Maarten Struys is .NET Compact Framework MVP, werkzaam bij PTS Software bv

.NET Compact Framework

DEEL 1: MULTITHREADED APPLICATIES

In dit eerste van twee artikelen over multithreaded applicaties voor het .NET Compact Framework staan we stil bij het creëren, starten en stoppen van verscheidene threads binnen een applicatie. We zien wanneer het noodzakelijk is gebruik te maken van multithreading, maar ook dat een multithreaded applicatie complexer is dan een single threaded applicatie. We schenken speciale aandacht aan het gebruik van userinterface-elementen vanuit workerthreads. In het volgende nummer van het .NET Magazine behandelen we de synchronisatie tussen verschillende threads, thread-safety en nieuwe mogelijkheden die beschikbaar komen in versie 2 van het .NET Compact Framework.

Ondanks het feit dat we ons specifiek richten op het .NET Compact Framework is de informatie in dit artikel ook relevant voor ontwikkelaars die gebruik maken van het volledige .NET Framework. De omvang van het .NET Compact Framework is ongeveer 10% van de omvang van het .NET Framework. Omdat het .NET Compact Framework slechts een deel van de functionaliteit van het .NET Framework bevat, zullen sommige voorbeeldcodes voor ervaren .NET-ontwikkelaars wellicht inefficiënt lijken. Alle in dit artikel gepresenteerde voorbeeldcodes werken echter ook in een niet-Windows CE-omgeving. De voorbeelden zijn bewust heel eenvoudig gehouden. Voordat we ingaan op multithreading zetten we eerst de verschillen tussen processen en threads in het Windows CE-besturingssysteem op een rijtje.

Windows CE Processen

Net zoals in elk ander Windows-besturingssysteem is een proces in Windows CE een 'omgeving' waarbinnen een applicatie 'leeft'. Anders gezegd, een proces is een instantie van een werkende applicatie. Een proces bevat zelf geen code die door een processor wordt uitgevoerd. Onder Windows CE is een proces de eigenaar van 32 MB aan geheugenruimte, waarin zowel code als data kunnen zijn opgeslagen. Om code te kunnen uitvoeren moet een proces minstens één thread bevatten. In tegenstelling tot andere Windows-besturingssystemen is het aantal processen dat binnen Windows CE tegelijk actief kan zijn beperkt tot 32. Vaak worden de begrippen proces en applicatie door elkaar heen gebruikt.

Afbeelding 1. Een eenvoudige PocketPC-applicatie

Windows CE Threads

Een thread binnen Windows CE kan worden beschouwd als een 'eenheid van code-uitvoering'. Met andere woorden, een thread is verantwoordelijk voor het uitvoeren van code op de processor. Een Windows CE-applicatie bestaat uit één of meer threads. Al deze threads kunnen verschillende prioriteiten hebben. Het Windows CE-besturingssysteem is onder andere verantwoordelijk voor het verdelen van de totale processortijd over alle verschillende threads die code moeten uitvoeren. Threads met een hogere prioriteit gaan voor threads met een lagere prioriteit. Als meer threads dezelfde prioriteit hebben, krijgen deze om beurten door het besturingssysteem een beetje processortijd toebedeeld. Threads kunnen vrijwillig de hun gegeven processortijd opgeven. Omdat veel Windows CE-apparaten werken op batterijen is het heel belangrijk voor elke applicatie om zo min mogelijk de processor te gebruiken. Daarom is het bijvoorbeeld uit den boze om in een SmartPhone- of PocketPC-applicatie te 'pollen' totdat gegevens beschikbaar zijn. Voor dergelijke applicaties geldt altijd de volgende vuistregel:

Maak nooit gebruik van de processor tenzij het absoluut niet anders kan.

In een desktopomgeving is dit veel minder relevant, hoewel ook niet onbelangrijk. Immers, minder gebruik maken van de processor zal over het algemeen een betere performance opleveren.

De noodzaak voor multithreaded applicaties

Er zijn verschillende scenario's denkbaar waarin de opdeling van een applicatie in meer threads handig en soms zelfs noodzakelijk is. Om dit te illustreren kijken we naar het volgende voorbeeld.

In een simpele PocketPC-applicatie (zie codevoorbeeld 1 voor de volledige broncode) wordt herhaaldelijk een langdurige bewerking uitgevoerd. Om het voorbeeld eenvoudig te houden wordt die langdurige bewerking gesimuleerd door een aanroep van de Thread.Sleep-methode. Omdat in dit eerste voorbeeld geen extra threads worden gecreëerd, wordt alle code uitgevoerd in één enkele thread: de main thread. Ondanks dat een WinForm-applicatie reageert op tal van events die worden gegenereerd door

bijvoorbeeld het klikken op knoppen, worden al deze events dus afgehandeld in dezelfde thread. Kijken we nu naar de code die wordt uitgevoerd op het klikken van de startknop (zie de functie *btnStart_Click* in codevoorbeeld 1), dan blijkt dat die functie niet wordt beëindigd. Er is namelijk geen mogelijkheid dat de variabele stopRequested in de functie SimulateWork op true wordt gezet. Dit zou moeten gebeuren in de eventhandler van de stopknop, maar die kan niet worden aangeroepen, omdat de eventhandler van de startknop niet is beëindigd. Voor de gebruiker lijkt deze applicatie te hangen, omdat de gebruikersinterface niet meer reageert.

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using System.Threading;

namespace MTSampleCode
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form
  {
    private System.Windows.Forms.Button btnStart;
    private System.Windows.Forms.Button btnStop;
    private System.Windows.Forms.ListBox listBox1;
    private System.Windows.Forms.MainMenu mainMenu1;
    private bool stopRequested = false;

    public Form1()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
      //
      // TODO: Add any constructor code after InitializeComponent call
      listBox1.SelectedIndex = -1;
#if DEBUG
      MinimizeBox = false;
#else
      MinimizeBox = true;
#endif
    }
    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      base.Dispose( disposing );
    }
    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
      this.mainMenu1 = new System.Windows.Forms.MainMenu();
      this.btnStart = new System.Windows.Forms.Button();
      this.btnStop = new System.Windows.Forms.Button();
      this.listBox1 = new System.Windows.Forms.ListBox();
      //
      // btnStart
      this.btnStart.Location = new System.Drawing.Point(12, 192);
      this.btnStart.Size = new System.Drawing.Size(104, 32);
      this.btnStart.Text = "Start";
      this.btnStart.Click += new System.EventHandler(this.btnStart_Click);
      //
      // btnStop
      this.btnStop.Enabled = false;
      this.btnStop.Location = new System.Drawing.Point(124, 192);
      this.btnStop.Size = new System.Drawing.Size(104, 32);
      this.btnStop.Text = "Stop";
      this.btnStop.Click += new System.EventHandler(this.btnStop_Click);
      //
      // listBox1
      this.listBox1.Location = new System.Drawing.Point(8, 8);
      this.listBox1.Size = new System.Drawing.Size(224, 142);
      //
      // Form1
      this.Controls.Add(this.listBox1);
      this.Controls.Add(this.btnStop);
      this.Controls.Add(this.btnStart);
      this.Menu = this.mainMenu1;
      this.Text = "MT Sample";
    }
    #endregion
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    static void Main()
    {
      Application.Run(new Form1());
    }

    private void btnStart_Click(object sender, System.EventArgs e)
    {
      AddListBoxEntry("Started working");

      btnStart.Enabled = false;
      btnStop.Enabled  = true;

      stopRequested = false;
      SimulateWork();
    }

    private void btnStop_Click(object sender, System.EventArgs e)
    {
      AddListBoxEntry("Stopped working");
      btnStart.Enabled = true;
      btnStop.Enabled  = false;

      stopRequested = true;
    }

    private void AddListBoxEntry(string newEntry)
    {
      listBox1.Items.Add(newEntry);
      listBox1.SelectedIndex++;
    }

    private void SimulateWork()
    {
      while (! stopRequested)
      {
        Thread.Sleep(1000);
      }
    }
  }
}
```

Codevoorbeeld 1. De noodzaak van multithreading

```
private void btnStart_Click(object sender, System.EventArgs e)
{
   AddListBoxEntry("Started working");

   btnStart.Enabled = false;
   btnStop.Enabled  = true;

   stopRequested = false;

   Thread myWorkerThread = new Thread(new ThreadStart(SimulateWork));
   myWorkerThread.Start();
}
```

Codevoorbeeld 2. SimulateWork ondergebracht in een aparte thread

```
private void Form1_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
   if (! stopRequested)
   {
      MessageBox.Show("Stop WorkerThread before closing the application");
      e.Cancel = true;
   }
}
```

Codevoorbeeld 3. Afsluiten van de applicatie is alleen toegestaan als de worker thread is gestopt

Natuurlijk is het voorbeeld een beetje kunstmatig, maar vertaal onze gesimuleerde actie bijvoorbeeld naar het herhaaldelijk lezen van informatie vanaf een seriële poort. Het is uiteraard onacceptabel dat een dergelijke activiteit een applicatie 'ophangt' als er geen nieuwe data over de poort worden ontvangen.

Om een dergelijke applicatie, die herhaaldelijk acties uitvoert, voor een gebruiker acceptabel te maken, is het bijvoorbeeld mogelijk de betreffende actie uit te voeren in een aparte thread. Om deze applicatie multithreaded te maken, hoeven we in principe slechts de code van de *btnStart_Click* eventhandler aan te passen. Dit is namelijk de plaats waarin we een nieuwe thread gaan maken en starten. In codevoorbeeld 2 is de aangepaste *btnStart_Click* eventhandler te zien. De aanroep van SimulateWork is vervangen door het aanmaken van een nieuwe thread, waarin SimulateWork als parameter wordt meegegeven. De functie SimulateWork wordt nu in een aparte thread uitgevoerd. Die thread wordt beëindigd op het moment dat de functie SimulateWork wordt beëindigd. Dankzij Garbage Collection hoeven we ons geen zorgen te maken over het opruimen van het nieuw geïnstantieerde thread-object. Omdat SimulateWork in een aparte thread wordt uitgevoerd, is het nu mogelijk op de stopknop te klikken. Daardoor wordt de variabele stopRequested op true gezet, waardoor SimulateWork gestopt kan worden, en daarmee dus ook de nieuw aangemaakte thread.

Stoppen van een multithreaded applicatie

Het veranderen van onze applicatie in een multithreaded applicatie brengt echter een nieuw probleem met zich mee. Dit is het beste inzichtelijk te maken door de voorbeeldapplicatie vanuit Visual Studio.NET 2003 in debug-mode te starten, in de emulator op de startknop te klikken en vervolgens in de emulator de applicatie af te sluiten door op 'OK' te klikken. In afbeelding 2 is te zien in welke toestand de debugger en de emulator zich nu bevinden.

Na klikken op de OK-knop van onze applicatie leek het alsof de applicatie werd afgesloten. In de *Running Program List* in de emulator is de applicatie niet meer te zien, wat erop lijkt te duiden dat de applicatie netjes is afgesloten. Toch is dit niet het geval. In afbeelding 2 is te zien dat Visual Studio.NET 2003 nog steeds aangeeft dat de applicatie actief is. De reden hiervoor is dat een managed Windows CE-applicatie pas kan worden afgesloten als alle threads van die applicatie gestopt zijn. Het gedrag van een .NET Compact Framework-applicatie en een .NET Framework-applicatie is op dit punt verschillend. Een multithreaded .NET Framework-applicatie zal netjes worden afgesloten in de volgende gevallen:

Afbeelding 2. Applicatie beëindigd of niet?

- Een applicatie maakt alleen gebruik van background threads. Deze threads worden automatisch gestopt als de main thread stopt.
- Een applicatie maakt gebruik van zowel foreground als background threads. De applicatie wordt pas gestopt als alle foreground threads zijn gestopt.

In het .NET Compact Framework (tenminste in versie 1.0) wordt geen verschil gemaakt tussen background threads and foreground threads. Alle threads in een applicatie worden beschouwd als foreground threads. Om nu toch netjes onze applicatie te kunnen afsluiten kunnen we bijvoorbeeld een eventhandler toevoegen die wordt aangeroepen op het moment dat de applicatie op verzoek van de gebruiker moet worden gestopt. In codevoorbeeld 3 is te zien hoe we een Closing-event gebruiken om te melden dat de applicatie pas kan worden afgesloten nadat de gebruiker de workerthread heeft gestopt. Natuurlijk hadden we er ook voor kunnen kiezen op de Closing-event automatisch de workerthread te stoppen.

Stoppen van een thread

Een vraag die regelmatig in nieuwsgroepen terugkomt, is hoe een workerthread in een .NET Compact Framework-applicatie moet worden gestopt. Deze vraag wordt voornamelijk gesteld omdat in het .NET Compact Framework de functie *Thread.Abort* niet beschikbaar is. Blijkbaar willen veel ontwikkelaars deze functie gebruiken om een thread af te sluiten. Hoewel dat zeker mogelijk is in het .NET Framework, is het maar de vraag of dit een gewenste manier is om een thread af te sluiten. Aanroepen van Thread.Abort resulteert namelijk in een exceptie. Normaal afsluiten van een thread is echter geen foutsituatie en zou dus niet in een exceptie mogen resulteren. Ondanks dat in versie 2.0 van het .NET Compact Framework Thread.Abort beschikbaar is, is het verstandig gebruik van deze functie te beperken tot echte foutsituaties.
Een thread die herhaaldelijk werk moet uitvoeren (bijvoorbeeld het elke seconde verversen van een klok in een statusbar) zal in het algemeen de structuur hebben die staat in codevoorbeeld 4.

Deze thread zal actief blijven totdat stopRequested op true wordt gezet, wat kenmerkend is voor een eventhandler die reageert op een gebruikersactie (bijvoorbeeld het klikken op een stopknop of het beëindigen van de applicatie). In het .NET Framework is het verstandig de variabele stopRequested als 'volatile' te declareren. Dit zorgt ervoor dat de variabele in het geheugen gegarandeerd direct een nieuwe waarde krijgt op een toekenning, zonder bijvoorbeeld de waarde als compileroptimalisatie voorlopig in een

register te laten staan. Het lezen van een volatile variabele gebeurt ook altijd gegarandeerd, zelfs als twee leesoperaties direct na elkaar worden uitgevoerd. In een multithreaded applicatie is dit belangrijk. In een workerthread kunnen bijvoorbeeld twee leesoperaties direct na elkaar worden uitgevoerd, maar in de tussentijd kan een andere thread de variabele, die wordt gelezen, van een andere waarde voorzien. Als we echter in een .NET Compact Framework-applicatie een variabele volatile willen maken, zal dat resulteren in een foutmelding tijdens de compilatie.

De volgende declaratie zal bijvoorbeeld in Visual Studio.NET 2003 de onderstaande foutmelding opleveren:

```
private volatile bool stopRequested = false;
```

```
The predefined type 'System.Runtime.CompilerServices.IsVolatile' is not
defined or imported
```

In het .NET Compact Framework is een aantal essentiële classes helaas niet geïmplementeerd. De class IsVolatile in de System.Runtime.CompilerServices is daar een voorbeeld van. Om volatile variabelen toch toe te staan kunnen we zelf de betreffende class implementeren door in een project de code uit codevoorbeeld 5 op te nemen.

Nu zal volatile wel herkend worden en wordt in Intermediate Language een indicatie opgenomen dat een variabele als volatile moet worden behandeld. Voor applicaties die alleen binnen het .NET Compact Framework werken is het niet echt noodzakelijk om een variabele als volatile te declareren. In versie 1.0 van het .NET Compact Framework worden alle variabelen als volatile beschouwd. Toch kan het handig zijn de bovenstaande class IsVolatile toch toe te voegen, in verband met de compatibiliteit tussen het .NET Framework en het .NET Compact Framework.

Informatie vanuit een worker thread tonen aan de gebruiker

We staan eens stil bij een uitermate belangrijk onderwerp en een bron van veelgemaakte fouten. In een managed applicatie (dit geldt dus voor zowel .NET Compact Framework-applicaties als voor .NET Framework-applicaties) mag een userinterface-control alleen maar worden gemodificeerd door de thread die de betreffende control heeft gecreëerd. Updaten van controls vanuit andere threads zorgt er voor dat de applicatie instabiel gedrag vertoont; bijvoorbeeld de update niet uitvoert of in het ergste geval de hele applicatie zelfs ophangt. Mochten we bijvoorbeeld onze worker thread in codevoorbeeld 1 zelf informatie in de listbox willen laten schrijven, dan mag dat ***niet*** op de manier zoals getoond in codevoorbeeld 6.

```
private void WorkerThread()
{
  while (! stopRequested)
  {
    // Perform specific action repeatedly ...
    // ... and sleep for a while afterwards to save batteries
    Thread.Sleep(1000);
  }
}
```

Codevoorbeeld 4. Een thread die herhaaldelijk werk uitvoert

```
using System;

namespace System.Runtime.CompilerServices
{
  public sealed class IsVolatile
  {
    private IsVolatile()
    {
    }
  }
}
```

Codevoorbeeld 5. Implementatie van class IsVolatile

AddListBoxEntry (zie codevoorbeeld 1) zal namelijk rechtstreeks, dus vanuit de workerthread, een string aan de listbox toevoegen. In plaats hiervan moeten we een manier vinden waarop de main thread namens de workerthread die string aan de listbox kan toevoegen. Die manier bestaat in de vorm van de Control.Invoke-functie. Deze functie, die in elk userinterface-element is geïmplementeerd, kan gegevens voor een userinterface-element updaten vanuit een workerthread. De update zelf wordt echter uitgevoerd door de thread die het betreffende userinterface-element heeft gecreëerd. In versie 1.0 van het .NET Compact Framework hebben we alleen de beschikking over de synchrone variant Control.Invoke (in het .NET Framework bestaat ook een asynchrone variant Control.BeginInvoke / Control.EndInvoke).

Op het eerste gezicht lijkt het gebruik van Control.Invoke wat moeilijk, omdat deze functie als parameter een 'delegate' verwacht. Een delegate is echter niets anders dan een verkapte pointer naar een functie. Een delegate is 'type safe', wat inhoudt dat alleen functies met de juiste parameters kunnen worden doorgegeven aan Control.Invoke. In het geval van Control.Invoke in het .NET Compact Framework kan alleen een eventhandler-delegate worden gebruikt voor het updaten van een userinterface-control. In codevoorbeeld 7 is te zien hoe Control.Invoke kan worden gebruikt vanuit de workerthread om veilig nieuwe regels aan de listbox toe te voegen.

Helaas is het niet mogelijk parameters vanuit de workerthread door te geven aan de functie die in Control.Invoke wordt gespecificeerd. Vandaar dat we in ons voorbeeld gebruik maken van een extra instance-variabele, die een nieuwe string toegekend krijgt vlak voordat Control.Invoke wordt aangeroepen. Omdat Control.Invoke synchroon werkt en we hier slechts één workerthread gebruiken, is het niet nodig de extra instance-variabele tegen ongewenst overschrijven te beschermen. Meer informatie over

```
private void SimulateWork()
{
  AddListBoxEntry("WorkerThread: Started working");
  while (! stopRequested)
  {
    Thread.Sleep(1000);
  }
  AddListBoxEntry("WorkerThread: Stopped working");
}
```

Codevoorbeeld 6. Niet op deze manier informatie in Listbox plaatsen

```
private void ListBoxEntryHandler(object sender, System.EventArgs e)
{
  AddListBoxEntry(listBoxEntry);
}

private void SimulateWork()
{
  listBoxEntry = "SimulateWork: Entering the workerthread";
  this.Invoke(new EventHandler(ListBoxEntryHandler));
  while (! stopRequested)
  {
    Thread.Sleep(1000);
    listBoxEntry = "SimulateWork: Waking up";
    this.Invoke(new EventHandler(ListBoxEntryHandler));
  }
  listBoxEntry = "SimulateWork: Leaving the workerthread";
  this.Invoke(new EventHandler(ListBoxEntryHandler));
}
```

Codevoorbeeld 7. Aanpassingen / aanvullingen voor veilig updaten van userinterface-controls

het updaten van userinterface-controls is te vinden in het MSDN-artikel: http://msdn.microsoft.com/library/en-us/dntake/html/yctiwy_multithreadingandui.asp. Dit artikel bevat ook een mogelijke oplossing voor het doorgeven van parameters en voor het asynchroon updaten van userinterface-controls.

Afbeelding 3 - in combinatie met codevoorbeeld 7 - laat zien dat het met Control.Invoke mogelijk is informatie vanuit een workerthread in een control te tonen. In afbeelding 3 is echter ook te zien dat het lijkt alsof de workerthread al is afgesloten, terwijl deze nog actief is. Om dit gedrag tegen te gaan is het noodzakelijk beide threads met elkaar te synchroniseren en te weten wanneer de workerthread precies beëindigd wordt. Dit is het onderwerp van het tweede deel van dit artikel dat in het volgende nummer van het .NET Magazine zal verschijnen.

Gebruik van ThreadPool of van Thread

Naast het expliciet gebruik maken van de Thread-class om zelf eigen threads te creëren is het ook mogelijk gebruik te maken van de ThreadPool-class. Zoals de naam van de class aangeeft is ThreadPool een verzameling van thread-objecten. Het gebruik van ThreadPool voor workerthreads in een applicatie biedt een aantal voordelen. Omdat alle threads in een ThreadPool van tevoren door het besturingssysteem worden aangemaakt, zal het gebruiken van een thread uit de ThreadPool erg efficiënt zijn. Zeker voor workerthreads met een korte levensduur kan dit veel performancewinst opleveren. Threads die een langere levensduur hebben, kunnen beter via een Thread-object worden aangemaakt. De reden hiervoor is dat dergelijke threads langdurig een ThreadPool-thread bezet houden. Dit is nadelig als alle threads in een ThreadPool in gebruik zijn, omdat dan een nieuwe ThreadPool-thread moet wachten totdat weer een lege plaats in de queue beschikbaar is. De manier waarop een ThreadPool-thread wordt gecreëerd, werkt wel iets anders. In plaats van een nieuw Thread-object aan te maken, wordt een thread vanuit de ThreadPool gebruikt. Dit gebeurt door simpelweg een WaitCallback-delegate mee te geven aan de ThreadPool.QueueUserWorkItem-methode. Naast het voordeel van minder overhead tijdens het aanmaken van een thread is het ook mogelijk een parameter in de vorm van een object mee te geven aan een ThreadPool-thread. Een beperking is dat de prioriteit van een ThreadPool-thread niet kan worden gewijzigd. Een codevoorbeeld waarin gebruik van een ThreadPool wordt vergeleken met gebruik van Thread-objecten is beschikbaar voor download op de Nederlandse MSDN-website.

Meer threads soms noodzakelijk

Multithreaded applicaties zijn complexer dan applicaties die geen extra threads creëren. Ondanks een toename in complexiteit is het soms noodzakelijk om een applicatie in meer threads op te delen. Al is het alleen maar om gebruikers een betere performance geven. In versie 1.0 biedt het .NET Compact Framework beperkte ondersteuning voor het werken met meer threads. De beschikbare functionaliteit is wel voldoende voor het maken van multithreaded applicaties, hoewel in sommige gevallen teruggegrepen moet worden op Win32-functies, die in de meeste gevallen via P/Invoke te benaderen zijn.

Afbeelding 3. Simultane updates van de listbox vanuit de workerthread en vanuit de main thread

Handig debuggen van PocketPC-applicaties

Om te voldoen aan het 'designed for Microsoft Windows for PocketPC' logo, moet een PocketPC-applicatie voorzien zijn van een zogenaamde *smart minimize*-knop. Deze knop, rechtsboven in de hoek geplaatst op de titelbar van een PocketPC-applicatie zorgt ervoor dat een applicatie niet wordt afgesloten, maar op de achtergrond actief blijft. Het oorspronkelijke idee hierachter is dat gebruikers van PocketPC's geen notie hoeven te hebben van het afsluiten van applicaties. Tijdens debuggen van een applicatie is het echter heel vervelend als een applicatie niet kan worden afgesloten. De debugger blijft immers actief totdat een applicatie is afgesloten. Om toch aan de logo-eisen te voldoen en tegelijkertijd debuggen van applicaties eenvoudiger te maken, is het handig gebruik te maken van conditionele compilatie. Vandaar dat in onze voorbeeldapplicatie de volgende code wordt gebruikt om al dan niet een smart minimize-knop te tonen, afhankelijk van het feit of we aan het debuggen zijn of niet:

```
#if DEBUG
      MinimizeBox = false;
#else
      MinimizeBox = true;
#endif
```

Afbeelding 4. OK-knop of Smart Minimize button

Meer informatie over het logoprogramma:
http://www.microsoft.com/whdc/getstart/testing.mspx

Maarten Struys is werkzaam bij PTS Software bv, dat zich specialiseert in technische systeemontwikkeling en in die hoedanigheid veel expertise heeft opgebouwd op het gebied van embedded oplossingen met behulp van Windows CE en Windows XP Embedded. Maarten is .NET Compact Framework MVP en Windows Embedded Evangelist bij PTS Software bv. Hij is geregeld als spreker te beluisteren tijdens MSDN WebCasts. Maarten schrijft veelvuldig over het .NET Compact Framework op zijn eigen website: www.dotnetfordevices.com.

Nuttige internetadressen

Webcast over multithreading:
http://msevents.microsoft.com/cui/WebCastEventDetails.aspx?EventID=1032257390
http://msevents.microsoft.com/CUI/WebCastEventDetails.aspx?EventID=1032265119
Development center:
http://msdn.microsoft.com/smartclient/understanding/netcf/
http://msdn.microsoft.com/library/en-us/dnanchor/html/NETCompactFrame.asp