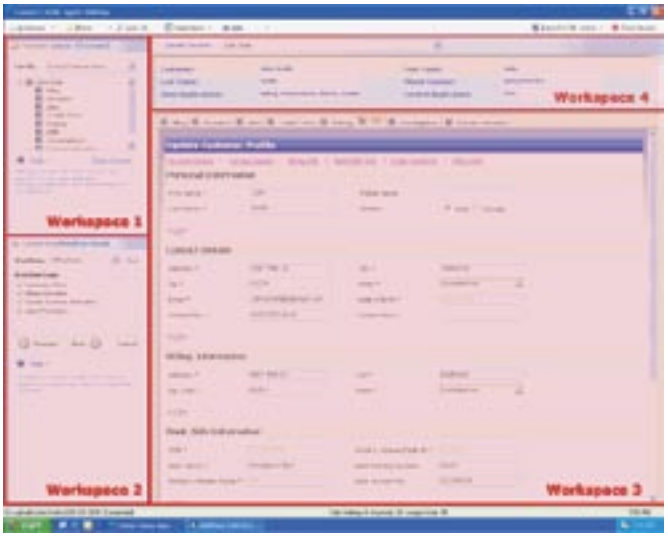
Jonne Kats is .NET-ontwikkelaar bij Ordina

Ontwikkelen van onderhoudbare smart clients

HET COMPOSITE USER INTERFACE APPLICATION BLOCK

Smart Client-applicaties bieden een oplossing voor situaties waarin een gedistribueerde oplossing gewenst is, maar waarin de tegenwoordig zo populaire webinterface niet toereikend is. De code voor de user-interface van deze smart client-applicaties kan echter snel complex worden en het goed structureren hiervan, met oog op onder andere het onderhoud, wordt al snel moeilijk. Het Composite User Interface Application Block (CAB) schiet te hulp door middel van bewezen oplossingen voor de problemen die de ontwikkeling van een smart client-applicatie met zich meebrengt. Dit artikel dient als korte introductie op het CAB en laat zien hoe de ontwikkelaar hiermee aan de slag kan gaan.

Tegenwoordig wordt er steeds vaker voor gekozen systemen een webinterface te geven, vanwege de voordelen voor onder andere het onderhoud en de beschikbaarheid. Wanneer er echter veel interactie en een snelle respons nodig is, kan een windows-user-interface een betere oplossing zijn. Om toch gebruik te kunnen maken van de voordelen van een gedistribueerde oplossing is er de zogenaamde Smart Client. Smart client-applicaties bieden de voordelen van een webapplicatie, maar met de 'look and feel' van een windowsapplicatie. Wanneer een smart client-applicatie uit veel schermen en componenten bestaat, die ook nog eens met elkaar moeten communiceren, neemt de complexiteit van de code snel toe. Zonder een gestructureerde oplossing zal de resulterende code al snel op de bekende spaghetticode lijken. Dit is een van de redenen waarom het Composite UI Application Block (CAB) tot stand is gekomen. Het CAB biedt ondersteuning bij het implementeren van bewezen oplossingen voor de problemen die complexe user-interface-applicaties tot gevolg hebben. Hierdoor ontstaat beter gestructureerde en onderhoudbare code. Daarnaast helpen deze oplossingen of patterns om de complexiteit van de infrastructuur voor de ontwikkelaar zo veel mogelijk te verbergen.



Afbeelding 1. Enkele workspaces waarin SmartParts hun content kunnenweergegeven

Architectuur

In het CAB wordt onderscheid gemaakt tussen de grafische interface (ook wel de shell genoemd) en domeinspecifieke modules. Voorbeelden van deze domeinspecifieke modules kunnen bijvoorbeeld een debiteuren-, crediteuren- en een facturatiemodule zijn. Deze opzet maakt het voor een ontwikkelaar of team mogelijk zich te richten op zijn of hun specialisme, bijvoorbeeld UI, infrastructuur of bepaalde domeinkennis. In de shell kunnen verschillende workspaces worden gedefinieerd, waarin verschillende SmartParts hun content kunnen weergegeven; zie afbeelding 1. Een SmartPart is in feite een Windows Forms user-control die een van de verschillende schermen vertegenwoordigt die in de workspaces getoond moeten worden. De SmartParts bevinden zich in de verschillende domeinspecifieke modules, die in principe als plug-ins worden geladen. In de modules wordt ook gedefinieerd in welke workspace de verschillende SmartParts moeten worden getoond. Naast deze architectuur biedt het CAB ook ondersteuning voor bijvoorbeeld het tonen van module-afhankelijke items in de toolbar, menubar of statusbar, maar ook voor de communicatie tussen modules door middel van events. Daarnaast zijn er mechanismen beschikbaar voor het gebruik van state en het persisteren daarvan.

Aan de slag

Om een indruk te geven hoe met het CAB ontwikkeld wordt, zullen we een opzet maken voor een smart client-applicatie, waarmee onder andere debiteuren kunnen worden getoond. Wanneer we het CAB downloaden, wordt duidelijk dat de code van het CAB eigenlijk een framework is, waarmee de eerder besproken architectuur en mechanismen efficiënt gerealiseerd kunnen worden. In de volgende stappen zullen we een simpele smart client-applicatie maken, die een indicatie geeft van de wijze waarop er met het CAB wordt ontwikkeld. Als eerste moet natuurlijk het CAB-framework worden gedownload, vervolgens maken we een nieuw Windows Forms-project aan en voegen we referenties toe aan de volgende assemblies:

- Microsoft.Practices.Composite.UI
- Microsoft.Practices.Composite.UI.WinForm
- Microsoft.Practices.ObjectBuilder

Met Visual Studio 2005 kan ook de CAB-solution zelf aan het project toegevoegd worden, wat gemakkelijk kan zijn voor het

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI;

namespace Sales
{
   public class ShellWorkItem : WorkItem
   {
   }
}
```
Codevoorbeeld 1.

debuggen van bepaalde problemen. Open nu het form en voeg de *Microsoft.Practices.Composite.UI.WinForm*-referentie toe aan de Visual Studio toolbox (rechtermuisknop op de toolbox en dan 'Choose Items'. Hier zit een aantal controls in dat we gaan gebruiken. Eén daarvan is de ZoneWorkspace, waarin de SmartParts op een bepaalde locatie op het form (de shell) weergegeven kunnen worden. Naast de ZoneWorkspace zijn er nog andere typen workspaces beschikbaar, zoals de MdiWorkspace, TabWorkspace en de WindowWorkspace. De WindowWorkspace laat de SmartParts bijvoorbeeld als aparte fysieke schermen zien. We voegen twee ZoneWorkspace-controls toe aan het form en zetten de DockStyle van één daarvan op Left en van de ander op Fill. We kunnen nu een workitem gaan maken.

Workitems

In een CAB-applicatie wordt onderscheid gemaakt tussen workitems. Een workitem is een runtime-container waarin componenten worden geplaatst die een bepaalde usecase vervullen. In een workitem worden events en state gedeeld en een workitem kan weer sub-workitems hebben. De eerder genoemde domeinspecifieke modules kunnen dus verschillende workitems bevatten. In het geval van een debiteurenmodule zal er waarschijnlijk een workitem voor het toevoegen van een debiteur aanwezig zijn. Een CAB-applicatie bestaat dus meestal uit een hiërarchie van workitems. Een CAB-applicatie heeft op zijn minst één workitem nodig, dus we definiëren een workitem, zie codevoorbeeld 1. In codevoorbeeld 2 definiëren we vervolgens een startpunt voor het CAB. Hiervoor moeten we de Program.cs-file aanpassen die standaard door Visual Studio wordt gegenereerd. We introduceren een nieuwe class die afstamt van FormShellAplication. Door middel van Generics geven we vervolgens een type workitem en Form mee. Het workitem wordt nu door het CAB als root workitem beschouwd en de form is de shell, deze zullen beide door de FormShellApplication worden geïnstantieerd. Wanneer we de applicatie nu compileren en starten, zal deze er uitzien als een normale .NET-applicatie, maar we kunnen nu gebruikmaken van het CAB.

Modules

Zoals eerder genoemd maakt CAB onderscheid tussen de UI en domeinlogica. Deze domeinlogica kan worden ondergebracht in zogenaamde modules, die fungeren als plug-ins en apart met de applicatie gedistribueerd kunnen worden. Voor het maken van een nieuwe module voegen we een class-library aan het project toe en voegen we nogmaals de referenties aan het CAB toe. Deze module zal functionaliteit bieden voor het weergeven en bewerken van debiteuren. We beginnen met het maken van een SmartPart die een lijst van debiteuren zal weergeven in een van onze workspaces. We voegen hiervoor een nieuwe user-control toe, vervolgens geven we met een attribuut aan dat deze user-control een SmartPart is; zie codevoorbeeld 3. Verder plaatsen we met de designer een DataGridView op de user-control, die uiteindelijk de lijst met debiteuren zal weergeven. In codevoorbeeld 4 maken we het workitem dat zal dienen als container voor het tonen van de debiteuren. Zoals afgebeeld 'override' het workitem de OnRunStarted-functie, die wordt aangeroepen zodra het workitem gestart wordt. In deze functie voegt het workitem

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using Microsoft.Practices.CompositeUI.WinForms;

namespace Sales
{
   public class ShellApplication : FormShellApplication<ShellWorkItem, Form1>
   {
      [STAThread]
      public static void Main()
      {
         new ShellApplication().Run();
      }
   }
}
```
Codevoorbeeld 2.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;
using Microsoft.Practices.CompositeUI.SmartParts;

namespace Debiteuren.Module.Views
{
   [SmartPart]
   public partial class DebiteurView : UserControl
   {
      public DebiteurView()
      {
         InitializeComponent();
      }
   }
}
```
Codevoorbeeld 3.

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI;
using Debiteuren.Module.Views;

namespace Debiteuren.Module
{
   public class DebiteurenWorkItem : WorkItem
   {
      protected override void OnRunStarted()
      {
         base.OnRunStarted();

         DebiteurView view = SmartParts.AddNew<DebiteurView>();

         Workspaces["dwsMain"].Show(view);
      }
   }
}
```
Codevoorbeeld 4.

de DebiteurView toe aan zijn eigen SmartPart-collectie. Vervolgens geeft het workitem de opdracht aan een van onze eerder gedefinieerde workspaces met de naam 'dwsMain' om deze view te tonen. Om de module ook daadwerkelijk te gebruiken, moeten we nog een class toevoegen die van ModuleInit zal afstammen. Deze class wordt

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.CompositeUI;
using Microsoft.Practices.CompositeUI.Services;

namespace Debiteuren.Module
{
  public class DebiteurenModuleInit : ModuleInit
  {
    private WorkItem rootWorkItem;

    [ServiceDependency]
    public WorkItem RootWorkItem
    {
      set { rootWorkItem = value; }
    }

    public override void Load()
    {
      base.Load();

      DebiteurenWorkItem workItem = rootWorkItem.WorkItems.AddNew<Debi
                    teurenWorkItem>();
      workItem.Run();
    }
  }
}
```

Codevoorbeeld 5.

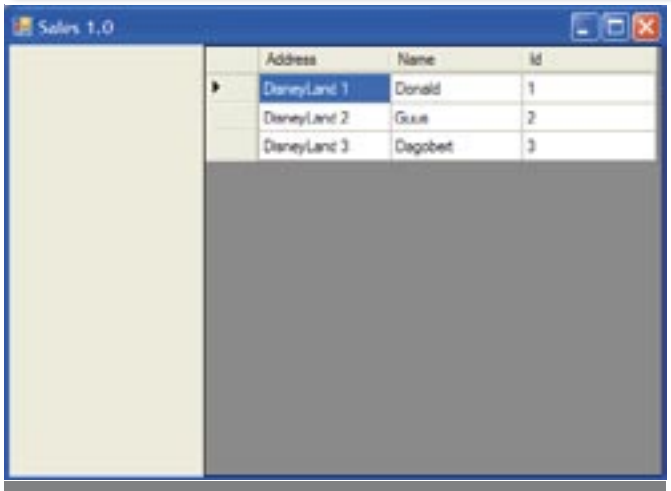
```
<?xml version="1.0" encoding="utf-8" ?>
<SolutionProfile xmlns="http://schemas.microsoft.com/pag/cab-profile" >
  <Modules>
    <ModuleInfo AssemblyFile="Debiteuren.Module.dll"/>
  </Modules>
</SolutionProfile>
```

Codevoorbeeld 6.

```
using System;
using System.Collections;
using System.Text;

namespace Debiteuren.Module.Services
{
  public interface IDebiteurManager
  {
    IList GetDebiteuren();
  }
}
```

Codevoorbeeld 7.



Afbeelding 2. De Cab-applicatie met de lijst van debiteuren.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using Microsoft.Practices.CompositeUI;

namespace Debiteuren.Module.Services
{
  [Service(typeof(IDebiteurManager))]
  public class DebiteurManager : IDebiteurManager
  {
    #region IDebiteurManager Members

    public IList GetDebiteuren()
    {
      ArrayList debiteuren = new ArrayList();

      debiteuren.Add(new Debiteur(1, "Donald", "DisneyLand 1"));
      debiteuren.Add(new Debiteur(2, "Guus", "DisneyLand 2"));
      debiteuren.Add(new Debiteur(3, "Dagobert", "DisneyLand 3"));

      return debiteuren;
    }
    #endregion
  }
}
```

Codevoorbeeld 8.

automatisch aangeroepen zodra de module gestart wordt, deze class is afgebeeld in codevoorbeeld 5. Door middel van een ServiceDependency, waar we zo op terug komen, wordt er een referentie verkregen naar het eerdere gedefinieerde root workitem. Vervolgens wordt ons nieuwe Debiteuren-workitem aan de workitems-collectie van het root workitem toegevoegd en wordt dit uiteindelijk gestart. Onze nieuwe module is nu klaar voor gebruik, we moeten alleen de shell nog van onze debiteurenmodule op de hoogte brengen en dat doen we door middel van een config-file. Voeg een nieuwe XML-file toe aan het Windows Form-project en noem deze ProfileCatalog.xml; zie codevoorbeeld 6. Zet vervolgens in de properties van deze file, Copy to Output Directory op Copy Always. Als nu de applicatie met F5 wordt gestart, laat het CAB inderdaad het DebieteurView SmartPart in de rechterworkspace zien. Allemaal leuk en aardig, maar we missen nog wat functionaliteit, zoals het daadwerkelijk tonen van de debiteuren.

Services

Om debiteuren in de view te laten zien, moeten we een service toevoegen. Een service biedt bepaalde diensten aan onze workitems. In codevoorbeeld 7 definiëren we eerst een interface voor onze service. Door gebruik te maken van een interface kunnen we gemakkelijk een andere implementatie kiezen voor onze service en wordt de code gemakkelijker te testen in combinatie met testframeworks als bijvoorbeeld NMock. In codevoorbeeld 8 staat de implementatie van de service voor het ophalen van de debiteuren uit een fictieve database. Zoals afgebeeld is de DebiteurManager een normale class die een aantal Debiteur-objecten teruggeeft in een ArrayList. Wat opvalt, is het serviceattribuut boven de class. Hiermee maken we aan CAB duidelijk dat dit een service is. CAB zal deze class vervolgens tijdens het opstarten van de applicatie instantiëren, zodat deze klaar is voor gebruik. Wanneer een andere class vervolgens van deze service gebruik wil maken, wordt dit duidelijk gemaakt door middel van een ServiceDependency-attribuut. Het CAB zal vervolgens automatisch de instantie van de service aan de met het ServiceDependency-attribuut gemarkeerde property toekennen. Dit is een vorm van Dependency Injection. Om van de DebiteurManager Service gebruik te maken in onze SmartPart, voegen we in de code van onze DebiteurView een property met het ServiceDependency-attribuut toe, zoals afgebeeld in codevoorbeeld 9. Door middel van het ServiceDependency-

```
private IDebiteurManager debiteurManager;

[ServiceDependency]
public IDebiteurManager DebiteurManager
{
   set
   {
      debiteurManager = value;
   }
}
```

Codevoorbeeld 9.

```
private void DebiteurenView_Load(object sender, EventArgs e)
{
   dataGridView1.DataSource = debiteurManager.GetDebiteuren();
}
```

Codevoorbeeld 10.

attribuut stelt het CAB automatisch de geïnstantieerde service ter beschikking en is deze klaar voor gebruik. Nu kunnen we in het loadevent van de DebiteurView de DataGridView vullen met klanten, zoals afgebeeld in codevoorbeeld 10. Normaal gesproken plaatsen we dit soort code niet in de view zelf, maar zouden we hier het Model-View-Presenter-pattern voor gebruiken. Als we nu vervolgens de applicatie starten, verschijnen de debiteuren netjes in de lijst zie afbeelding 2.

CAB heeft nog meer te bieden

We hebben nu gezien dat het CAB ondersteuning biedt voor het structureren van domeinspecifieke logica in verschillende modules en het maken van workitems als container voor code met een gezamenlijk doel. Ook hebben we gezien dat we gemakkelijk verschillende views of SmartParts kunnen maken die in workspaces van de shell kunnen worden getoond, en dat we services kunnen aanbieden die gemakkelijk door middel van Dependency Injection door workitems kunnen worden gebruikt. Dit is slechts het topje van de ijsberg van wat het CAB te bieden heeft. Als we ook nog zouden ingaan op overige aspecten van het CAB, zoals de Event Broker, State en Instrumentation, dan zouden we daar nog vele pagina's aan kunnen besteden. Al met al biedt het CAB de ontwikkelaar veel ondersteuning bij het maken van complexe smart client-applicaties, waardoor de ontwikkelaar meer tijd heeft om zich te richten op de businessvalue van de applicatie. Het enige nadeel van het CAB is de leercurve, het kost echt wat tijd en energie om de verschillende mechanismen te begrijpen en te kunnen toepassen. Misschien dat dit al is gebleken bij het lezen van dit artikel. Ik hoop desondanks dat ik mensen met dit artikel heb kunnen motiveren om eens wat meer over CAB op te steken, of misschien zelfs te downloaden om zelf een poging te wagen. Het CAB mag dan misschien lastig zijn om onder de knie te krijgen, met de prima 'hands on labs' die worden aangeboden, kan iedereen er al snel mee aan de slag. Het Composite UI Application Block is overigens onderdeel van de Smart Client Factory. Deze Smart Client Factory is eigenlijk een complete verzameling van best practises, voorbeeldcode en tooling die ondersteuning bieden bij het ontwikkelen van smart clients. Naast het CAB biedt de Smart Client Factory nog vele extra's, waarmee de ontwikkelaar nog beter smart clients kan ontwikkelen

Jonne Kats is .NET-ontwikkelaar bij het Development Centre Microsoft van Ordina. Jonne is via mail te bereiken jonne.kats@ordina.nl

Referenties

Smart Client Factory – Patterns and Practises home: http://practices.gotdotnet.com/projects/scbat

Composite UI application Block: http://practices.gotdotnet.com/projects/cab

CAB best practices, Steve Eichert: http://www.emxsoftware.com/Smart+Clients/CAB+Best+Practices

Dependency Injection, Martin Fowler: http://www.martinfowler.com/articles/injection.html

Model View Presenter: http://www.martinfowler.com/eaaDev/uiArchs.html