Ed van de Pitte en Pepijn Kramer

Het bouwen van een responsive UI

METHODEN EN TECHNIEKEN OM EEN UI BETER TE LATEN REAGEREN

Veel gebruikers zullen bekend zijn met het fenomeen dat de user-interface van een programma de gebruiker niet kan bijhouden of, erger nog, helemaal niet reageert op pogingen om een (stop) knop in te drukken. In het ergste geval kan de taskmanager aangeven dat het programma niet meer reageert. Dit artikel heeft tot doel een aantal inzichten en methodieken aan te geven waarmee dit soort problemen in applicaties voorkomen kan worden.

Voordat we in de techniek duiken is het goed kort stil te staan bij wat we met een responsive user-interface bedoelen. We definiëren een responsive user-interface als een user-interface die commando's van een gebruiker uitvoert, maar daarbij blijft reageren op (nieuwe) commando's.

Oorzaken van non-responsiveness

Er zijn enkele hoofdoorzaken waardoor een user-interface voor een gebruiker niet meer responsive is.

- Logische fout. Door een bug of ontwerpfout in het state-model van de applicatie blijven alle user-interface-controls op het scherm oncontroleerbaar.
- Expliciet. Er zijn gevallen waarin expliciet gekozen is voor het tonen van een dialoogvenster waar je 'niets' mee kunt. In deze situatie zal de applicatie nog wel op systeemberichten, zoals WM_PAINT, reageren.
- Ontwerpfout. De windows message-queue is geblokkeerd en het systeem kan geen input meer verwerken, maar ook geen beeld-updates meer doen. Als de message-queue geblokkeerd is, moeten nieuwe berichten wachten totdat de message-queue weer vrijgegeven is. Hierdoor is het niet meer mogelijk buttons te activeren, menu's te selecteren of het venster te verslepen. Als deze situatie met korte tussenpozen voorkomt, zal de gebruiker een user-interface als traag ervaren.

In dit artikel zullen we ons focussen op oorzaken van de message-queue-blokkade en mogelijkheden om dit te voorkomen.

De message-queue

Iedere Windows-applicatie met een user-interface heeft een thread die de windows-messages afhandelt, ook wel de main-thread genoemd. Als het een GUI-applicatie betreft, dan is deze thread verantwoordelijk om berichten (zoals mouse-moves, button-press of paint-berichten) van de gebruiker of het systeem af te handelen. Deze berichten worden door het systeem met een 'postmessage' in de message-queue gezet en de applicatie zal deze berichten een voor een afhandelen. Als het afhandelen van een bericht lang duurt, zullen de overige berichten in de message-queue moeten wachten totdat ze worden afgehandeld. Om een applicatie responsive te houden, is het dus belangrijk berichten zo snel mogelijk af te handelen zodat de message-queue niet verstopt raakt. Het is in dit kader van belang om te weten dat Windows met 'postmessage' een bericht in de message-queue plaatst en dan direct door gaat. Als een 'sendmessage' uitgevoerd wordt, zal het bericht synchroon afgehandeld worden zonder eerst in de message-queue geplaatst te worden. Statusberichten (zoals het indrukken van een knop) worden door Windows altijd met een 'postmessage' bekendgemaakt, terwijl instructies (zoals het disablen van een knop) door het systeem met een 'sendmessage' afgehandeld worden. De basisregel is eigenlijk heel eenvoudig, zorg dat je nooit veel tijd besteedt aan het afhandelen van een windows-message of een event. Op het moment dat er langdurige acties plaatsvinden als reactie op een event, zijn er de volgende mogelijkheden om te voorkomen dat de message-queue geblokkeerd blijft:

- Application.DoEvents
- Afhandeling in andere thread
- Meerdere applicaties (meerdere main-threads).

Application.DoEvents

Een eenvoudige manier om wachtende berichten in de message-queue af te handelen is de applicatie instructie te geven om de wachtende berichten af te gaan handelen. Dit is vaak wenselijk om een control opnieuw te tekenen, terwijl deze wordt gevuld. Dit is eenvoudig voor elkaar te krijgen door het commando 'Application.DoEvents' op te nemen in je event-handler. Als je dat doet, dan lijkt je programma weer helemaal responsive, maar meestal zal het gebruik van Application.DoEvents juist nieuwe problemen opleveren. Dit komt omdat nieuwe messages afgehandeld worden terwijl je 'huidige' code even stil staat. De afhandeling van deze messages kunnen (en zullen) conflicten opleveren, zoals het aanroepen van andere event-handlers die de state van je applicatie veranderen zonder dat de originele event-handler daar weet van heeft. Een eenvoudig voorbeeld van een probleem dat optreedt bij het gebruik van Application.DoEvents is het feit dat de 'close' van je applicatie afgehandeld kan worden. Hierdoor zal je event-handler uitgevoerd worden, terwijl je form al disposed is. De applicatie wordt dan nog 'in leven gehouden' totdat de event-handler klaar is. Dergelijke problemen zijn met veel extra vlaggen, of met een message-filter wel op te lossen, maar in onze opinie kun je beter kijken naar andere oplossingen om je applicatie responsive te houden.

Enkele voordelen van deze oplossing:

- Zeer snel en eenvoudig in te bouwen.
- UI-controls worden 'life' up-to-date gehouden, omdat paint-events afgehandeld worden.

Enkele nadelen:

- Andere systeemberichten worden ook afgehandeld (zoals close-event).
- Synchronisatiemechanismen (zoals locks) werken niet, omdat alle acties vanuit dezelfde thread worden uitgevoerd.
- Geen directe controle over de volgorde waarin zaken plaatsvinden. Het is maar net welke message er als volgende in de queue staat. Dit kan een probleem zijn voor correct gedrag van een systeem.

```
private void button_Click(object sender, System.EventArgs e)
{
   WaitCallback operation = new WaitCallback(this.LengthyOperation);
   ThreadPool.QueueUserWorkItem(operation,0);
}

// Function that is executed in separate thread.
private void LengthyOperation(Object param)
{
   for (int iVal=0;iVal<20000;iVal++)
   {
      InsertDataInListBox(0,iVal.ToString());
   }
}

// Add data to listbox regardless of the thread it is called from.
delegate void InsertDataInListBoxDelegate(int Index, string Data);
public void InsertDataInListBox(int Index, string Data)
{
   // Check if called from main-thread.
   if (this.InvokeRequired)
   {
      // Called from other thread.
      this.Invoke(
         new InsertDataInListBoxDelegate(InsertDataInListBox),
         new object[]{Index, Data});
   }
   else
   {
      // Perform operation. This is called from the main-thread.
      this.listBox1.Items.Insert(Index,Data);
   }
}
```

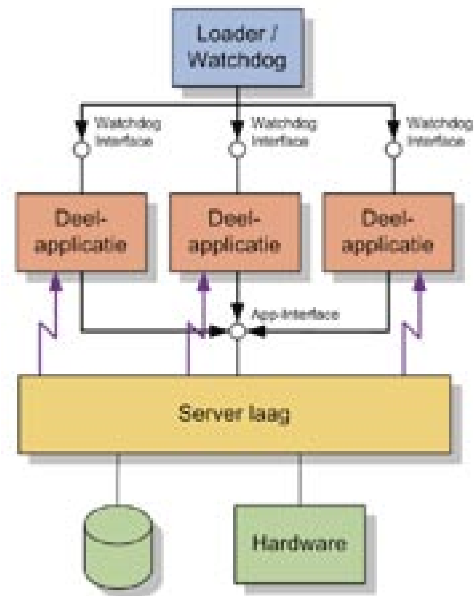
Codevoorbeeld 1. Aparte thread + Invoke

Afhandeling in andere thread

De beste oplossing om de message-queue te ontlasten, is de afhandeling van messages in een andere thread dan de main-thread te laten plaatsvinden. In .NET gaat dit zeer gemakkelijk met de threadpool of de threadclass die delegates asynchroon uitvoeren. Helaas moet hierbij rekening worden gehouden met de regel dat instanties van objecten die afgeleid zijn van System. Windows.Forms.Control (alle UI-controls dus) alleen benaderd mogen worden vanuit de thread die de instantie aangemaakt heeft. Om vanuit een andere thread acties uit te voeren op de main-thread, moet je gebruikmaken van de Invoke- of BeginInvoke-functies. In .NET Framework 1.1 kon je nog wel zondigen tegen deze regel, maar in .NET Framework 2.0 treden exceptions op als je functies van een UI-control aanroept vanuit een andere thread.

Met Invoke wordt een functie door een ‘PostMessage’ achteraan in de message-queue geplaatst en uitgevoerd als het bericht aan de beurt is. Je komt pas terug uit de Invoke als de functie is uitgevoerd. Om te controleren of een functie via Invoke aangeroepen moet worden, kun je gebruik maken van de ‘InvokeRequired’-functie. InvokeRequired controleert of de thread van waaruit de functie aangeroepen wordt, gelijk is aan de main-thread (dit wordt door middel van de GetWindowsThreadProcessID API-call gedaan). Codevoorbeeld 1 geeft een voorbeeld van het gebruik van de Invoke-functie.

Codevoorbeeld 1 voegt items aan een de listbox toe door een Invoke. Hoewel Invoke onder water een PostMessage uitvoert, waardoor de message-queue belast wordt, krijgt het systeem toch nog genoeg gelegenheid om overige berichten van de applicatie af te handelen. De situatie zou slechter zijn wanneer je in plaats van Invoke een BeginInvoke gebruikt. Bij BeginInvoke wordt het bericht in de message-queue geplaatst en gaat de functie die de BeginInvoke aanriep, door zonder op het resultaat te wachten. Als dit heel snel achter elkaar gebeurt, kan de message-queue verstopt raken, omdat de overige messages dan weinig gelegenheid krijgen om afgehandeld te worden. Om een applicatie responsive te houden moet je dus steeds goed in gedachten houden wat de consequenties zijn voor de message-queue. Zo zou codevoorbeeld 1 aangepast kunnen worden, zodat de Invoke een (gevulde) collectie aan de listbox toevoegt in plaats van allemaal losse elementen.



Afbeelding 1. Architectuur met deelapplicaties

Enkele voordelen van meerdere threads:

- Parallel uitvoeren van commando’s is mogelijk zonder belasting van de event-handler-functie.
- Eenvoudig in te bouwen vanwege goede standaardfaciliteiten van .NET.
- Hoge performance.

Enkele nadelen:

- UI-controls moeten door middel van Invoke of BeginInvoke aangeroepen worden.
- Bescherming van shared resources kan nodig zijn.
- Synchronisatiemechanismen kunnen nodig zijn.

Verscheidene processen

In plaats van één applicatie en dus ook maar één main-thread, kun jij je applicatie ook opdelen in verscheidene deelapplicaties die elk een eigen main-thread kennen. Deze oplossing komt voor bij zeer grote applicaties (met name applicaties die verscheidene monitoren gebruiken zoals medische systemen). Vaak heb je dan ook een los programma nodig dat dienst doet als ‘loader’ en ‘watchdog’ voor de deelapplicaties. Daarnaast zul je bijna altijd een gemeenschappelijke laag maken die de data-sharing, of resource-locking afhandelt tussen de diverse processen; zie afbeelding 1.

Enkele voordelen van deze oplossing:

- Deelapplicaties kunnen door verschillende groepen parallel worden ontwikkeld.
- De testbaarheid is groter, omdat de deelapplicaties ‘standalone’ getest kunnen worden.
- Hogere robuustheid. Als een deelapplicatie crasht, dan kunnen de andere delen van het systeem doorgaan (de watchdog kan dit detecteren en de deelapplicatie opnieuw starten).

Enkele nadelen:

- Er is een noodzaak om een loader te maken.
- Er is een synchronisatiemechanisme tussen de deelapplicaties nodig.
- Communicatie over procesgrenzen heen is langzamer dan in-process communicatie.

Snelle input

Er zijn situaties waarbij de gebruiker sneller events genereert dan het systeem aan kan. Een voorbeeld hiervan is een slider-control die direct acties uitvoert op een foto. Iedere aanpassing van de foto kan enige tijd kosten en toch wil de gebruiker tussentijdse resultaten zien. Om dergelijke problemen op te lossen kun je de afhandeling van de slider-events in een andere thread laten plaatsvinden, waarbij je pas op nieuwe waarden reageert als de vorige handeling klaar is. Codevoorbeeld 3 laat zien hoe je een thread aanmaakt die reageert op events van een slider. Als je dit voorbeeld uitprobeert, zul je merken dat je de slider heel soepel kunt blijven bewegen, terwijl de updates op hun eigen snelheid worden uitgevoerd. Dezelfde oplossing kun je ook gebruiken voor controls die 'pulsen' geven (zoals externe input-devices of een spincontrol). In dat geval moet je steeds de verandering van de waarde bij de bestaande waarde optellen voordat de 'set'-functie van de WaitHandler wordt aangeroepen. Daarbij is het ook raadzaam het veranderen van de waarde op een thread-safe-manier te doen; bijvoorbeeld door Interlocked.Increment, locks of mutexes te gebruiken.

Tekenen van componenten

Het (her)tekenen van controls vindt altijd plaats op de main-thread van je applicatie. Als je de standaard UI-controls gebruikt, dan zal dit zelden aanleiding geven tot een merkbare belasting van de main-thread. Deze situatie kan echter veranderen als je eigen grafische UI-controls maakt die veel tekenwerk nodig hebben. Doordat dit tekenwerk meestal als reactie op een paint-event wordt uitgevoerd, kan ook dit de afhandeling van de message-queue vertragen. Om deze belasting laag te houden, kunnen controls die veel grafische output genereren hun tekenwerk uitvoeren op een aparte thread, waarbij het tekenwerk uitgevoerd wordt op een memory-bitmap. Zodra de memory-bitmap gevuld is, kan deze als reactie op een paint-event die met Invalidate gegenereerd wordt, getekend worden in één operatie. Bij UI-controls van enige schermomvang is het verstandig alleen dat deel van de UI-control opnieuw te laten tekenen dat is veranderd. Dat deel kun je als argument aan de Invalidate doorgeven. De property ClipRectangle van de PaintEventArgs van de paint event handler bevat dan het deel dat opnieuw getekend moet worden. Codevoorbeeld 3 laat zien hoe je in een aparte thread een memory-bitmap kunt 'voorbereiden' en laat tekenen.

COM/OCX

In de technische automatisering wordt veel gebruikgemaakt van COM-componenten om een systeem te besturen. Opdrachten op de user-interface worden hierbij door een COM-component uitgevoerd. Ook hier is het verstandig om de call naar die component op een andere thread uit te voeren. Dit omdat je niet weet hoe lang die component erover gaat doen, of als de component op een andere pc draait (DCOM), hoe lang de communicatie met die component nu echt gaat duren. COM-componenten kunnen in een single-threaded appartment (STA) of in een multi-threaded appartment (MTA) leven. OCX'en kunnen alleen maar in een STA-omgeving leven. Indien een COM-component in de STA-omgeving is aangemaakt, zullen alle calls en COM-events met behulp van de message-queue worden afgehandeld. De afhandeling van events van COM-componenten die in een MTA leven, gaat zonder tussenkomst van de message-queue. Bij OCX'en heb je geen keuze, maar reguliere COM-componenten (zeker COM-servers) kunnen het beste aangemaakt worden in een MTA-omgeving. Dit laatste zal in een UI-applicatie echter niet zo maar lukken, aangezien een UI-applicatie altijd STA zal zijn, zelfs als je de appartmentstate van je UI op MTA zet. Als je een aparte thread opstart, de appartmentstate van die thread op MTA initialiseert, en dan de COM-componenten aanmaakt, dan leven die COM-componenten wel in een MTA. Belangrijk bij het gebruik van COM-componenten is, dat jij je er van bewust bent dat

```
private Thread         _sliderThread;
private AutoResetEvent _newSliderDataAvailable;
private AutoResetEvent _stopSlider;
private int            _newSliderValue;

private void Form1_Load(object sender, System.EventArgs e)
{
  _newSliderDataAvailable = new AutoResetEvent(false);
  _stopSlider          = new AutoResetEvent(false);
  _sliderThread        = new Thread(
    new ThreadStart(this.SliderHandlingThread));
  _sliderThread.Start();
}

// Function that runs in its own thread. It handles
// time-consuming processing of slider events.
private void SliderHandlingThread()
{
  AutoResetEvent[] waitHandles = {_stopSlider,
                                  _newSliderDataAvailable};

  // Wait for notification that data is available.
  // WaitAny returns index of event that occurred. 0 = stopslider.
  while (0 != WaitHandle.WaitAny(waitHandles))
  {

  // Get slider value and perform the operation.
    int val;
    Interlocked.Exchange (ref val, _newSliderValue);
    // Perform lengthy operation.
    ...
    // Update the UI.
    UpdateUIForSliderChange(val);
  }
}

// Function to update UI with result of slider-operation.
delegate void UpdateUIForSliderChangeDelegate(int val);
public void UpdateUIForSliderChange(int val)
{
  // Check if called from main-thread.
  if (this.InvokeRequired)
  {
    // Invoke needed.
    this.BeginInvoke(new UpdateUIForSliderChangeDelegate(
        UpdateUIForSliderChange ), new object[]{ val });
  }
  else
  {
    // Perform operation. In this case, just change some color.
    this.listBox1.BackColor = Color.FromArgb(val,val,val);
  }
}

// Slider changed. Perform the operation via handling thread.
private void hScrollBar2_Scroll(object sender, ScrollEventArgs e)
{
  Interlocked.Exchange (ref _newSliderValue, e.NewValue);
  // Notify thread to perform job.
  _newSliderDataAvailable.Set();
}

// Form closing. Stop the slider event handling thread.
private void Form1_Closing(object sender, CancelEventArgs e)
{
  _stopSlider.Set();
}
```

Codevoorbeeld 2. Slider-updates

```
private Bitmap         _displayBitmap;
private Bitmap         _workBitmap;
private Thread         _backGroundPrepareThread;
private AutoResetEvent _newDataAvailable;
private AutoResetEvent _stopBackGroundPrepare;
private int            _newValue = 0;

public UserControl1()
{
   InitializeComponent();

   _displayBitmap = new Bitmap (this.Width,this.Height, Format24bppRgb);
   _workBitmap = new Bitmap (this.Width,this.Height, Format24bppRgb);

   _backGroundPrepareThread = new Thread (new ThreadStart(
                       this.BackgroundPrepareThread));
   _newDataAvailable = new AutoResetEvent(true);
   _stopBackGroundPrepare = new AutoResetEvent(false);
   _backGroundPrepareThread.Start();
}

// Prepare images in memory background.
private void BackgroundPrepareThread()
{
   AutoResetEvent[] waitHandles =
      {_stopBackGroundPrepare, _newDataAvailable, };

   // Wait for notification that data is available.
   while (0 != WaitHandle.WaitAny(waitHandles))
   {
      // Swap display and work bitmap
      lock( _displayBitmap )
      {
         Bitmap tmp = _displayBitmap;
         _displayBitmap = _workBitmap;
         _workBitmap = tmp;
      }
      int val = 0;
      Interlocked.Exchange(ref val, _newValue);

      // Get graphics instance from the image.
      Graphics offScreenDC = Graphics.FromImage(_workBitmap);
      offScreenDC.Clear(this.BackColor);

      // Perform all painting operation on the offScreenDC.
         ...
      // Generate paint-message on main-thread.
      this.Invalidate();
   }
}

// Property to set new value
public int Value
{
   get { return _newValue; }
   set
   {
      _newValue = value;
      _newDataAvailable.Set();
   }
}

// Paint memory bitmap on screen. For example reasons, the whole
// bitmap is blitted on screen in stead of only a portion.
private void UserControl1_Paint(object sender, PaintEventArgs e)
{
   // Paint the "actual bitmap".
   lock( _displayBitmap )
   {
      e.Graphics.DrawImageUnscaled(_displayBitmap,0,0);
   }
}

private void Form1_Closing(object sender, CancelEventArgs e)
{
   _stopBackGroundPrepare.Set();
}
```

Codevoorbeeld 3. Tekenen in een achtergrond thread

COM-componenten je responsiveness kunnen beïnvloeden. De applicatie-appartmentstate kan ook de performance van je COM-component beïnvloeden, omdat de event-afhandeling van COM-events via de message-queue de COM-component zelf ook vertraagt.

Parallel uitvoeren van gebruikeracties

Veel applicaties gaan nog altijd uit van het model dat een gebruiker slechts één taak tegelijk kan uitvoeren. Als een gebruiker een actie start, wordt door middel van het disabled maken van UI-controls afgedwongen dat andere acties niet meer mogelijk zijn. Als de achterliggende actie (die je veelal als 'use-case' kunt aanmerken) in een aparte thread wordt afgehandeld, kun je de gebruiker de mogelijkheid bieden nieuwe acties te starten, terwijl de vorige acties nog bezig zijn. In omgevingen waar een strakke user-workflow is gedefinieerd, zal dat niet direct nodig zijn, maar in andere omgevingen kan het zelfs een harde requirement zijn dat verscheidene acties parallel kunnen worden uitgevoerd. Zodra een gebruiker verscheidene acties tegelijk kan uitvoeren, werkt het niet meer om de UI-controls te disablen bij het starten van de actie. Andere acties, die parallel uitgevoerd kunnen worden, zullen mogelijk weer andere UI-controls enabled of disabled willen zien. Om de gebruiker een user-interface aan te bieden, waarbij de UI-controls niet staan te knipperen van enabled naar disabled en vice versa, is het wenselijk om de gebruikersacties die parallel uitgevoerd kunnen worden in een state-model te modelleren, waarbij je er voor zorgt dat dit state-model het enabled/disabled/showgedrag van de UI-controls afhandelt. Dit state-model kan dan in één functie geïmplementeerd worden wat het onderhoud zal verlagen.

Meten van responsiveness

De belasting van de message-queue is een belangrijke factor voor de responsiveness van een UI-applicatie. Het verkeerd implementeren van een event-handler kan er al voor zorgen dat de gehele responsiveness van een UI verloren gaat. Daarom is het belangrijk de belasting van de message-queue gedurende de gehele ontwikkeling van de applicatie in de gaten te houden, want fouten tegen de responsiveness die pas in de testfase worden gevonden, zullen meestal niet meer gecorrigeerd worden. Helaas biedt Windows geen standaardvoorziening om de message-queue-belasting te meten en daarom zal er zelf een mechanisme in je applicatie moeten worden opgenomen. Dit kan door een thread te maken die af en toe een message naar je eigen applicatie 'post' (met BeginInvoke). Vervolgens meet je de tijd tussen het verzenden en het ontvangen van je bericht. Als je deze waarde in een performance-counter schrijft, kun je de responsiveness met behulp van PerfMon eenvoudig in de gaten houden.

Bewust van belasting

In dit artikel hebben we een paar factoren genoemd die de responsiveness van een systeem bepaalt. Kern van de zaak is dat de message-queue en de main-thread niet te veel belast mogen worden. Een ontwikkelaar dient zich bewust te zijn van de consequenties van zijn of haar acties op de message-queue en de main-thread. Aangezien het veel inspanning kan kosten om problemen met de responsiveness later nog te corrigeren, zal er gedurende de implementatie en (integratie-)testfase van het ontwikkelproces gecontroleerd moeten worden of de responsiveness van het systeem nog in orde is. Hierbij zijn performance-counters een perfect hulpmiddel.

Ed van de Pitte werkt als softwarearchitect bij Atos Origin Technical Automation (www.atosorigin.com) waarbij hij vooral betrokken is bij grote applicaties om machines te besturen. Voor vragen en opmerkingen is Ed bereikbaar via Ed.vandePitte@AtosOrigin.com

Pepijn Kramer werkt als softwarearchitect bij FEI (www.fei.com) waar hij werkt aan alle aspecten van de multi-tier platformarchitectuur. Pepijn is bereikbaar via Pepijn.Kramer@FEI.com

Referenties

http://www.developer.com/design/article.php/2244501
http://www.codeproject.com/csharp/begininvoke.asp
http://msdn.microsoft.com/library/en-us/dnvs05/html/threadinginvb2005.asp
http://msdn.microsoft.com/msdnmag/issues/03/02/multithreading/toc.asp