Maurice de Beijer

Een oplossing voor langlopende events

WORKFLOW FOUNDATION-ACTIVITEITEN ONTWIKKELEN

Workflow Foundation (WF) is één van de onderdelen van het .NET 3.0 Framework. Waar de andere onderdelen, te weten Windows Communication Foundation (WCF) en Windows Presentation Foundation (WPF) nieuwe manieren zijn om iets te doen wat we al langer deden, is dat met Workflow Foundation niet het geval. Met Workflow Foundation is een heel nieuwe stijl van ontwikkelen mogelijk, namelijk die van de langlopende en gebeurtenisgestuurde applicatie.

Uiteraard worden er al veel langer event-driven applicaties gemaakt, dat is niets nieuws, maar meestal zijn deze kortlopend, dus een gebruiker start een applicatie, doet iets waarop de applicatie reageert en de applicatie wordt weer afgesloten. Een volgende keer dat de applicatie gestart wordt, vormt een op zichzelf staand geval, waarbij wel de data, dus de status, bewaard zijn maar de gebruiker zelf weer nieuwe acties moet starten. Er waren natuurlijk wel applicaties die ergens bewaarden dat ze op een bepaalde gebeurtenis wachten en dan zelf op de één of andere manier gingen kijken of dat al gebeurd was, maar dat moest dan zelf in een applicatie geprogrammeerd worden.
Met Workflow Foundation wordt dat allemaal anders. Nu hoeft een applicatie niet langer zelf te controleren of iets gebeurd is, maar kan deze applicatie aan de workflow-runtime laten weten dat er op een bepaalde gebeurtenis gewacht wordt en dan in een soort slaapstand terecht komen. Zodra de gebeurtenis optreedt, kan de workflow-runtime de applicatie laten weten dat dit het geval is en kunnen we verder gaan. Dit kan zelfs maanden later zijn, de workflow-runtime hoeft hierbij niet constant te draaien. Uiteraard is de workflow zelf dan ook niet actief en kan hij geen events ontvangen, maar de huidige status blijft gewoon bewaard. Het kan zelfs op een heel andere machine doorgaan. Dit is nogal anders dan we in het verleden gewend waren en de applicatie heeft zelf een andere naam gekregen en heet nu een workflow.
De workflow-runtime is het onderdeel van Workflow Foundation dat zorgt dat alles kan draaien. Om dat te doen bestaat deze uit een aantal runtime-services, sommige verplicht maar vele optioneel, die dingen voor de ontwikkelaar regelen. De workflow-runtime is hierbij de spin in het web die alles aan elkaar knoopt. Een workflow zelf is de eenheid die gestart kan worden om een bepaalde functionaliteit te bieden. Deze workflow wordt via de runtime, om precies te zijn via de WorkflowLoaderService, aangemaakt. Elke workflow bestaat weer uit een aantal activiteiten. Deze activiteiten zijn het best te zien als herbruikbare klassen of functies die in meer workflows gebruikt kunnen worden. Van deze activiteiten worden er standaard een aantal meegeleverd, maar men kan zelf ook nieuwe maken. Runtime-services zijn die objecten die aan de runtime toegevoegd worden om allerlei diensten voor workflows te verrichten. Zo zijn er services die een workflow laden, uitvoeren of op schijf kunnen bewaren. Het is ook mogelijk zelf runtime-services te maken en toe te voegen, bijvoorbeeld om te communiceren met onderdelen buiten de workflow zelf.

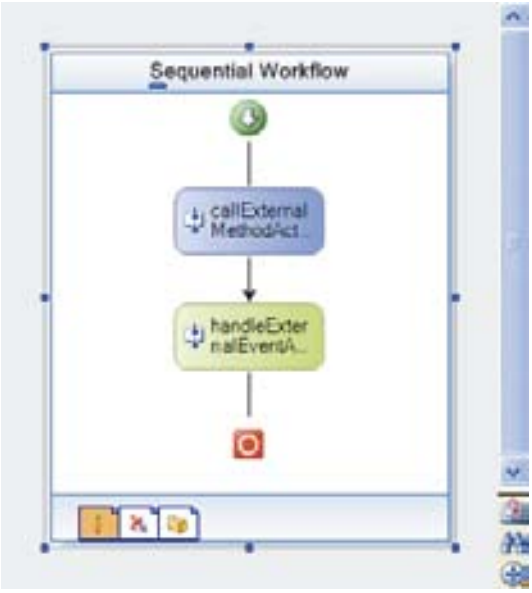
```
private void codeActivity1_ExecuteCode(object sender, EventArgs e)
{
   Console.WriteLine("Wat is de volgende regel?");
   string input = Console.ReadLine();
}
```

Codevoorbeeld 1.
Een slecht idee, deze workflow blijft binnen het uitvoeren van een activiteit wachten

Waarom zelf activiteiten ontwikkelen?

Standaard komt Workflow Foundation al met een hele lijst van activiteiten die men kan gebruiken om workflows te ontwikkelen. Als er standaard al een lijst van activiteiten is, waarom zouden we dan zelf nieuwe activiteiten moeten ontwikkelen? Per slot van rekening, als we eigen code in een workflow willen uitvoeren, kunnen we eenvoudigweg een CodeActivity toevoegen en in het ExecuteCode-event de nodige code schrijven. Deze code is dan nog niet herbruikbaar, maar als we die code in een aparte klasse zetten en die aanroepen, hoeven we maar twee regels code te schrijven, niet echt een belemmering zou je zeggen. Deze aanpak heeft echter een fundamentele beperking door wat we in het ExecuteCode-event kunnen doen.

Kijk eens goed naar codevoorbeeld 1, in dit eenvoudige voorbeeld doe ik een Console.ReadLine() binnen de ExecuteCode-activiteit. Op het eerste gezicht lijkt daar misschien niets mis mee te zijn, maar het heeft een aantal grote problemen tot gevolg.
Het eerste grote probleem is de schaalbaarheid. Standaard zal de workflow-runtime een beperkt aantal workflow-activiteiten tegelijk uitvoeren. Dit standaard aantal is afhankelijk van het aantal processors in de machine, vier workflows per processor in het geval van een multiprocessor-machine en anders vijf workflows. Het aantal activiteiten kan eventueel verhoogd worden, maar is altijd beperkt tot de grootte van de ThreadPool die standaard 25 threads per processor bevat. Als we een workflow met deze activiteit starten, zal alles gewoon lijken te werken, maar aangezien de activiteit nooit aan de runtime laat weten dat deze niks aan het doen is, blijft deze een workflow-thread blokkeren. Als we nu deze workflow zes keer starten, zullen de eerste vijf kopieën gaan lopen, maar zal de zesde nooit aan bod komen. Dit is uiteraard geen gewenst gedrag, omdat geen enkele andere workflow meer door kan gaan terwijl de eerste vijf workflows eigenlijk helemaal niets aan het doen zijn.
Maar er is nog een veel groter probleem. Een workflow kan langlopend zijn, omdat de workflow-runtime en niet de workflow zelf in controle is over wat er wanneer gebeurt. Dit geeft de runtime de mogelijkheid de workflow op schijf te bewaren als dat de runtime beter uitkomt. Deze controle kan de workflow-runtime alleen heb-



Afbeelding 1.
Een workflow met een activiteit om een ExternalDataExchangeService aan te roepen

ben als een functieaanroep binnen een activiteit niet te lang duurt. Zolang een functie loopt, heeft de workflow-runtime namelijk geen enkele weet van wat er intern gebeurt en kan de workflow niet op schijf bewaren, het zogenaamde dehydraten. Pas als alle workflows opgeslagen zijn, kan de runtime afgesloten worden en later weer verder gaan waar deze was gebleven.

Belangrijk: Een workflow-activiteit mag nooit een thread-blokkerend statement bevatten!

Nu is dit een leuke regel, maar vroeger of later moeten we binnen een workflow toch kunnen wachten op een gebeurtenis. Als we dat niet kunnen, zou dat wel erg veel beperkingen met zich meebrengen en is het concept langlopende workflow ook niet echt nuttig. Dit dilemma kunnen we oplossen door gebruik te maken van een ExternalDataExchangeService en een combinatie van een CallExternalMethodActivity en een HandleExternalEventActivity.

Het probleem van de blokkerende ExecuteCode in de eerste CodeActivity is nu opgelost en als we de workflow zes keer starten, zal die ook echt zes keer om input vragen. Overigens heb ik de ExternalDataExchangeService hier zeer simpel geïmplementeerd en gebruik ik nog steeds een thread uit de ThreadPool, zodat we bij 25 workflows nog steeds vastlopen. Daarnaast zou de ReadLineService na een herstart van de workflow-runtime nooit meer om input vragen, dus ook op het gebied van herstartbaarheid laat deze oplossing nog wel te wensen over, maar het geeft in ieder geval aan wat het principe achter deze oplossing is.

Hoewel deze oplossing werkt, en met een robuustere implementatie van de ReadLineService betrouwbaar zou zijn, laat de oplossing nog wel te wensen over. Zo zal er steeds een combinatie van twee activiteiten op een workflow gebruikt moeten worden en is er op het gedrag van de activiteit binnen de workflow geen invloed uit te oefenen. Verder is de service best ingewikkeld en moet er zowel een interface, de implementatie hiervan, als een specifieke ExternalDataEventArgsklasse gemaakt worden. Het gevolg van deze beperkingen is dat men al snel geneigd zal zijn om zelf activiteiten te gaan ontwikkelen.

Zelf een activiteit maken

Het is in principe niet moeilijk om zelf een activiteit te maken. Het probleem is echter dat er nogal veel bij komt kijken om een activiteit te maken die onder alle omstandigheden goed volgens de regels van Workflow Foundation blijft werken. Een extra complicerende factor hierbij is dat de principes achter Workflow Foundation voor veel programmeurs nieuw zijn. Waar je normaal gewend bent om een object aan te maken en een functie hierop aan te roepen, moet je er bij Workflow Foundation steeds voor zorgen dat de workflow-runtime in controle blijft. In veel gevallen moet je dingen dus in een indirecte manier doen door de workflow-runtime te vertellen wat je wilt doen in plaats van dit direct te doen. Door dit laatste principe is de workflow-runtime in staat om de workflow naar schijf te bewaren en op een later tijdstip weer verder te laten gaan. Laten we eens beginnen met het lezen van één regel tekst in een eigen activiteit te stoppen.

In codevoorbeeld 3 kunnen we zien hoe deze eenvoudige workflow-activiteit gemaakt wordt. Ondanks dat dit maar een eenvoudig voorbeeld is, komt hier al een behoorlijk aantal nieuwe concepten voor. Zo gaan we nu gebruikmaken van een ActivityExecutionContext-parameter die aan de activiteit Execute meegegeven wordt. Deze

```
using System;
using System.Threading;
using System.Workflow.Activities;
using System.Workflow.Runtime;

namespace WorkflowConsoleApplication1
{
  [ExternalDataExchange]
  public interface IReadLineService
  {
    void ReadLine(string prompt);
    event EventHandler<LineReadEventArgs> LineRead;
  }

  public class ReadLineService : IReadLineService
  {
    public void ReadLine(string prompt)
    {
      Guid instanceId = WorkflowEnvironment.WorkflowInstanceId;
      ThreadPool.QueueUserWorkItem(delegate(object state)
      {
        Console.WriteLine(prompt);
        string input = Console.ReadLine();

        LineReadEventArgs e =
          new LineReadEventArgs(instanceId, input);
        LineRead(null, e);
      });
    }

    public event EventHandler<LineReadEventArgs> LineRead;
  }

  [Serializable]
  public class LineReadEventArgs : ExternalDataEventArgs
  {
    private string _input;
    public string Input
    {
      get { return _input; }
      set { _input = value; }
    }

    public LineReadEventArgs(Guid instanceId, string input)
      : base(instanceId)
    {
      Input = input;
    }
  }
}
```

Codevoorbeeld 2. Invoer lezen via een ExternalDataExchangeService

ActivityExecutionContext is één van de belangrijkere klassen binnen Workflow Foundation en ik zal er later nog uitgebreid op terugkomen. Verder maken we nu gebruik van een WorkflowQueuingService waar we een WorkflowQueue op aanmaken die we dan weer gebruiken om een event-handler toe te voegen en de ingevoerde regel tekst te versturen. Dit is vergelijkbaar, zij het wat eenvoudiger opgezet, met de eerder gebruikte HandleExternalEventActivity die achter de schermen ook een workflow-queue gebruikt om de data van het event te ontvangen.

Belangrijk: Communicatie tussen de workflow runtime en activiteiten gaat op basis van workflow-queues.

Verder zien we dat de Execute()-functie een ActivityExecutionStatus met als waarde Executing teruggeeft. Dit geeft de controle weer terug aan de workflow-runtime. maar geeft aan dat deze activiteit nog steeds bezig is en dat de volgende activiteit binnen dezelfde tak nog niet gestart mag worden.
In codevoorbeeld 4 is de bijbehorende runtime-service te zien. Deze is nu veel eenvoudiger en bestaat uit een enkele klasse die de data zo direct mogelijk naar de workflow stuurt. Nu kan ik me goed

```
using System;
using System.Workflow.ComponentModel;
using System.Workflow.Runtime;

namespace WorkflowConsoleApplication1
{
  public partial class ReadLineActivity : Activity
  {
    public ReadLineActivity()
    {
      InitializeComponent();
    }

    protected override ActivityExecutionStatus Execute(
      ActivityExecutionContext executionContext)
    {
      string queueName = "MijnEigenQueueNaam";
      WorkflowQueuingService wqs =
        executionContext.GetService<WorkflowQueuingService>();

      WorkflowQueue queue = wqs.CreateWorkflowQueue(queueName, true);
      queue.QueueItemAvailable +=
        new EventHandler<QueueEventArgs>(queue_QueueItemAvailable);

      ReadLineServiceV2 rls =
        executionContext.GetService<ReadLineServiceV2>();
      rls.ReadLine("Wat is de volgende regel?", queueName);

      return ActivityExecutionStatus.Executing;
    }

    void queue_QueueItemAvailable(object sender, QueueEventArgs e)
    {
      ActivityExecutionContext executionContext =
        sender as ActivityExecutionContext;
      WorkflowQueuingService wqs =
        executionContext.GetService<WorkflowQueuingService>();
      WorkflowQueue queue = wqs.GetWorkflowQueue(e.QueueName);

      object data = queue.Dequeue();
      wqs.DeleteWorkflowQueue(e.QueueName);

      executionContext.CloseActivity();
    }
  }
}
```

Codevoorbeeld 3. De ReadLineActivity

voorstellen dat men zich afvraagt waarom er nog steeds een aparte service nodig is. Zouden we niet eenvoudigweg het lezen van de regel tekst in een andere thread in de activiteit zelf kunnen doen? Het zou zo veel eenvoudiger zijn als we niet een losse service aan de runtime hoeven toe te voegen zoals nu het geval is. Helaas is dit om een aantal redenen niet mogelijk. Het eerste probleem is dat we de workflow-runtime gebruiken in de service om een referentie te krijgen naar het WorkflowInstance-object. Dit WorkflowInstance-object is echter niet een object van de workflow-klasse zoals wij die gedefinieerd hebben, maar een wrapper-object van Workflow Foundation zelf waar we vanuit een workflow zelf niet bij kunnen. Een ander probleem is dat we ook niet bij de workflow-runtime zelf kunnen. Uiteraard zijn er wel manieren te bedenken om die referentie mogelijk te maken, zoals een static property ergens definiëren of een parameter, maar binnen de opzet van Workflow Foundation wordt er met opzet veel gedaan om te voorkomen dat een workflow of activiteit de runtime aanpast. Uiteindelijk hebben we alleen de queue nodig om de data in te stoppen en aangezien we die queue via de ActivityExecutionContext aanmaken, kunnen we die misschien doorgeven. Ook dat lukt niet, de ActivityExecutionContext is een object waar Workflow Foundation nogal bezitterig over doet. Het object komt als parameter bij de Execute binnen, maar wordt daarna meteen 'gedisposed' en nooit meer gebruikt. Het doorgeven en later gebruiken geeft direct een ObjectDisposedException tot gevolg. Dit context-object mag dan ook nooit bewaard worden voor later gebruik. Dan is er nog de optie om de queue zelf door te geven en te gebruiken vanuit een andere thread. Helaas gaat dat ook niet werken, want dit resulteert in een InvalidOperationException met als melding 'This method can only be called on a runtime thread.'. We kunnen de werkwijze met een extra service object dus niet voorkomen, maar misschien kunnen we voorkomen dat de gebruiker onze activiteit zelf aan de workflow-runtime moet toevoegen; zie codevoorbeeld 5.

```
workflowRuntime.AddService(new ReadLineServiceV2());
```

Codevoorbeeld 5. De workflow-service aan de runtime toevoegen

Helaas gaat dit ook niet lukken. Waar de workflow-runtime zowel een AddService() als een GetService()-functie heeft, blijkt de ActivityExecutionContext alleen een GetService() te ondersteunen. Deze scheiding van runtime-services en wat er binnen een workflow- activiteit kan gebeuren, blijft altijd een moeilijk punt zodra ontwikkelaars verder gaan met Workflow Foundation.

```
using System;
using System.Threading;
using System.Workflow.Runtime;
using System.Workflow.Runtime.Hosting;

namespace WorkflowConsoleApplication1
{
  public class ReadLineServiceV2 : WorkflowRuntimeService
  {
    public void ReadLine(string prompt, IComparable queueName)
    {
      Guid instanceId = WorkflowEnvironment.WorkflowInstanceId;
      ThreadPool.QueueUserWorkItem(delegate(object state)
      {
        Console.WriteLine(prompt);
        string input = Console.ReadLine();

        WorkflowInstance instance = Runtime.GetWorkflow(instanceId);
        instance.EnqueueItem(queueName, input, null, null);
      });
    }
  }
}
```

Codevoorbeeld 4. De bijbehorende nieuwe ReadLineService

Belangrijk: Workflow Foundation is zo opgezet dat er binnen een workflow nooit iets aan de workflow-runtime gewijzigd kan worden!

De levenscyclus van een activiteit.

Voor we verder gaan is het van belang de levenscyclus van een workflow-activiteit te begrijpen. Ook hier komen we weer enige verrassingen tegen waarvan het begrip van belang is als we eigen activiteiten gaan ontwikkelen. Om te beginnen wordt van een activiteit altijd de Initialize()-functie aangeroepen, waarna deze in de Initialized-status is. Deze eerste Initialize()-functie krijgt ook een parameter mee van het type IServiceProvider die in werkelijkheid ook weer een ActivityExecutionContext- object is. De statuswisselingen van een activiteit kunnen alleen in de voorgeschreven volgorde plaatsvinden. Vanuit de Initialized-status kunnen we alleen naar de Executing-status gaan, niet naar één van de andere mogelijke statussen. Nu kunnen we tijdens de Initialize()-functie via de parameter ActivityExecutionContext proberen een CloseActivity()-functie aan te roepen. Het gevolg is echter dat we een InvalidOperationException-fout krijgen met als tekst 'Activity status not suitable for closing'. Het mag dus duidelijk zijn dat de workflow-runtime actief controleert of we ons aan de regels houden en zo niet gelijk preventief optreedt.

Als we afbeelding 2 bekijken, zien we dat er in elke status alleen bepaalde overgangen mogelijk zijn. Als we proberen daarvan af te wijken, zal de workflow-runtime een fout genereren. Alle statussen die hier beschreven staan, komen ook terug in virtual/overridable functies. Zo hebben we de volgende functies om mee te werken:

- **Initialize()** Bij het starten van de workflow.
- **Execute()** Zodra een activiteit echt aan bod komt.
- **OnClosed()** Zodra een activiteit klaar is. Dit kan op verschillende manieren en eventueel meer keren gebeuren.
- **Cancel()** Indien een activiteit wordt geannuleerd. Dit kan ook op verschillende manieren gebeuren.
- **HandleFault()** Indien er een exceptie optreedt die niet afgevangen wordt.
- **Uninitialize()** Als de workflow klaar is. Dit is de tegenhanger van de Initialize(). Er is geen bijbehorende status, omdat de workflow hierna is beëindigd.
- **Compensate()** De Compensate() kan alleen aangeroepen worden op activiteiten die de ICompensatableActivity-interface implementeren.

Naast deze lijst van virtuele functies zijn er nog twee extra virtuele functies op een activiteit te vinden die van belang zijn voor de levenscyclus van een activiteit. Dit zijn de functies OnActivityExecutionContextLoad() en de OnActivityExecutionContextUnload() die elke keer worden aangeroepen als een activiteit geladen en ontladen wordt. Dit lijkt op het eerste gezicht hetzelfde als de functies Initialize() en Uninitialize(), maar deze laatste gaan maar één keer af, terwijl de OnActivityExecutionContextLoad() en OnActivityExecutionContextUnload() ook elke keer aangeroepen worden als de workflow in een persistence store wordt bewaard. Een voor de hand liggende

Afbeelding 2. De mogelijke levenscyclus van een workflow-activiteit

functie die binnen Workflow Foundation niet erg vaak bruikbaar blijkt te zijn, is de constructor. Binnen Workflow Foundation worden activiteiten namelijk nogal eens geserializeerd en gedeserializeerd, waarbij de standaard constructor nooit aangeroepen wordt.

Spawned Contexts

Zoals uit de levenscyclus duidelijk mag zijn, kan een activiteit nooit voor een tweede keer de Executing-status krijgen. Maar binnen Workflow Foundation bestaat er zoiets als een WhileActivity en zoals de naam doet vermoeden, gaat die alle activiteiten die hierbinnen staan meer keren uitvoeren. Hoe kan dit als de activiteiten hierbinnen maar één keer uitgevoerd kunnen worden? Om dit dilemma op te lossen, is het concept van de spawned context uitgevonden. Deze spawned context is iets dat nogal eens verwarring geeft, maar is eigenlijk heel eenvoudig.

Belangrijk: Een activiteit kan maar één keer in de Executing-state komen, ook als deze binnen een repeterende activiteit staat.

Als een workflow-activiteit meer keren uitgevoerd kan worden, wordt niet de activiteit zelf uitgevoerd, maar wordt er eerst een kloon van gemaakt. Deze kloon wordt dan uitgevoerd in plaats van de originele activiteit zelf. De originele activiteit dient hierbij alleen als template-activiteit. Eén van de gevolgen hiervan is dat eventuele activiteit-properties, die binnen een Execute() gezet worden, op de kloon gezet worden en niet op de template-activiteit zelf, zodat ze tijdens een volgende iteratie van de lus weg lijken te zijn. Er wordt immers een nieuwe kloon van de oorspronkelijke template-activiteit uitgevoerd. Overigens wordt niet alleen de activiteit zelf gekloond, maar ook alle onderliggende activiteiten en de bijbehorende ActivityExecutionContext.

Overigens komt een spawned context niet alleen voor bij een WhileActivity, maar bij alle activiteiten waarvan onderdelen meer keren uitgevoerd kunnen worden zoals een ReplicatorActivity, ConditionedActivityGroup en StateActivity.

Properties toevoegen

Vroeger of later moeten we ook nieuwe properties aan onze eigen activiteiten toe gaan voegen. Ook hier zijn dingen net even anders dan normaal, al is de verandering hier niet specifiek voor Workflow Foundation. Binnen een workflow-activiteit kan men het best werken met een zogenaamde DependencyProperty in plaats van een normale property zoals we deze op een standaard .NET-klasse gebruiken. Niet dat het gebruik van een normale property niet mogelijk zou zijn, maar een DependencyProperty heeft voordelen die ze de moeite waard maken.

Eén van de belangrijkste voordelen van dependency property is activity-binding. Met activity-binding wordt een property niet direct van een waarde voorzien, maar van een expressie met welke tijdens het uitvoeren van de activiteit de waarde uitgevraagd wordt. Deze expressie wordt met behulp van een ActivityBind-object in het property-sheet opgegeven met de volgende syntax: 'Activity=ActiviteitNaam, Path=PropertyNaam'. Het gevolg hiervan is dat bij het lezen van de waarde van onze property de runtime op zoek gaat naar de activiteit met de opgegeven naam en daarvan de property met de opgegeven naam leest en die waarde teruggeeft. Deze property kan zelf ook weer een ActivityBind bevatten en zo kan een hele zoekketen ontstaan om de werkelijke waarde te bepa-

```
Activity childActivity = EnabledActivities[0];
ActivityExecutionContext childContext =
   executionContext.ExecutionContextManager.
   CreateExecutionContext(childActivity);
childContext.ExecuteActivity(childContext.Activity);
```

Codevoorbeeld 6. Het aanmaken van een spawned context

len. Groot voordeel van deze werkwijze is dat het niet nodig is om code te schrijven die de inhoud van een property naar een andere kopieert. Overigens kunnen we met een ActivityBind wel een normale property, of zelfs een field, uitlezen.

Belangrijk: Gebruik een DependencyProperty in plaats van een standaard property.

Een tweede belangrijk voordeel van een DependencyProperty is dat deze als metadata gekenmerkt kan worden. Door dit te doen is een property niet meer te wijzigen tijdens het uitvoeren van een activiteit. De Enabled-property is hier een voorbeeld van. Die is het best te vergelijken met een stuk code in commentaar veranderen en dat kan alleen tijdens het ontwikkelen en niet tijdens het uitvoeren. Een ander voordeel is hoe de data geserializeerd worden. Bij het serializeren van een workflow naar schijf wordt een binair formaat gebruikt. Bij dit binaire formaat kunnen gemakkelijk problemen ontstaan als er verschillende versies van een activiteit in omloop zijn en men hier niet goed over nagedacht heeft. Iets wat gemakkelijk kan gebeuren door het langlopende karakter van een workflow. Door een DependencyProperty te gebruiken, heeft de workflow-runtime alle controle over hoe data bewaard worden en zo kunnen eventuele versieproblemen voorkomen worden. Overigens kan er bij het definiëren van een DependencyProperty via de DependencyPropertyOptions-enumeratie ook opgegeven worden dat een property helemaal niet geserializeerd moet worden. Een vierde voordeel van het gebruik van een DependencyProperty is de mogelijkheid om attached properties te gebruiken. Met een attached property wordt een property aan een andere klasse dan de eigen toegevoegd, de andere klasse wordt dus eigenlijk tijdens het uitvoeren uitgebreid. Dit is niet iets dat vaak nodig is, maar wat in sommige gevallen heel gemakkelijk kan zijn. Een voorbeeld hiervan is de ConditionedActivityGroup die een WhenCondition-property toevoegt aan de geneste activiteiten. Als laatste is er nog een duidelijke performance-reden. Door de manier waarop een workflow wordt uitgevoerd, gebeurt het vaak dat activiteiten geserializeerd en weer gedeserializeerd worden. Omdat dependecy-properties een 'lazy loading'-eigenschap hebben en niets doen tot de property echt gebruikt wordt, kan dit een behoorlijke snelheidswinst opleveren.

Event-gestuurde activiteiten

De activiteit die we hierboven ontwikkeld hebben om een regel van de console te lezen, werkt prima maar heeft een beperking, omdat deze alleen in de normale executie-flow gebruikt kan worden. De workflow-runtime zal dus altijd wachten tot er een regel tekst is gelezen. Dat kan best het gewenste gedrag zijn, maar in andere gevallen wil je misschien maximaal een bepaalde tijd wachten en indien er niets ingevoerd is wat anders gaan doen. Dat kan binnen een workflow gemakkelijk door een ListenActivity toe te voegen met de ReadLineActivity als eerste tak en een tweede tak met een DelayActivity die zorgt dat de workflow altijd doorgaat als er na een bepaalde tijd geen invoer gedetecteerd is.
De ReadLineActivity zoals we die eerder gemaakt hebben, is echter niet geschikt om als eerste activiteit binnen een EventDrivenActivity geplaatst te worden, omdat die de verschillende takken van een ListenActivity vormt. Op die plaats moet namelijk altijd een activiteit komen die de IEventActivity-interface implementeert. Nu kunnen we een tweede ReadLineActivity maken die de IEventActivity-interface implementeert, maar dat maakt het voor een gebruiker van onze activiteiten niet bepaald gemakkelijker. De beste optie is om de twee verschillende manieren te combineren. Om dit te doen, moeten we zowel de IEventActivity- als de IActivityEventListener<T>-interfaces gaan implementeren en bijhouden hoe de activiteit het eerst gestart wordt. Als de activiteit gewoon in de normale executievolgorde van een rij activiteiten staat, zal de functie Execute() als eerste aangeroepen worden. Als echter de functie Subscribe(), die onderdeel is van de IEventActivity-interface, eerder wordt aangeroepen, staat de activiteit als eerste activiteit in een EventDrivenActivity. Afhankelijk van de twee opties moeten we of de EventDrivenActivity of de ReadLineActivity als doel registreren voor een notificatie als er gegevens in de queue beschikbaar zijn. Aangezien de EventDrivenActivity de IActivityEventListener<T> implementeert, kunnen we dit het beste zelf doen, hoewel dat niet verplicht is.

```
public string Question
{
   get { return (string)GetValue(QuestionProperty); }
   set { SetValue(QuestionProperty, value); }
}

public static readonly DependencyProperty QuestionProperty =
   DependencyProperty.Register("Question",
   typeof(string),
   typeof(ReadLineActivity));
```

Codevoorbeeld 7. Een dependency-property binnen een activiteit

Zoals we in codevoorbeeld 8 kunnen zien, is de code die hiervoor nodig wel een stuk langer dan de originele ReadLineActivity. Gelukkig is het grootste deel van de code generiek en slechts een klein deel hiervan is implementatiespecifiek, zodat een groot deel gemakkelijk in een abstracte basisklasse verwerkt kan worden. In dit geval zitten de enige implementatiespecifieke codedelen in de gebruikte queue-naam, het laatste stuk van de InternalSubscribe()-functie en de hele ReceiveLine()-functie. Overigens zou je deze abstracte activiteit als standaard activiteit in de workflow-bibliotheek verwachten, maar helaas is dit niet gebeurd. Naast de eerder genoemde functies heb ik hier ook de Cancel()-functie geïmplementeerd. Deze functie wordt aangeroepen als de activiteit geannuleerd wordt. Dit laatste kan op meer manieren gebeuren, onder meer door een andere parallelle tak of door de runtime.

Activiteiten en langlopende workflows

Eén van de grote voordelen van Workflow Foundation is dat een workflow lang kan duren en tussentijds op schijf geserializeerd kan worden. Dat laatste is bijna een verplichting, want als je een workflow maakt die dagen, maanden of misschien zelfs jaren kan duren, is het niet realistisch om te verwachten dat het proces waarbinnen alles gebeurt al die tijd actief blijft. Dit heeft echter wel wat gevolgen voor de zelf te ontwikkelen activiteiten, zeker als deze samen moeten werken met externe services en op een gebeurtenis wachten. De activiteit zelf is meestal geen groot probleem aangezien de workflow in zijn geheel geserializeerd wordt, inclusief eventueel aangemaakte workflow-queues.

Belangrijk: Activiteiten moeten kunnen omgaan met het serializeren naar schijf en herstarten van het proces, eventueel op een andere machine.

Een optie is om de externe service zelf te laten onthouden dat er op een gebeurtenis gewacht wordt en deze service deze gebeurtenis na een eventuele herstart zelf te laten herstellen. Dit zou kunnen gebeuren door de functies OnStarted() en OnStopped() van de WorkflowRuntimeService te gebruiken. Binnen de service wordt dan via de workflow-runtime de functie GetWorkflow() gebruikt om een referentie te krijgen naar het betreffende WorkflowInstance-object. Mocht een workflow op het moment van de gebeurtenis nog op schijf bewaard zijn, dan zal de functie GetWorkflow() deze eerst laden. Het nadeel van deze optie is wel dat niet alleen de workflow-runtime maar ook de workflow- service een persistence-service moet hebben zodat de interne status voor langere tijd op schijf bewaard kan blijven. En niet alleen dat. Dit moet ook nog op dezelfde punten gebeuren als de workflow-runtime dat doet, omdat beide uiteraard onderdeel zijn van een groter geheel en dus als zodanig bewaard moeten blijven. Dit laatste kan het best bereikt worden door gebruik te maken van een IPendingWork-

```
using System;
using System.ComponentModel;
using System.Workflow.Activities;
using System.Workflow.ComponentModel;
using System.Workflow.Runtime;

namespace WorkflowConsoleApplication1
{
  public partial class ReadLineActivity :
    Activity,
    IEventActivity,
    IActivityEventListener<QueueEventArgs>
  {
    public ReadLineActivity()
    {
      InitializeComponent();
    }

    public static DependencyProperty IsInEventActivityModeProperty =
      DependencyProperty.Register(
      "IsInEventActivityMode",
      typeof(bool),
      typeof(ReadLineActivity));

    [Description("Omschrijving")]
    [Category("Categorie")]
    [Browsable(true)]
    [DesignerSerializationVisibility(DesignerSerializationVisibility.
    Visible)]

    public bool IsInEventActivityMode
    {
      get
      {
        return ((bool)(base.GetValue(
          ReadLineActivity.IsInEventActivityModeProperty)));
      }
      set
      {
        base.SetValue(ReadLineActivity.IsInEventActivityModeProperty,
          value);
      }
    }

    protected override ActivityExecutionStatus Execute(
      ActivityExecutionContext executionContext)
    {
      ActivityExecutionStatus status = ExecutionStatus;

      if (IsInEventActivityMode)
      {
        ReceiveLine(executionContext);
        status = ActivityExecutionStatus.Closed;
      }
      else
      {
        InternalSubscribe(executionContext, this, false);
        status = ActivityExecutionStatus.Executing;
      }

      return status;
    }

    protected override ActivityExecutionStatus Cancel(
      ActivityExecutionContext executionContext)
    {
      InternalUnsubscribe(executionContext, this);

      WorkflowQueuingService wqs =
        executionContext.GetService<WorkflowQueuingService>();
      if (wqs.Exists(QueueName))
        wqs.DeleteWorkflowQueue(QueueName);

      return ActivityExecutionStatus.Closed;
    }

    #region IEventActivity Members

    public IComparable QueueName
    {
      get { return "MijnEigenQueueNaam"; }
    }

    public void Subscribe(ActivityExecutionContext parentContext,
      IActivityEventListener<QueueEventArgs> parentEventHandler)
    {
      InternalSubscribe(parentContext, parentEventHandler, true);
    }

    public void Unsubscribe(ActivityExecutionContext parentContext,
      IActivityEventListener<QueueEventArgs> parentEventHandler)
    {
      InternalUnsubscribe(parentContext, parentEventHandler);
    }

    #endregion

    #region IActivityEventListener<QueueEventArgs> Members

    public void OnEvent(object sender, QueueEventArgs e)
    {
      ActivityExecutionContext executionContext =
        sender as ActivityExecutionContext;
      ReceiveLine(executionContext);
    }

    #endregion

    private void InternalSubscribe(
      ActivityExecutionContext executionContext,
      IActivityEventListener<QueueEventArgs> eventHandler,
      bool eventActivityMode)
    {
      WorkflowQueuingService wqs =
        executionContext.GetService<WorkflowQueuingService>();
      WorkflowQueue queue = wqs.CreateWorkflowQueue(QueueName, true);
      queue.RegisterForQueueItemAvailable(eventHandler, QualifiedName);
      IsInEventActivityMode = eventActivityMode;

      ReadLineServiceV2 rls =
        executionContext.GetService<ReadLineServiceV2>();
      rls.ReadLine("Wat is de volgende regel?", QueueName);
    }

    private void InternalUnsubscribe(
      ActivityExecutionContext executionContext,
      IActivityEventListener<QueueEventArgs> eventHandler)
    {
      WorkflowQueuingService wqs =
        executionContext.GetService<WorkflowQueuingService>();
      WorkflowQueue queue = wqs.GetWorkflowQueue(QueueName);
      queue.UnregisterForQueueItemAvailable(eventHandler);
    }

    private void ReceiveLine(ActivityExecutionContext executionContext)
    {
      WorkflowQueuingService wqs =
        executionContext.GetService<WorkflowQueuingService>();
      WorkflowQueue queue = wqs.GetWorkflowQueue(QueueName);

      object data = queue.Dequeue();

      // Verwerk de intvangen data.
      Console.WriteLine("Recieved: {0}", data);

      InternalUnsubscribe(executionContext, this);
      wqs.DeleteWorkflowQueue(QueueName);

      executionContext.CloseActivity();
    }
  }
}
```

Codevoorbeeld 8. De ReadLineActivity die zowel als event-gestuurd als normaal gebruikt kan worden.

object in combinatie met een WorkBatch. Als de runtime-service de workflow weer laadt, wordt in de activiteit de functie OnActivityExecutionContextLoad() aangeroepen. In deze functie kan dan een eventuele registratie bij de service opnieuw gedaan worden.
Een andere manier om dit probleem op te lossen, is misbruik te maken van het systeem dat voor een DelayActivity gebruikt wordt. Door dit te doen zal de SqlWorkflowPersistenceService een next-Timer-waarde bewaren en de workflow opnieuw laden als deze is verlopen. Hierbij is het geen probleem dat de workflow niet echt op een verlopen DelayActivity staat te wachten. Om dit systeem te gebruiken, moet binnen de activiteit een TimerEventSubscription aan de TimerEventSubscriptionCollection toegevoegd worden. Met dit systeem wordt de functie OnActivityExecutionContextLoad() automatisch aangeroepen zodra de workflow-runtime opnieuw start. Dit lijkt wel een gemakkelijk systeem te zijn, maar aangezien het gebaseerd is op het misbruiken van de interne werking van de DelayActivity blijft het een minder ideale werkwijze.

Transacties en Persistence Points

En net als we denken dat we er zijn, blijken er nog meer potentiele problemen te zijn waar we rekening mee moeten houden. Dit gebeurt met name als we een combinatie van een externe service en een eigen activiteit gebruiken. Het probleem is eigenlijk heel eenvoudig en draait helemaal om hoe we alle verschillende databronnen synchroon houden. Stel je voor, we hebben een activiteit die wacht op een bericht dat via MSMQ moet binnenkomen. Er is een externe service die de MSMQ-queue controleert. Als deze het bericht ziet, dan wordt het gelezen en in de workflow-queue gezet. Nu leest de activiteit het bericht uit de workflow-queue en gaat de workflow verder. Het geheel draait in een workflow-runtime voorzien van een SqlWorkflowPersistenceService, zodat langlopende workflows mogelijk zijn. Alleen is er nu een stroomstoring en herstart de hele machine, nadat de activiteit het bericht gelezen heeft, maar voordat de SqlWorkflowPersistenceService de nieuwe staat van de workflow weer opslaat. Als de machine nu herstart is en de workflow-runtime weer actief wordt, zal de SqlWorkflowPersistenceService er voor zorgen dat onze activiteit weer op het MSMQ-bericht gaat wachten. Alleen hebben we dat bericht al gelezen en komt het niet voor een tweede keer. Dit kunnen we niet oplossen door een TransactionScope voor het lezen van het bericht uit de MSMQ-queue in te zetten en dit bericht in de workflow-queue te plaatsen. Dit is immers goed gegaan en de activiteit heeft het bericht zelfs al uit de workflow-queue gelezen. We kunnen ook geen TransactionScope maken die alle code bevat tot de volgende keer dat de SqlWorkflowPersistenceService de workflow bewaart. Dit is namelijk allemaal onder controle van workflow-runtime en niet van onze code.
De oplossing voor dit probleem kan worden gevonden in de derde en vierde parameter van de functie WorkflowInstance.EnqueueItem() waar we eerst null-waardes aan meegaven. De derde parameter is van het type IPendingWork, de vierde een WorkItem-object met data die wat later aan het IPendingWork-object worden gegeven. Nu kunnen we in de service het bericht uit de MSMQ-queue niet gewoon lezen, maar met een Peek() ophalen en de LookupId als het data deel meegeven. Als we nu in de ReadLineService de IPendingWork implementeren, krijgen we daar later een event van met in het bericht de eerder verwerkte LookupId en kunnen we deze definitief verwijderen uit de MSMQ-queue met behulp van de functie ReceiveByLookupId().

Belangrijk: Gebruik de WorkBatch-service met een WorkItem en de IPendingWork-interface om binnen een transactie te kunnen werken.

Dan rest de vraag: wanneer wordt het IPendingWork-object aangeroepen met de opgegeven data? Deze aanroep gebeurt op de zogenaamde persistence points en die komen overeen met de punten waarop de status van een workflow definitief te noemen is.

De gemakkelijkste manier om een definitieve status te herkennen, is te bedenken wat er bewaard zou blijven als het proces met de workflow-runtime herstart zou worden. Er zijn vijf zogenaamde persistence points:

1. Voor het workflow Completed-event. Dus als de workflow tot een goed einde gekomen is.
2. Voor het workflow Terminated-event. Als een workflow niet tot een goed eind gekomen is, maar wel afgelopen.
3. Als er een workflow Unload()- of TryUnload()-functie wordt aangeroepen.
4. Als een workflow idle wordt en er een SqlWorkflowPersistenceService is met de optie unloadOnIdle enabled.
5. Nadat er een activiteit klaar is die voorzien is van het attribuut PersistOnClose.

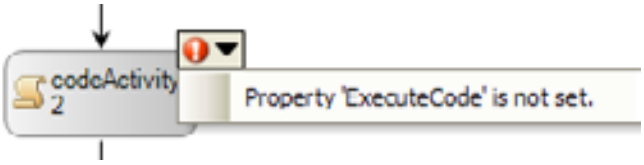
De eerste vier opties zijn eigenlijk vrij duidelijk. In al deze gevallen is de workflow klaar of wordt de status door de SqlWorkflowPersistenceService op schijf bewaard. De laatste optie, die van het attribuut PersistOnClose is interessanter om naar te kijken, want die heeft een aantal andere bijwerkingen. Misschien is het wel eens opgevallen dat je bij sommige activiteiten verplicht bent om een WorkflowPersistenceService te gebruiken. Dit gebeurt bij de TransactionScopeActivity en bij de CompensatableTransactionScopeActivity. Beide zorgen er voor dat alle uitgevoerde activiteiten binnen een transactie gebeuren. Om er nu voor te zorgen dat de workflow status en de transactie altijd synchroon blijft, ook als de machine herstart wordt, zijn deze twee activiteiten voorzien van het attribuut PersistOnClose. Dit attribuut verplicht de runtime om de status van de workflow als laatste onderdeel van de transactie te bewaren. En niet alleen de status van de workflow gaat daarbij naar schijf, ook alle pending WorkItems worden naar hun IPendingWork-object gestuurd.

Belangrijk: Alle activiteiten die een extern bijeffect hebben, zoals het wegschrijven van een bestand, moeten voorzien zijn van het attribuut PersistOnClose om te voorkomen dat dit een tweede keer kan gebeuren.

Overigens is het systeem met de pending WorkItems ook buiten de WorkflowInstance.EnqueueItem() te gebruiken. Via de property WorkflowEnvironment.WorkBatch en de Add()-functie is overal tijdens het uitvoeren van een activiteit een WorkItem met bijbehorend IPendingWork-object toe te voegen aan de workflow. Deze worden dan tijdens het volgende persistence point afgehandeld.

Valideren van activiteiten tijdens het bouwen van workflows
Bij het maken van de eerste workflows zul je vast wel eens een CodeActivity gebruikt hebben. Als je die dan op een workflow sleept, verschijnt er aan de rechterbovenhoek een rood icoon met een uitroepteken dat aangeeft dat de property ExecuteCode nog niet gezet is. Een voorbeeld hiervan is te zien in afbeelding 3. Dit gebeurt door een zogenaamde ActivityValidator-klasse die aan de CodeActivity toegevoegd is met behulp van een ActivityValidator-attribuut.

Het valideren van een eigen activiteit is niet moeilijk om te doen, zoals in codevoorbeeld 9 te zien is. Als eerste moeten we een nieuwe klasse maken die is afgeleid van de ActivityValidator-klasse. Daarin maken we een nieuwe Validate()-functie die de eventuele fouten in een collectie teruggeeft. En om de activiteit en de validatie aan elkaar te koppelen, wordt op de activiteit een ActivityValidator-attribuut toegevoegd met het type van de validatieklasse als parameter.



Afbeelding 3. Een CodeActivity die volgens de activiteit validatie niet geldig is

```
[ActivityValidator(typeof(ReadLineActivityValidator))]
public partial class ReadLineActivity :
  Activity
{

  public string Question
  {
    get { return (string)GetValue(QuestionProperty); }
    set { SetValue(QuestionProperty, value); }
  }

  public static readonly DependencyProperty QuestionProperty =
    DependencyProperty.Register("Question", typeof(string),
    typeof(ReadLineActivity));
}

public class ReadLineActivityValidator : ActivityValidator
{
  public override ValidationErrorCollection Validate(
    ValidationManager manager, object obj)
  {
    ValidationErrorCollection errors =
      new ValidationErrorCollection();
    ReadLineActivity activity =
      obj as ReadLineActivity;

    if (activity != null &&
      activity.Parent != null &&
      string.IsNullOrEmpty(activity.Question))
    {
      ValidationError error =
        new ValidationError("De vraag is niet ingevuld!", 0);
      error.PropertyName = "Question";
      errors.Add(error);
    }
    errors.AddRange(base.Validate(manager, obj));
    return errors;
  }
}
```

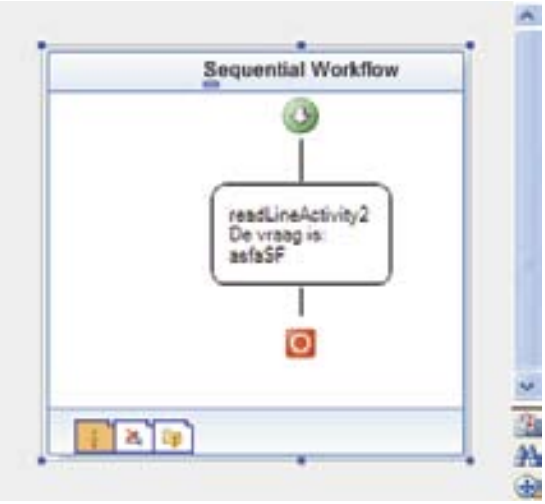
Codevoorbeeld 9. Een activiteit met validatie

```
[Designer(typeof(ReadLineActivityDesigner))]
[ActivityValidator(typeof(ReadLineActivityValidator))]
public partial class ReadLineActivity :
  Activity
public class ReadLineActivityDesigner : ActivityDesigner
{

  protected override Size OnLayoutSize(ActivityDesignerLayoutEventArgs e)
  {
    Size size = base.OnLayoutSize(e);

    size.Height += 25;
    size.Width += 20;
    return size;
  }

  protected override void OnPaint(ActivityDesignerPaintEventArgs e)
  {
    ReadLineActivity activity = this.Activity as ReadLineActivity;

    MethodInfo mi = typeof(ActivityDesignerPaint)
      .GetMethod("DrawDesignerBackground",
      BindingFlags.Static | BindingFlags.NonPublic);
    mi.Invoke(null, new object[] { e.Graphics, this });

    Rectangle rect = e.ClipRectangle;
    rect.Offset(10, 0);

    string text = string.Format("{0}\nDe vraag is:\n{1}",
      activity.QualifiedName, activity.Question);
    ActivityDesignerPaint.DrawText(
      e.Graphics, e.DesignerTheme.Font,
      text,
      rect,
      StringAlignment.Near,
      e.AmbientTheme.TextQuality,
      e.DesignerTheme.ForegroundBrush);
  }
}
```

Codevoorbeeld 10. Een activiteit met een eigen vormgeving in de workflow-designer

De vormgeving in de Workflow Designer

Ook de vorm van de activiteit in de designer is tot op detail aan te passen. Ik geef hier slechts een heel beperkt voorbeeld aangezien alle mogelijkheden van een ActivityDesigner een artikel op zich waard zijn. Dit beperkte voorbeeld is te vinden in codevoorbeeld 10 waar we de vormgeving zelf regelen.



Afbeelding 4. Een activiteit met een eigen designer

Eén noemenswaardig stukje code hierin is het gebruik van de ActivityDesignerPaint-klasse. Deze klasse wordt intern binnen workflow veel gebruikt om delen van activiteiten te tekenen. Helaas zijn nogal wat van de functies, zoals de functie DrawDesignerBackground(), niet publiek beschikbaar. Aangezien ik de code niet wil dupliceren, gebruik ik reflection om deze interne functie toch aan te kunnen roepen.

Rekening houden met

Workflow Foundation is een krachtig mechanisme om applicaties te schrijven. Om er goed gebruik van te kunnen maken, is het al snel nodig om zelf nieuwe activiteitklasses te ontwikkelen. In principe is dit niet erg moeilijk, maar gezien de lengte van dit artikel, is er wel veel waar je als ontwikkelaar rekening mee moet houden. En dan heb ik sommige exotische en minder gebruikte opties als codegeneratie met behulp van ActivityCodeGenerator-klasses nog niet eens behandeld.

Maurice de Beijer is een zelfstandig softwareontwikkelaar, Most Valuable Professional en bètatester voor Microsoft. Hij specialiseert zich in .NET, objectoriëntatie en het oplossen van technisch moeilijke problemen. Hiernaast is Maurice ook de voorzitter van de Visual Basic-sectie van het Software Developer Netwerk, www.sdn.nl. Speciaal voor Workflow Foundation heeft hij de site www.WindowsWorkflowFoundation.eu en het bijbehorende wiki opgezet. Maurice is te bereiken via Maurice@TheProblemSolver.nl of www.TheProblemSolver.nl.