag zijn tussen .NET Framework 1.1 en 2.0.

Case 1. Mijn XmlWriter doet het niet meer

De *XmlWriter* (of *XmlTextWriter*) wordt gebruikt om op een snelle manier XML volgens de W3C-standaard in een *Stream* of bestand te schrijven. De standaard dwingt af dat XML voldoet aan een bepaalde opmaak. Dit wil bijvoorbeeld zeggen dat elementen die worden geopend ook moeten worden afgesloten. De XmlWriter zorgt ervoor dat deze regels worden toegepast. In principe is het onmogelijk daarmee XML te schrijven die niet aan de standaard voldoet. Er treedt een exception op wanneer dat geprobeerd wordt. Een verschil in implementatie van de XmlWriter in de beide frameworks kan toch tot verrassingen leiden.

Het probleem

De XmlWriter in .NET Framework 1.1 is tolerant genoeg om beschikbaar te blijven voor het schrijven van XML, nadat eerst is geprobeerd ongeldige XML te schrijven in de onderliggende *Stream*. Een voorbeeld hiervan staat beschreven in codevoorbeeld 1. Binnen de method *Test* wordt een nieuwe XmlTextWriter aangemaakt. Hiermee wordt vervolgens een XML-declaratie-element geschreven. Daarna wordt een element met de naam 'onbekende_inhoud' geschreven. De inhoud van dat element wordt verzorgd door een andere method, die de XmlWriter als parameter meekrijgt. De foutafhandeling in die method is zo ingericht dat exceptions worden genegeerd, die door de XmlWriter zijn gegooid. Er wordt geprobeerd ongeldige XML met de XmlWriter te schrijven. We proberen namelijk een XML-node die als element is gestart, als attribute af te sluiten. Doordat de exception niet wordt doorgegeven aan de aanroepende code, is het daar niet bekend dat er iets is misgegaan bij het gebruiken van de XmlWriter. Het is in de code die onder .NET Framework 1.1 draait nog prima mogelijk de XmlWriter te gebruiken om het document af te sluiten. De resulterende XML kan daarna bijvoorbeeld worden ingelezen als geldig *XmlDocument*. Het is niet zichtbaar dat er informatie ontbreekt; in dit geval de inhoud van het element 'subElement'. In .NET Framework 2.0 heeft de XmlWriter een extra statusmogelijkheid gekregen voor zijn property *WriteState*, namelijk *WriteState.Error*. Wanneer de property WriteState deze waarde heeft, kan geen enkele XML-schrijffunctie van de XmlWriter meer worden gebruikt. Bij het aanroepen is een *InvalidOperationException* het resultaat. De property WriteState krijgt de waarde *'Error'* wanneer met de XmlWriter is geprobeerd ongeldige XML te schrijven. Wanneer de voorbeeldcode onder .NET Framework 2.0 draait, is het niet meer mogelijk tot een geldige XML-structuur te komen. Dit komt doordat de XmlWriter niet langer in staat is de method *WriteEndDocument* uit te voeren, nadat in de method *WriteDocumentInhoud* een fout is opgetreden.

Een oplossing

In dit geval wordt het verschil in werking niet door de C#-compiler veroorzaakt, maar door de implementatie van de XmlWriter zelf. De oplossing voor deze case wordt dus geboden door de nieuwe implementatie. Een betere foutafhandeling in de method *WriteDocumentInhoud* voorkomt ook het probleem, maar dit kan onmogelijk zijn wanneer je bijvoorbeeld een XML-component van een andere leverancier gebruikt. We raden aan om na het aanroepen van een method zoals *WriteDocumentInhoud*, de *WriteState*-property te inspecteren, voordat je de XmlWriter opnieuw gebruikt om XML te schrijven.

Case 2. De verkeerde functie wordt uitgevoerd

Overloading is het definiëren van verscheidene methods met dezelfde naam in één type, met verschillende parameters. Een delegate is een type dat naar een method verwijst. Wanneer aan een delegate een method is toegewezen, gedraagt hij zich als die method. Net als bij een method kun je parameters en een returnwaarde gebruiken. Naast de mogelijkheid om via de delegate een method aan te roepen, kan hij ook als parameter aan andere methods worden gegeven. In het .NET Framework wordt veel gebruik gemaakt van delegates. Je kunt bijvoorbeeld via delegates methods registreren voor:

- notificaties voor niet afgehandelde exceptions
- control-events in windows- en webforms
- afgehandelde asynchrone operaties

In de eerste case wordt besproken hoe het kan dat bij gebruik van overloading en delegates in .NET Framework 2.0 andere code wordt uitgevoerd dan in .NET Framework 1.1.

```
public static void Test(){
   XmlWriter writer = null;
   using (MemoryStream memoryStream = new MemoryStream()){
      try{
         writer = new XmlTextWriter(memoryStream,
            UTF8Encoding.UTF8);
         writer.WriteStartDocument();
         writer.WriteStartElement("onbekende_inhoud");
         //De inhoud van het 'onbekende_inhoud'-element
         // wordt door een andere method geleverd.
         WriteDocumentInhoud(writer);
         //Document afsluiten
         writer.WriteEndDocument();
      }
      catch { [..] }
      finally{ [..] }
   }
}

private static void WriteDocumentInhoud(XmlWriter writer){
   try{
      writer.WriteStartAttribute("attribuut", "");
      writer.WriteString("attribuut waarde");
      writer.WriteStartElement("subElement");
      //Verkeerd, het element moet eerst worden
      //afgesloten:
      writer.WriteEndAttribute();
      //wordt niet meer uitgevoerd:
      writer.WriteString("element waarde");
      writer.WriteEndElement();
   }
   catch { }
}
```

Codevoorbeeld 1.

Het probleem

Een voorbeeld van een delegate is te zien in codevoorbeeld 2. De delegate wordt gedeclareerd met een method signature. De signature is een omschrijving van de types die de methodparameters en -returnwaarde hebben. De gedefinieerde delegate kan alleen een method toegewezen krijgen die een overeenkomstige signature heeft. Welke methods geldig zijn, wordt door de compiler bepaald. Bij het zoeken naar de methods gelden in .NET Framework 2.0 andere regels dan in 1.1. Het verschil tussen deze regels wordt besproken in deze case. In ieder scenario, waarbij er verscheidene overloads van een method zijn en een delegate gebruikt wordt om één van de methods aan te roepen, kan het probleem optreden.

In .NET Framework 1.1 is alleen een method met een exact gelijke signature te registreren bij een delegate. Nieuw in .NET Framework 2.0 is covariance en contravariance. Covariance betekent dat het return-type van een method - naast hetzelfde type - ook een overerving kan zijn van het return-type van de delegate. Contravariance betekent dat een method - naast hetzelfde type - ook een parameter mag ontvangen die een base van het type van de delegate-parameter is. In tabel 1 staat weergegeven welke combinaties geldig zijn. Deze nieuwe functionaliteit biedt meer flexibiliteit bij het werken met delegates. Er kan bijvoorbeeld een universele eventhandler worden gemaakt, die kan reageren op zowel een muisbeweging als toetsenbordaanslagen.

Door de verschillen die in tabel 1 zijn weergegeven, kan het gebeuren dat in .NET Framework 1.1 andere code wordt uitgevoerd dan in .NET Framework 2.0. Dit illustreren we aan de hand van een codevoorbeeld. We beginnen met het definiëren van een *BaseType* en een *Inherited* type. Daarnaast wordt in de namespace een delegate gedeclareerd, die een parameter krijgt van het type *Inherited*. Het geheel staat beschreven in codevoorbeeld 2.

In de method *Test* wordt een delegate van het type *DoWorkDelegate* aangemaakt. Deze wordt toegewezen aan de method *DoWork* op een instantie van *Inherited*. Als de code met .NET Framework 1.1 wordt gecompileerd, wijst de delegate naar de method met commentaar 'Method1'. Dit ligt voor de hand, aangezien de signature identiek is aan de delegate. Maar als de code gecompileerd wordt met .NET Framework 2.0 wijst de delegate naar de method *DoWork* met commentaar 'Method 2' op type *Inherited*. Dit ondanks dat de method *DoWork* met signature *void(Inherited)* ook beschikbaar is door overerving. Bij het zoeken naar de juiste overload wordt het meest specifieke type eerst door de compiler bekeken. Het feit dat Method 2 in .NET Framework 2.0 geldig is, wordt veroorzaakt door *contravariance*. Omdat *System.Object* een base is van *Inherited*, is deze method de uiteindelijke kandidaat. De compiler geeft een waarschuwing (CS1707) wanneer deze code wordt gecompileerd.

Een oplossing

Een oplossing voor dit probleem is het declareren van de variabele *inherited* als *BaseType*. De compiler doorzoekt dan alleen dat type voor de method die aan de delegate wordt toegewezen. Omdat het type *BaseType* zelf maar één method heeft, wordt Method 1 aangeroepen. De uitwerking is te zien in codevoorbeeld 3. Er is ook een andere oplossing. Bij het toewijzen van de method aan de delegate kun je de variabele *inherited* eerst naar *BaseType* casten. In dat geval kan de variabele ook gewoon als type *Inherited* worden gedeclareerd.

Case 3. Mijn applicatie crasht

Om verschillende taken tegelijk uit te voeren binnen een applicatie wordt multi-threading gebruikt. In het .NET Framework zijn er diverse manieren om dat te ondersteunen. Verschillen in de implementatie van beide .NET Frameworks kan ertoe leiden dat code in .NET Framework 1.1 goed lijkt te functioneren en in .NET Framework 2.0 onverwacht tot het stoppen van de applicatie leidt. De code van een applicatie in het .NET Framework wordt uitgevoerd in een thread. Een applicatieproces bestaat daarom altijd uit minimaal één thread. Het is ook mogelijk taken parallel uit te voeren. In dat geval worden in het applicatieproces één of meer extra threads gemaakt. De thread die de applicatie start en vervolgens als een draad door de gehele applicatie loopt, noemen we

Delegate signature	Method Signature	Geldig in 1.1?	Geldig in 2.0?
Void (BaseType)	void (BaseType)	Ja	Ja
BaseType (void)	BaseType (void)	Ja	Ja
Void (BaseType)	void (Inherited)	Nee	Nee
Void (Inherited)	void (BaseType)	Nee	Ja (contravariance)
Inherited (void)	BaseType (void)	Nee	Nee
BaseType (void)	Inherited (void)	Nee	Ja (covariance)

Tabel 1. Een aantal voorbeelden van delegate- / method-combinaties

```
delegate void DoWorkDelegate(Inherited param);
class BaseType{
   //Method 1.
   public void DoWork(Inherited param){}
}
class Inherited : BaseType{
   //Method 2.
   public void DoWork(object param){}
}

class Test{
   private static void Test(){
      Inherited inherited = new Inherited();
      DoWorkDelegate dlgt = new
         DoWorkDelegate(inherited.DoWork);
      //Via de delegate aanroepen
      dlgt(inherited);
   }
}
```

Codevoorbeeld 2.

in dit artikel de mainthread. Threads die vanuit de mainthread worden gecreëerd voor parallelle taken, noemen we in dit artikel workerthreads.

Het probleem

Wanneer in de applicatie een fout optreedt die niet wordt afgevangen door een *catch*-blok, treedt een unhandled exception op. In het .NET Framework 2.0 is de manier gewijzigd waarop unhandled exceptions in workerthreads worden afgehandeld.
Stel je een weerstationapplicatie voor, waarbij één workerthread wordt gebruikt om continu de temperatuur te lezen en een andere om de luchtvochtigheid te meten. Wanneer de temperatuur onder een minimum komt, levert de temperatuur-workerthread een unhandled exception op. In het .NET Framework 1.1 zal de temperatuur-workerthread worden beëindigd. De unhandled exception van de temperatuur-workerthread komt niet in de mainthread terecht, waardoor de luchtvochtigheid-workerthread in leven blijft. In het .NET Framework 2.0 zal de unhandled exception worden doorgegeven aan de mainthread. Het resultaat is dat de mainthread wordt beëindigd. Daarmee wordt ook de workerthread voor het lezen van de luchtvochtigheid beëindigd. Standaard zorgen unhandled exceptions in het .NET Framework 2.0 ervoor dat de mainthread onderuitgaat, ongeacht op welke thread deze worden veroorzaakt. Een uitzondering op deze regel is het asynchroon gebruikmaken van delegates. Daarvoor is het gedrag in 2.0 nog steeds hetzelfde als in .NET Framework 1.1.
In codevoorbeeld 4 wordt in het *try* / *catch*-blok een nieuwe workerthread gestart. Wanneer op de workerthread een unhandled exception optreedt, kan deze niet afgevangen worden door de *catch* van de mainthread. Dat komt omdat threads geen informatie delen over de method die ze uitvoeren. Met andere woorden: unhandled exceptions afkomstig van een workerthread kunnen niet afgevangen worden door een *try* / *catch*-blok in de mainthread.

Een oplossing

Een oplossing voor dit probleem is weergegeven in codevoorbeelden 4 en 5. Bij het creëren van een thread moet worden aangegeven welke code de thread moet gaan uitvoeren; zie codevoorbeeld 4. Dit kun je bereiken door een *ThreadStart*-delegate te creëren, waaraan vervolgens de method *DoWork* wordt toegewezen. Vervolgens creëren we een *Thread*-object door in de constructor het *ThreadStart*-object mee te geven. Zodra op het gecreëerde *Thread*-object de method *Start* wordt aangeroepen, zal de method *DoWork* van het object *Worker* worden uitgevoerd. De method *DoWork* staat volledig binnen een *try* / *catch*-blok, dit om unhandled exceptions te voorkomen op de workerthread; zie codevoorbeeld 5. In de method *DoWork* wordt een unhandled exception gesimuleerd door het gooien van een exception van het type *SampleException*. De exception wordt afgevangen door het *catch*-blok van de method *DoWork* en toegekend aan de public property *MyException*.
Zodra de workerthread in de mainthread is gestart, moet de mainthread wachten totdat de workerthread klaar is. Dit bereiken we door de method *Thread.Join* aan te roepen; zie codevoorbeeld 5. Voordat we verder gaan, moet eerst worden gecontroleerd of er geen fouten zijn opgetreden. Het object waarvan zojuist de method *DoWork* is uitgevoerd, is nog steeds beschikbaar. Merk op dat in het *Worker*-type een property *MyException* is gedefinieerd die de exception bevat zoals we deze hebben toegekend in de *catch* van de method *DoWork*. Door deze te inspecteren, heeft de mainthread de beschikking over de eventuele exception die is opgetreden in de workerthread. Afhankelijk van het type exception kan de mainthread verdergaan met zijn werk of de exception in een nieuwe exception verpakken en deze gooien.
De essentie van deze oplossing zit hem in het feit dat de volledige code, die door de workerthread wordt uitgevoerd, binnen een *try* / *catch*-blok staat. In het *catch*-blok vangen we alle fouten af van het type *Exception*, zodat er geen exception kan optreden voor welke geen *catch*-blok is gedefinieerd. Vervolgens beslist de mainthread wat met de opgetreden exception moet gebeuren. Op deze manier kunnen er in principe geen unhandled exceptions in de workerthread meer optreden. De oplossing voorkomt dat in het .NET Framework 2.0 de applicatie eindigt en dat in het .NET Framework 1.1 de workerthread wordt beëindigd zonder dat de applicatie hiervan op de hoogte is. De oplossing is dus voor beide .NET Frameworks geschikt.

```
class Test{
   private static void Test(){
      BaseType inherited = new Inherited();
      MethodDelegate dlgt = new
         MethodDelegate(inherited.Method);
      //Via de delegate aanroepen
      dlgt((Inherited)inherited);
   }
}
```

Codevoorbeeld 3.

```
static void RunWorker()
{
   Worker w = new Worker();
   ThreadStart ts = new ThreadStart(w.DoWork);
   Thread t = new Thread(ts);
   try {
      t.Start();
      t.Join();
      if (!(w.MyException is SampleException))
         throw w.MyException;
   }
   catch (Exception ex) {
      // Exception komt niet in de catch
      [..]
   }
}
```

Codevoorbeeld 4.

```
public Exception MyException { get[..] }

public void DoWork() {
   // Voorkom unhandled exceptions op de mainthread.
   try  {
      // Simuleer workerthread exception
      throw new SampleException("exception from a thread");
   }
   catch (Exception ex) {
      //Bewaar de opgetreden exception
      this._myException = ex;
   }
}
```

Codevoorbeeld 5.

Ten slotte

Je kunt niet altijd controleren of alle exceptions netjes worden afgevangen, bijvoorbeeld wanneer je gebruikmaakt van componenten van andere partijen. Om een vangnet te hebben voor die gevallen, bestaat het event *UnhandledException*. Hierbij is belangrijk te weten dat de handler van het event *UnhandledException* geen exceptionhandler is, waarbij de applicatie verdergaat na de afhandeling van de exception. Het event *UnhandledException* is een event dat je de mogelijkheid geeft, om bij een unhandled exception in de applicatie (ongeacht op welke thread deze voorkomt), de applicatie netjes af te sluiten. Dit kun je bereiken door bijvoorbeeld de exception te loggen in het eventlog en de gebruiker op de hoogte te stellen dat de applicatie eindigt.
In codevoorbeeld 6 registreren we een eventhandler voor het event *UnhandledException* van het AppDomain; feitelijk de applicatie zelf. De eventhandler ontvangt de parameter *UnhandledExceptionEventArgs* die de property *ExceptionObject* bevat. Het *ExceptionObject* kan op zijn beurt weer gecast worden naar het *Exception* type. Het gecaste object is de oorspronkelijke exception van de workerthread. In codevoorbeeld 6 heeft de workerthread de exception gegooid, die nu beschikbaar is op de mainthread. De informatie uit het exception-

```
static void Main(string[] args)  {
   AppDomain.CurrentDomain.UnhandledException += new
      UnhandledExceptionEventHandler(MyHandler);
   RunWorker();
}

static void MyHandler(object sender, UnhandledExceptionEventArgs args) {
   Exception e = (Exception)args.ExceptionObject;
   Console.WriteLine("MyHandler caught : " + e.Message);
   Environment.Exit();
}
```

Codevoorbeeld 6.

object kan nu gebruikt worden voor het loggen van de fout die is opgetreden in de workerthread. Een laatste redmiddel is gebruik te maken van het configuratie-element *legacyUnhandledExceptionPolicy*. Hiermee worden unhandled exceptions in .NET Framework 2.0 op dezelfde manier afgehandeld als in .NET Framework 1.1.

Samenvattend

Bij de migratie van code van .NET Framework 1.1 naar 2.0 moet rekening worden gehouden met verschillende implementaties van de beide .NET Frameworks. De compiler geeft waar mogelijk waarschuwingen, maar zelfs wanneer deze waarschuwingen zijn weggewerkt, kan de identieke code nog verschillend gedrag vertonen. Bij *XmlWriters* is in .NET Framework 1.1 niet aan het object zelf te zien of er een onderdrukte fout is opgetreden tijdens het gebruik daarvan. Dit is met name een probleem als de *XmlWriter* wordt gebruikt in meer (externe) methods. De resulterende XML is op dat moment mogelijk onvolledig, zonder dat de aanroepende partij daarvan op de hoogte is. In .NET Framework 2.0 krijgt de *XmlWriter* bij fouten de waarde '*Error*' in zijn *WriteState*-property. In dat geval kan de *XmlWriter* niet langer worden gebruikt om XML mee te schrijven. In code waarin gebruik wordt gemaakt van delegates, kan het voorkomen dat in het .NET Framework 2.0 een andere method wordt aangeroepen dan in het .NET Framework 1.1. Een oplossing hiervoor is dat je bij de toekenning van de method expliciete casting gebruikt om zeker te zijn dat de gewenste method wordt uitgevoerd. Voor applicaties die gebruikmaken van meer threads moet goed worden gekeken of alle exceptions wel in de workerthread worden afgevangen. Voor die gevallen waarin er toch een unhandled exception optreedt, is het aan te raden het event *UnhandledException* van het *AppDomain* te gebruiken. Op deze manier kan de fout die is opgetreden op de workerthread worden achterhaald en worden gelogd. Naast het doorlopen van de eigen code moet ook kritisch gekeken worden naar het gebruik van componenten van andere partijen. Deze componenten kunnen prima werken in .NET Framework 1.1, maar na de migratie een ander gedrag tonen.

Donald Hessing is technisch teamleider bij VX Company. Hij ontwerpt en ontwikkelt al vanaf de eerste bèta gedistribueerde systemen met .NET-technologie. Donald is bereikbaar op dhessing@vxcompany.com.

Loek Duys is als senior developer werkzaam bij VX Company. Zijn werkzaamheden richten zich op het ontwerpen en implementeren van .NET-oplossingen. Hij is te bereiken via lduys@vxcompany.com.

Referenties

MSDN-artikel over .NET Framework 1.1 en 2.0 compatibiliteit:
http://msdn.microsoft.com/library/en-us/dnnetdep/html/netfxcompat.asp
Covariance en contravariance in delegates:
http://msdn2.microsoft.com/en-us/library/ms173174.aspx
De unhandled exception policy van .NET Framework 1.1 in 2.0 gebruiken:
http://msdn.microsoft.com/msdnmag/issues/05/10/Reliability/default.aspx?side=true