**Erik van de Ven**
Erik is werkzaam als senior development consultant bij Microsoft Services. Hij houdt zich voornamelijk bezig met de technische architectuur, ontwerp en bouw van enterprise applicaties in .NET.

# Flexibele en onderhoudbare user-interfaces in .NET

## GEBRUIK VAN HET NIEUWE USER INTERFACE PROCESS APPLICATION BLOCK

**Het ontwikkelen van een goede user-interface wordt nog vaak onderschat. Het maken van een paar schermpjes is natuurlijk niet zo ingewikkeld, maar wanneer een productierijpe user-interface voor een omvangrijke enterprise-applicatie dient te worden ontwikkeld, tekenen zich toch vaak donkere wolken af aan die immer strakblauwe .NET-lucht. Er ontstaan dan ineens allerlei lastige vragen waar van tevoren goed over dient te worden nagedacht.**

Gaan we een webinterface of een WinForm-interface bouwen, of misschien wel allebei? Hoe kunnen we de logica dan hergebruiken? Hoeveel schermen moeten we bouwen? Kunnen we schermen gemakkelijk hergebruiken op andere plaatsen in de user-interface? Hoe gaan we informatie doorgeven tussen schermen? Dit zijn zomaar een paar spannende vragen die je tegenkomt wanneer je een user-interface moet ontwikkelen voor een omvangrijke applicatie. In dit artikel zal worden getoond hoe je het User Interface Process application block kunt gebruiken om al deze problemen op eenvoudige wijze op te lossen.

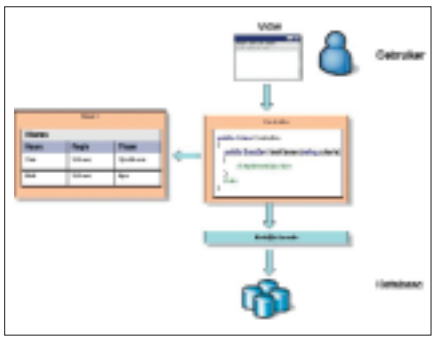
## Ontkoppeling tussen scherm en logica

De eerste vraag die we moeten beantwoorden is al een lastige. Moeten we nu een WinForm- of een WebForm-user-interface bouwen? Een afweging van de voor- en nadelen is te vinden in een van de patterns & practices [1]. Vaak zie je dat men kiest voor een combinatie van beide. De volledige functionaliteit van de applicatie is dan beschikbaar voor interne gebruikers via een WinForm-applicatie, terwijl een deel van de functionaliteit ook voor externe gebruikers via het web wordt aangeboden. Dit is een krachtige combinatie: voor een interne gebruiker die bijvoorbeeld de hele dag administratieve handelingen moet verrichten is een webomgeving veel te omslachtig. Een externe gebruiker daarentegen die

```
<navigationGraph
   iViewManager="SimpleViewManager"
   name="KlantBeheer"
   state="State"
   statePersist="SqlServerPersistState"
   startView="SelecteerKlant">

   <node view='SelecteerKlant'>
      <navigateTo navigateValue="selected" view='KlantOverzicht' />
   </node>

   <node view='KlantOverzicht'>
      <navigateTo navigateValue="orders" view='KlantOrders' />
      <navigateTo navigateValue="edit" view="KlantWijzigen" />
   </node>

   <node view='KlantOrders'>
      <navigateTo navigateValue="done" view='SelecteerKlant' />
   </node>

   <node view='KlantWijzigen'>
      <navigateTo navigateValue="saved" view='SelecteerKlant' />
      <navigateTo navigateValue="cancelled" view='KlantOverzicht' />
   </node>

</navigationGraph>
```

**Codevoorbeeld 1. Use-case KlantBeheer in XML**

slechts delen van de functionaliteit gebruikt, kan zonder enige vorm van installatie toch de applicatie gebruiken. Hoe kunnen we er in deze situatie voor zorgen dat we toch onderhoudbare code opleveren, die zowel ondersteuning biedt voor een Web- als WinForm-omgeving. Het model view-controller design pattern[2] biedt de oplossing.



Afbeelding 1. Toepassing databeheermodel in .NET

Het model-view-controller (MVC) design pattern maakt een duidelijk onderscheid tussen de view enerzijds en de controller anderzijds. De view is een component die verantwoordelijk is voor de interactie met de eindgebruiker; dit is dus de WinForm- of WebForm-pagina. De controller interpreteert de muis- en toetsenbordacties van de gebruiker en neemt aan de hand daarvan de juiste beslissing om het model te manipuleren. Het model beheert de data (state) en reageert op instructies van de controller enerzijds en op datawijzigingen van de view anderzijds. Hoe we dit model in .NET kunnen toepassen is weergegeven in afbeelding 1. De eindgebruiker werkt via het scherm (de view). Stel dat hij klikt op de knop Zoeken om klanten te zoeken die voldoen aan zijn gestelde criteria. Achter deze knop bevindt zich de code om de bijbehorende controller aan te roepen. Het ophalen van klanten implementeren we dus niet 'in het scherm zelf', maar in een afzonderlijke component – de controller genoemd. Deze controller haalt via de businessfaçade de gevraagde gegevens op en plaatst deze in het model. Het model bewaart dus de informatie die voor de user-interface van belang is. Het model vinden we terug in het User Interface Process application block als een object met de naam state. Dit object bevat een hashtable (System.Collections.Hashtable) en kan zodoende allerlei soorten informatie voor ons bewaren die we eenvoudig kunnen opvragen op basis

```
<views>
  <view
      name="SelecteerKlant"
      type="NorthwindTraders.Views.vwSelectCustomer, NorthwindTraders,
          Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
      controller="ctlSelectCustomer"
      stayOpen="true"/>
  <view
      name="KlantOverzicht"
      type="NorthwindTraders.Views.vwCustomerOverview, NorthwindTraders,
          Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
      controller="ctlCustomer" />
  <view
      name="KlantWijzigen"
      type="NorhtwindTraders.Views.vwEditCustomer, NorthwindTraders,
          Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
      controller="ctlCustomer" />
  <view
      name="KlantOrders"
      type="NorthwindTraders.Views.vwCustomerOrders, NorthwindTraders,
          Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
      controller="ctlCustomer"
      stayOpen="true"/>
  <view
      name="vwCreateShipment"
      type="NorthwindTraders.Views.vwCreateShipment, NorthwindTraders,
          Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
      controller="ctlCustomer" />
</views>
```

Codevoorbeeld 2. Sectie views

van een sleutel. Een strikte scheiding tussen view en controller stelt ons in staat om de logica (controllers) te kunnen hergebruiken voor zowel een Web- als WinForm-omgeving. De volgende vraag is natuurlijk hoe we schermen zelf kunnen hergebruiken.

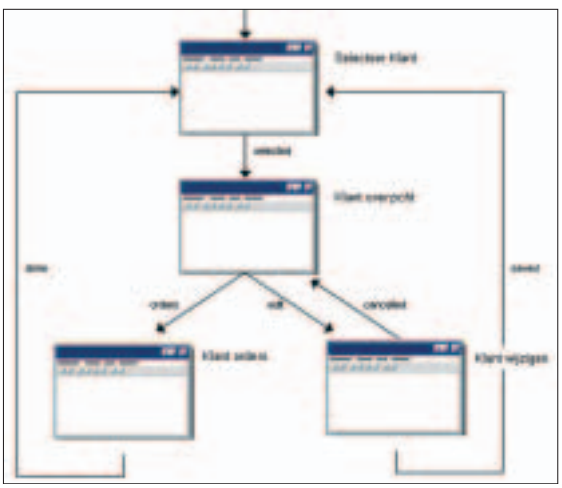
## Ontkoppeling tussen schermen

Om schermen op verschillende plaatsen in de user-interface te kunnen hergebruiken, is meer nodig dan alleen de scheiding tussen view en controller. Immers, het MVC-design pattern ontkoppelt binnen een scherm de interactie met de gebruiker van de achterliggende logica. Wat we nu nog nodig hebben is een ontkoppeling tussen schermen. Neem als voorbeeld het scherm Selecteer Klant. Stel dat dit een scherm is waarin een gebruiker enkele zoekcriteria (plaats, naam, enzovoort) kan noemen en vervolgens een lijst kan zien van alle klanten die hieraan voldoen. Het enige doel in het leven van dit scherm is een door de gebruiker geselecteerde klant op te leveren. Dit is typisch een scherm dat je op meer plaatsen kunt gebruiken. Om dit mogelijk te maken dienen schermen te worden beschouwd als losse componenten met een bepaalde input en output. Een scherm mag en moet niets weten van de omgeving waarin hij wordt gebruikt. Als we dit patroon stug volgen kunnen we onze user-interface in 'elkaar zetten' door schermen achter elkaar te plaatsen. Het achter elkaar plaatsen van schermen wordt ook wel een window flow genoemd. Een window flow is eigenlijk niets anders dan een gerichte graph waarin de nodes een scherm voorstellen en een pijl de overgang tussen twee schermen. Een voorbeeld van de use-case KlantBeheer is weergegeven in afbeelding 2. In dit voorbeeld – geïnspireerd door de Northwind-database – zien we duidelijk dat de gebruiker eerst een klant moet selecteren en vervolgens op het scherm KlantOverzicht terecht komt. Vanaf

```
/// <summary>
/// The select customers view
/// </summary>
public class vwSelectCustomer : WinFormView
{

  private controls...

  private ctlSelectCustomer Controller
  {
      get
      {
      // cast it to our type
      return((ctlSelectCustomer) base.Controller);
      }
  }

  private void cmdZoek_Click(object sender, System.EventArgs e)
  {
      dgCustomers.DataSource = ctlSelectCustomer.GetCustomers(txtId.Text,
               txtName.Text, txtCity.Text);
  }

      private void dgCustomers_DoubleClick(object sender,
          System.EventArgs e)
  {

      //double click -> customer has been selected

      // retrieve customer id from first columnn
      string customerId = dgCustomers[dgCustomers.CurrentCell.RowNumber,
               0].ToString();

      // invite controller to move on ...
      Controller.SelectCustomer(customerId);
  }
}
```

**Codevoorbeeld 3. Code om de juiste controller te laden**

dit scherm kan hij twee kanten op navigeren: of hij wil de orders van de gekozen klant zien, of hij wil de gekozen klant wijzigen. In beide gevallen zal hij weer terechtkomen in het startscherm Selecteer Klant en kan hij opnieuw beginnen. In de volgende paragraaf zullen we laten zien hoe we dit eenvoudig kunnen implementeren met behulp van het User Interface Process application block.

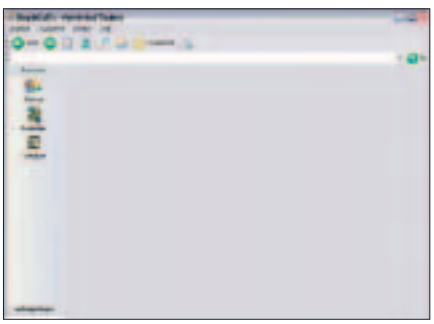


**Afbeelding 2. Use-case KlantBeheer**

## Het gebruik van het application block

In deze paragraaf zullen we laten zien hoe we daadwerkelijk het application block kunnen gebruiken om de hierboven genoemde use-case KlantBeheer te implementeren. In eerste instantie zullen we de flow van afbeelding 2 moeten formaliseren. Het application block definieert hiervoor een XML-formaat. In codevoorbeeld 1 zie je hoe we de dezelfde informatie van afbeelding 2 kunnen weergeven in XML. We voegen dit als een extra sectie toe aan ons app.config-bestand. Om te voorkomen dat we schermdefinities verscheidene keren moeten opnemen, worden logische namen gebruikt voor de schermen. De naam SelecteerKlant bijvoorbeeld verwijst naar de sectie views – zie codevoorbeeld 2 – waarin de exacte definitie van het scherm en zijn bijbehorende controller zijn te vinden.

Een use-case wordt in het application block ook wel een navigation graph of task genoemd. In het element navigationGraph zien we een aantal attributen. Het attribuut iViewManager bevat de naam van een klasse die IViewManager implementeert. Een viewmanager is verantwoordelijk voor het tonen van het huidige actieve scherm. In het application block vindt je een standaard implementatie die je kunt gebruiken voor het tonen van WinForm-schermen (WinFormViewManager) en een klasse die je kunt gebruiken voor het tonen van WebForm-schermen (WebFormViewManager). In dit voorbeeld hebben we een eigen viewmanager ontwikkeld met de naam SimpleViewManager. In afbeelding 3 zie je hoe deze viewmanager eruit ziet. De viewmanager biedt een raamwerk waarin schermen kunnen worden getoond. Een ander attribuut is startView dat het startscherm van de use-case vastlegt. De overige attributen state en statePersist zullen later worden besproken.

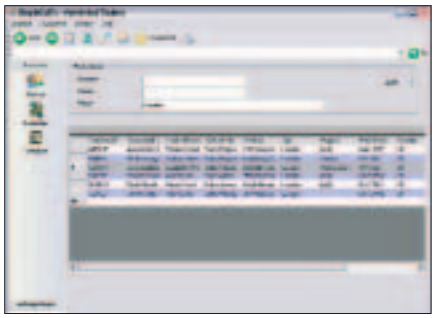


**Afbeelding 3. De viewmanager**

Nu we de navigatiemogelijkheden tussen de schermen hebben vastgelegd, is de volgende stap het ontwikkelen van de schermen zelf. Dat is vrij eenvoudig. We voegen een nieuwe WinForm toe en laten deze erven van het voorgedefinieerde type Microsoft.ApplicationBlocks.UIPro-

```
private void buttonItem1_Click(object sender, System.EventArgs e)
{
// start window flow 'KlantBeheer'
  UIPManager.StartTask("KlantBeheer");
}
```

cess.WinFormView. Ook de overige schermen ontwikkelen we op gelijke wijze. Nu we zowel de navigatie als de schermen zelf hebben gedefinieerd, kunnen we de navigatie starten om ons eerste scherm te laten tonen. Hiervoor biedt het application block het type UIPManager. Deze klasse bevat een static methode StartTask met een drietal overloads. Om de use-case (dat is: de task) BeheerKlanten te starten, plaatsen we de volgende code (zie boven) 'achter het klanten pictogram' van afbeelding 3.



**Afbeelding 4. Viewmanager toont resultaat**

Het User Interface Process application block zal nu de configuratie van afbeelding 3 en 4 lezen om het startscherm te kunnen vinden en vervolgens de viewmanager te vragen deze te tonen. Het resultaat hiervan is te zien in afbeelding 4. De laatste stap die ons nu nog rest is het ontwikkelen van een controller voor de logica achter dit scherm. Het scherm Selecteer Klant bevat slechts twee functionaliteiten: een gebruiker kan klanten zoeken op basis van opgegeven criteria, en hij kan een gebruiker selecteren. Deze beide acties zien we terug als public methoden op een controller. Op het scherm plaatsen we vervolgens code achter de knop Zoeken om de controller aan te roepen. Hetzelfde doen we voor de code om een dubbelklik op een rij af te vangen. De UIPManager zorgt dat de juiste controller wordt geladen en dat deze aanwezig is in de view. De code is weergegeven in codevoorbeeld 3.

Voor de ontwikkeling van een controller definiëren we een klasse die we laten erven van het voorgedefinieerde type Microsoft.ApplicationBlocks.UIProcess.ControllerBase; zie codevoorbeeld 4. De ControllerBase-klasse bevat een public state-object dat alle toestandsinformatie kan bewaren die we willen doorgeven tussen de schermen. In de methode SelectCustomer kun je zien dat de customerID van de geselecteerde klant wordt bewaard in de state-collectie. In een volgend scherm kunnen we deze weer uitvragen.

Daarnaast bevat het state-object een property NavigateValue en een public methode Navigate(). Deze twee worden gebruikt om vanuit de controller aan te geven dat je wilt navigeren naar het volgende scherm. De controller genereert een event waar de UIPManager naar luistert. Het is vervolgens de UIPManager die op basis van de configuratie uit codevoorbeeld 1 bepaalt wat het volgende scherm is dat geladen dient te worden. In dit geval levert de combinatie van het actieve scherm Selecteer Klant en de navigatiewaarde selected als resultaat het scherm KlantOverzicht. De UIPManager zal nu dit scherm met zijn bijbehorende controller laden. Op deze wijze krijgt de gebruiker de indruk dat hij met een aaneengesloten applicatie werkt, terwijl het in feite losse schermen zijn die dynamisch aan elkaar worden gekoppeld. Hierdoor zijn we in staat zeer flexibele user-interface te bouwen en kunnen we schermen heel gemakkelijk in een andere use-case hergebruiken.

## Herstarten van taken

Terwijl de eindgebruiker door de schermen wandelt, wordt eigenlijk continue de toestandsinformatie (state) van de applicatie gewijzigd. De gebruiker maakt bepaalde keuzes en heeft een bepaald scherm in zijn

```
/// <summary>
/// Controller to support Select customers view
/// </summary>
public class ctlSelectCustomer : ControllerBase
{

  private const decls

  public ctlselectcustomer(state controllerstate) :
      base (controllerstate)
  {
      // nothing to do
  }

  public void SelectCustomer(string customerId)
  {
      // store selected customer
      State["customerId"] = customerId;

      // navigate to next step
      State.NavigateValue = "selected";
      Navigate();
  }

  public static DataTable GetCustomers(string id, string name, string city)
  {

      implementation to retrieve customers

  }
}
```

**Codevoorbeeld 4. Definiëren klasse op basis van Microsoft.ApplicationBlocks.UIProcess.ControllerBase**

```
public class UserTask : ITask
{
   private Guid _taskId = Guid.Empty;
   private string _gebruikersNaam;

   public UserTask(string gebruikersNaam)
   {
       // bepaal of er reeds een lopende taak is voor
       // deze gebruiker
       _gebruikersNaam = gebruikersNaam;

       // haal de taak op
       _taskId = HaalTaak(gebruikersNaam);
   }
   #region ITask Members

   public void Create(Guid taskId)
   {
       // hier wordt je op aangeroepen als het application block
       // een nieuwe task voor je aanmaakt
       _taskId = taskId;
       // bewaar de taak
       BewaarTaak(_gebruikersNaam, taskId);
   }

   public Guid Get()
   {
       return (_taskId);
   }
   #endregion

   private Guid HaalTaak(String naam) ...
   private void BewaarTaak(string naam, Guid taak) ...
}
```

**Codevoorbeeld 5. Eenvoudige implementatie gebruikersnaam als state-identificatie**

applicatie actief. Dat hij bijvoorbeeld zojuist een bepaalde klant heeft gekozen en zich nu bevindt in het scherm KlantOverzicht vormen samen de state van de applicatie. In codevoorbeeld 1 is te zien dat voor de opslag van dit state-object (attribuut statePersist) is gekozen voor een MemoryStatePersistence-provider. Dat betekent dat de state alleen bewaard blijft zo lang de applicatie loopt. Dit is voor een WinForm-applicatie vaak voldoende. Het application block biedt naast deze state-provider echter nog drie standaard implementaties:

- SqlServerPersistState: voor het bewaren van de state in Microsoft SQL Server
- SecureSqlServerPersistState: voor het versleuteld bewaren van de state in Microsoft SQL Server, en
- SessionStatePersistence: voor het bewaren van de state in het Websession-object.

Wanneer we de state in SQL Server bewaren kunnen we de state bewaren, terwijl onze applicatie wordt afgesloten. Doordat de state weer gemakkelijk uit de database is te halen, hebben we onze applicatie herstartbaar gemaakt. Een eindgebruiker kan de applicatie stoppen en afsluiten en even weer exact verder gaan waar hij was gebleven. Om gebruik te maken van deze faciliteit moeten we een drietal zaken regelen.

In eerste instantie zullen we een relatie moeten leggen tussen een bepaalde gebruiker en zijn bijbehorende state. We moeten de state uniek kunnen identificeren en deze kunnen koppelen aan een bepaalde gebruiker. Hoe deze relatie wordt vastgelegd, is uiteraard applicatiespecifiek. Vandaar dat het application block hiervoor slechts een interface aanbiedt en geen standaard implementatie. Het type dat hiervoor wordt geïntroduceerd is ITask. codevoorbeeld 5 toont een eenvoudige implementatie waarin een gebruikersnaam wordt gebruikt als identificatie voor de state.

Het volgende dat we moeten aanpassen is de state-provider. Onze state moet bewaard blijven als de applicatie wordt afgesloten. Zoals we eerder besproken hebben, kunnen we dit realiseren door het statePersist-attribuut in codevoorbeeld 1 te wijzigen in SqlServerPersistState.

De laatste stap die we zetten is het opstarten van de use-case. De aanroep naar UIPManager.StartTask veranderen we als in codevoorbeeld 6.

Sluiten we de applicatie af terwijl we op het scherm KlantOverzicht zijn, en de applicatie opnieuw starten, komen we terug in het scherm zoals we dit hadden achtergelaten.

```
// eenvoudig tbv demonstratiedoeleinden.
// Overweeg het gebruik van de windows identity name
UserTask task = new UserTask("JOE");
UIPManager.StartTask("KlantBeheer", task);
```

**Codevoorbeeld 6. De aanroep naar UIPManager.StartTask veranderen**

## Flexibele user-interface

Het User Interface Process Application Block stelt ons in staat om snel flexibele en goed onderhoudbare user-interfaces te ontwikkelen. De ontkoppeling tussen scherm en logica biedt maximale flexibiliteit en zorgt ervoor dat schermen eenvoudig herbruikbaar zijn in verschillende contexten. Het kunnen bewaren en herstellen van de state met behulp van SQL Server stelt ons in staat een applicatie eenvoudig herstartbaar te maken.

**Referenties**

1 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/AppArchCh2.asp, Application Architecture for .NET: Designing Applications and Services

2 http://msdn.microsoft.com/practices/type/Patterns/Enterprise/DesMVC/ - beschrijving van het model view controller design pattern.

3 Application Programming in Smalltalk-80: use Model-View-Controller (MVC)."University of Illinois in Urbana-Champaign (UIUC) Smalltalk. Available at: http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, Burbeck, Steve.