Composite UI Application Block (CAB)

Lose Kopplung mit CAB

Mit der Smart Client Factory der Pattern and Practices Group von Microsoft ist der Composite UI Application Block erneut ins Rampenlicht gerückt. Dieser Artikel nimmt den Application Block unter die Lupe und wirft einen Blick hinter die Kulissen.

Ein Magier lässt eine Münze in seiner Hand verschwinden und zieht sie dann hinter dem Ohr eines erstaunten Kindes hervor. Ein Erwachsener bemerkt dazu abfällig: „Das ist nur ein Trick. Ganz einfach, wenn man weiß wie's geht." Auch der Composite UI Application Block (CAB) hat für so manchen Entwickler etwas Magisches. Um den Application Block effizient einzusetzen, braucht es jedoch keine Magie, sondern nur ein wenig Verständnis für die Funktionsweise des CAB.

Die Bezeichnung Composite UI Application Block verleitet zu der Annahme, dass der CAB ausschließlich zum Bau von Benutzeroberflächen gedacht ist. Aber obwohl einige seiner Fähigkeiten speziell auf Anwendungen mit Benutzeroberfläche ausgerichtet sind, bietet das Framework auch viele andere Vorteile.

Das wirkliche Herzstück des CAB ist die lose Kopplung (loose coupling). Einerseits findet man sie in den *SmartParts*, *Workspaces* und *UIExtensionSites*. Sie ist aber auch Teil der *Commands*, *States*, *Services*, *Events* und *Modules*, welche unabhängig von einer Benutzeroberfläche einsetzbar sind.

Lose Kopplung ist grundsätzlich eine gute Idee. Sie erleichtert die Arbeit in Teams, den Ausbau von bestehenden Anwendungen oder die Wiederverwendung von Code. Dieser Artikel zeigt, wie Sie dieses Ziel mit dem CAB erreichen.

Tabelle 1

Application-Klassen.

Application-Klasse	Beschreibung
FormShellApplication	Entspricht einer Windows-Forms-Anwendung, die mit Application.Run gestartet wird. Erbt von WindowsFormsApplication.
ApplicationContextApplication	Wie FormShellApplication. Verwendet jedoch einen ApplicationContext. Erbt von WindowsFormsApplication.
WindowsFormsApplication	Abstrakte Basisklasse für CAB-Anwendungen, die die Windows-Forms-Technologie verwenden. Erbt von CabShellApplication.
CabShellApplication	Abstrakte Basisklasse für CAB-Anwendungen, die eine Shell besitzen. Erbt von CabApplication.
CabApplication	Abstrakte Basisklasse für alle CAB-Anwendungen.

Auf einen Blick

Autor

Adrian Krummenacher arbeitet als Softwareingenieur bei bbv Software Services in der Schweiz. Seine Interessenschwerpunkte liegen in den Bereichen UI-Design und Patterns. Sie erreichen ihn unter adrian.krummenacher@bbv.ch.

dotnetpro.code
A0612CAB

Sprachen C#

Technik Composite UI Application Block

Voraussetzungen .NET 2.0, VS 2005, Composite UI Application Block

Einer für alle und alle für einen

In einem eng gekoppelten System erstellen Objekte andere Objekte und greifen direkt auf deren Funktionalität zu. Um diese Kopplung zu lockern, gibt es Design Patterns wie *Abstract Factory*, *Command*, *Dependency Injection* oder *Service Locator* [1]. Der CAB nimmt dem Entwickler einen beträchtlichen Teil der Arbeit ab, da er bereits viele dieser Patterns integriert.

Ein Framework wie der CAB stellt auch Ansprüche. Es diktiert zu einem gewissen Grad die Architektur der Anwendung, was nicht immer wünschenswert ist. Außerdem muss das Framework die Objekte kennen, deren Kopplung es aufheben soll, damit es zwischen ihnen vermitteln kann. Dazu instanziert das Framework die Objekte entweder selbst, oder es wird über Instanzen informiert, die es nicht selbst erstellt hat.

Um das Framework zu starten benötigt man eine Instanz einer CAB-*Application*-Klasse. Hier zuerst eine normale Windows-Forms-Anwendung:

```
public class MyApplication
{ public static void Main()
  { Application.Run(new MyShellForm()); }}
```

Dieses Beispiel öffnet nach dem Start *MyShellForm*. In einer CAB-Anwendung muss das Framework die Instanz von *MyShellForm* kennen. Deshalb wird die *Application*-Klasse als Erstes instanziert.

```
public class MyApplication :
  FormShellApplication<WorkItem, MyShellForm>
{ public static void Main()
  { new MyApplication.Run();  }}
```

Die Methode *Run* der *Application*-Klasse setzt das Framework auf, erstellt eine Instanz von *MyShellForm* und macht diese sichtbar. Dies ist übrigens ein gutes Beispiel, wie der CAB die Verantwortung für die Instanzierung einer Klasse übernimmt. Freilich startet nicht jede Anwendung gleich mit dem Hauptfenster. Darum gibt es verschiedene Typen von *Application*-Klassen, siehe Tabelle 1.

Im Beispiel heißt das Hauptfenster *MyShellForm*, da es als Shell für die CAB-Anwendung dient. Für das Framework ist die Shell eine Art Behälter für die Benutzeroberfläche. Eine häufig anzutreffende

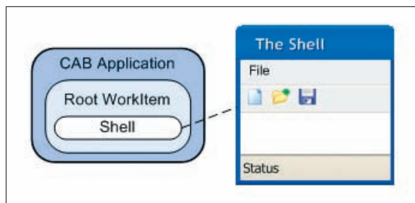
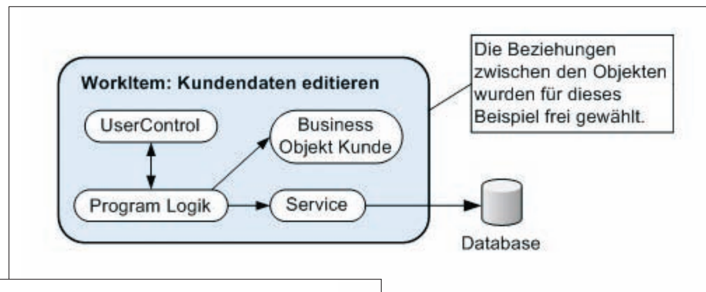


Abbildung 1 Beispiel für eine Shell.

Abbildung 2 Ein WorkItem als Container.



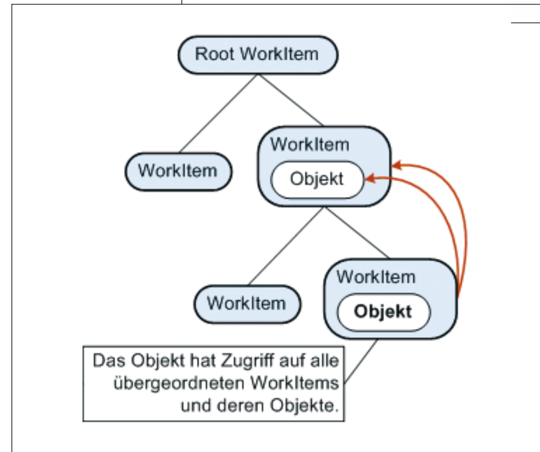


Abbildung 3 Die Baumstruktur der WorkItems.

Shell besteht aus einem Hauptfenster mit einer Menüleiste, einer Werkzeugleiste und einer Statuszeile, wie in Abbildung 1 zu sehen. Innerhalb dieses Fensters werden die verschiedenen Use Cases der Anwendung dargestellt, wie zum Beispiel Mail, Kontakte oder Kalender in Microsoft Outlook.

Der CAB schreibt jedoch nicht vor, wie die Shell auszusehen hat. Es ist sogar möglich, CAB-Anwendungen zu entwickeln, die ganz ohne Shell auskommen. Bei der Shell handelt es sich einfach um den Ort, wo der CAB mit dem Aufbau der Benutzeroberfläche beginnt.

WorkItem

Ein einfaches Framework könnte alle Objekte unter seiner Verwaltung in einer simplen Liste ablegen. Für eine komplexe Anwendung ist das jedoch weder besonders übersichtlich noch effizient. Um Ordnung in das Chaos zu bringen verwendet der CAB das *WorkItem* als Container für eng verwandte Objekte. Ein enthaltenes Objekt kann einfach auf den Container zugreifen und damit auch auf die anderen enthaltenen Objekte. Das *WorkItem* repräsentiert üblicherweise einen Use Case.

Hier ein Beispiel: Ein Entwickler soll den Use Case *Kundendaten editieren* umsetzen. Dazu verwendet er einen Dialog, um die Daten darzustellen, ein Objekt mit der Programmlogik, ein Businessobjekt *Kunde* und einen Service, mit dem er die Kundendaten von einem Backend-System lädt und dahin zurückschreibt. Mit dem CAB würden die entsprechenden Instanzen von einem *WorkItem* verwaltet, welches den Use Case *Kundendaten editieren* repräsentiert, siehe Abbildung 2.

Aber was tun, wenn der Service, welcher die Kundendaten zur Verfügung stellt, auch von anderen *WorkItems* genutzt werden soll? Praktischerweise sind die *WorkItems* im CAB als Objektbaum angeordnet. Jedes *WorkItem* hat damit Zugriff auf die übergeordneten *WorkItems* und deren Objekte. Im Beispiel würde der Entwickler also den Service einem übergeordneten *WorkItem* zuordnen, damit nicht nur *Kundendaten editieren* sondern auch andere *WorkItems* den Service nutzen können. Um ein eigenes *WorkItem* zu schreiben, leitet man von der Basisklasse *WorkItem* ab.

```
public class CustomerWorkItem : WorkItem
```

Nun muss das *WorkItem* wissen, welche Objekte es verwalten soll. Der korrekte Weg nach der offiziellen Microsoft Dokumentation ist, die Methode *OnRunStarted* zu überschreiben.

```
protected override void OnRunStarted()
{ base.OnRunStarted();
  this.Items.AddNew<MyObject>(); }
```

Ein *WorkItem* hat einen von drei Zuständen: *Active*, *Inactive* oder *Terminated*. Nur ein *WorkItem* ist zur selben Zeit aktiv. Wenn dem *WorkItem* ein Benutzeroberflächenelement (Control) zugeordnet ist, aktiviert der CAB das *WorkItem* automatisch, sobald das Control aktiviert wird. *WorkItems* können auch programmgesteuert aktiviert werden.

```
workItem.Activate();
```

Aber Vorsicht, dieser Aufruf von *Activate* wechselt nicht automatisch den Fokus auf das Control des *WorkItems*.

Wird ein *WorkItem* terminiert, entfernt der CAB das *WorkItem* und alle enthaltenen Objekte aus seiner Verwaltung.

Auch das *WorkItem* selbst muss man in die Verwaltung des CAB übergeben.

```
workItem.WorkItems.AddNew<CustomerWorkItem>();
```

Diese Codezeile fügt das *WorkItem* einer Collection des übergeordneten *WorkItems* hinzu. Die Instanz erstellt der CAB selbst. Aber was ist mit dem Root-*WorkItem* des Baums, welches kein übergeordnetes Element hat?

```
public class MyApplication :
  FormShellApplication<WorkItem, MyShellForm>
```

In der Deklaration der *Application Class* wird der Typ des Root-*WorkItems* mitgeteilt. Auch in diesem Fall erstellt das Framework die Instanz. Der CAB benötigt die direkte Kontrolle, weil er dem Root-*WorkItem* Objekte, wie zum Beispiel Standard-Services, hinzufügen muss. Es ist aber auch möglich, ein eigenes *WorkItem* zu verwenden, wenn der entsprechende Typ bei der Deklaration der *Application*-Klasse angegeben wird.

```
public class MyApplication :
  FormShellApplication<MyWorkItem, MyShellForm>
```

Das Framework erstellt und startet das Root-*WorkItem* in seiner Initialisierungsphase.

SmartParts and WorkSpaces

SmartParts sind die visuellen Objekte innerhalb eines *WorkItems*. Die *Smart-*

Parts sind smart, weil sie von jedem beliebigen Typ sein können. Alles was das Framework dazu braucht, ist eine Klasse, die den Typ des *SmartParts* darstellen kann. Diese Klasse heißt *Workspace*. Der *Workspace* entfernt die enge Kopplung zwischen der visuellen Komponente und dem Objekt, welches die visuelle Komponente darstellt. Das darstellende Objekt kennt nur den *Workspace* und der wiederum weiß, wie der *SmartPart* dargestellt wird, siehe Abbildung 4.

Das übliche Beispiel ist die Trennung der Shell vom Use Case. Der Shell-Entwickler erstellt mit *Workspaces* nur das Layout. Wenn ein *WorkItem* in der Shell dargestellt werden soll, werden die *SmartParts* nicht direkt der Shell hinzugefügt, sondern dem *WorkSpace*. Das erlaubt eine beinahe unabhängige Entwicklung der Shell und der *WorkItems*, siehe Abbildung 5. Hier wird der *SmartPart anyControl* im *Workspace MyWorkspace* der Shell dargestellt:

```
this.Workspaces["MyWorkspace"].Show(anyControl);
```

Die *Workspaces*, welche zurzeit im CAB-Paket ausgeliefert werden, sind auf Windows-Forms-Anwendungen ausgerichtet und unterstützen darum nur *SmartParts* vom Typ *System.Windows.Forms.Control*. Um Benutzeroberflächenelemente zu unterstützen, welche nicht von *Control* ableiten, muss der Entwickler seine eigenen *Workspaces* schreiben.

Aber was ist der Unterschied zwischen einem *SmartPart* und einem normalen Control? Ein *SmartPart* ist ein Control, das einem *WorkItem* hinzugefügt wurde. Wie bereits erwähnt hat ein Objekt eines *WorkItems* Zugriff auf andere Objekte innerhalb desselben *WorkItems*. Ein *SmartPart* kann im Gegensatz zu einem normalen Control von dieser Tatsache profitieren. Zusätzlich aktiviert das Framework ein *WorkItem* automatisch, wenn der zugehörige *SmartPart* aktiviert wird. Auch das funktioniert nur, weil sich *SmartPart* und *WorkItem* kennen. Das Control wird dem *WorkItem* mit *AddNew* hinzugefügt.

```
CustomerView view =
this.SmartParts.AddNew<CustomerView>();
```

Erstaunlicherweise enthält die Collection nach dem Ausführen dieser Codezeile nur dann einen neuen *SmartPart*, wenn die entsprechende Klasse mit dem Attribut *SmartPart* markiert ist.

```
[SmartPart]
public class CustomerView : UserControl
```

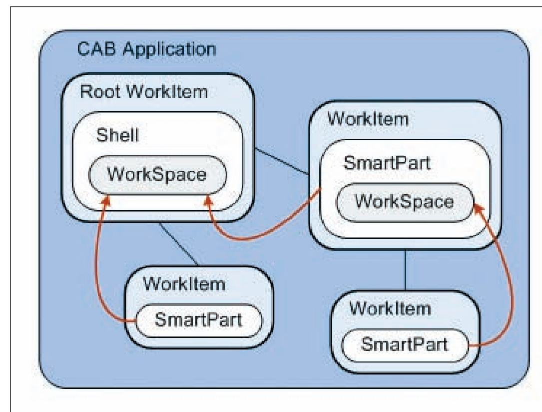


Abbildung 4 SmartParts und Workspaces im Objektbaum.

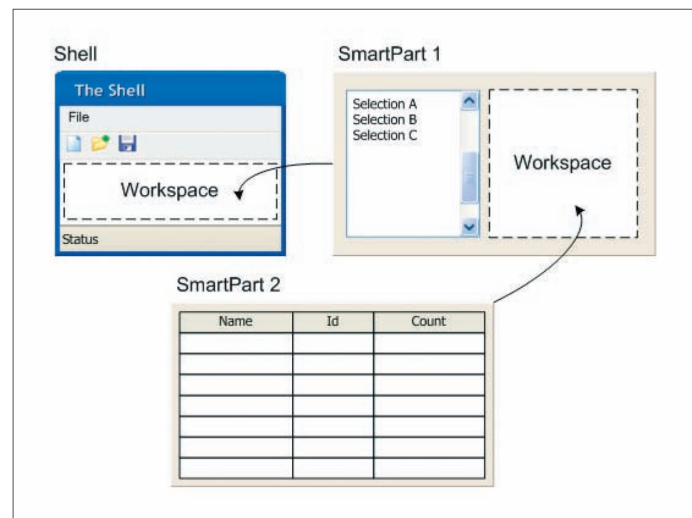


Abbildung 5 Eine Shell zusammensetzen.

Enthält der *SmartPart* weitere Objekte, die mit diesem Attribut markiert sind, werden sie auch der *SmartPart*-Collection hinzugefügt. Zusätzlich zu den *Workspaces* stellt der CAB auch das *SmartPartPlaceHolder*-Control zur Verfügung. Mit dem *SmartPartPlaceHolder* ist es möglich, einen einzelnen *SmartPart* an einer bestimmten Position darzustellen. Aber obwohl der *SmartPartPlaceHolder* wie ein sehr einfacher *Workspace* aussieht, arbeitet er genau anders herum. Während ein *SmartPart* den *Workspace* kennen muss, kennt der *SmartPartPlaceHolder* den Namen des *SmartParts*, den er darstellen soll. Existiert ein *SmartPart* mit passendem Namen, wird er automatisch in den *SmartPartPlaceHolder* geladen. Hier ein Beispiel, wie der Name des *SmartParts* gesetzt wird:

```
this.SmartParts.AddNew<CustomerView>(
  "CustomerView");
```

State

Der *State* eines *WorkItems* ist nichts weiter als ein Verzeichnis von Objekten, siehe Abbildung 6. Einen *State* hinzufügen ist einfach:

```
workItem.State["Customer1"] = customer;
```

Das *WorkItem* erstellt automatisch einen neuen *State* mit dem Namen *Customer1*. Nun kann man auf den *State* mit seinem Namen zugreifen.

```
Customer customer = (Customer)State["Customer1"];
```

Ein *SmartPart* könnte den *State* über das *WorkItem* erreichen.

Es gibt aber noch einen einfacheren Weg.

```
[State]
public Customer Customer
{ set { customer = value; }}
```

Mit dem Attribut *State* teilt man dem Framework mit, dass es Dependency Injection verwenden soll, um dem *SmartPart* Zugriff auf den *State* zu geben. Wenn das Framework auf ein Objekt unter seiner Verwaltung trifft, welches das Attribut *State* verwendet, sucht es im zugehörigen und allen übergeordneten *WorkItems* nach einem *State* mit einem passenden

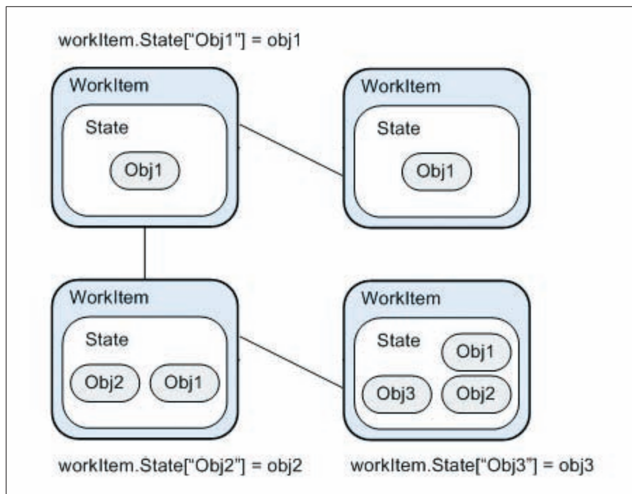


Abbildung 6 WorkItems und States.

Typ und setzt die markierte Property. Gibt es mehr als einen passenden *State*, ist ein weiterer Hinweis nötig.

```
[State("Customer1")]

public Customer Customer
{ set { customer = value; }}
```

Der *State* ist der Teil des *WorkItems*, welcher bei einem Aufruf der Methoden *Save* und *Load* des *WorkItems* gespeichert und geladen wird. Dazu benötigt der CAB jedoch einen Service, der das Interface *IStatePersistenceService* implementiert. Der CAB stellt bereits die Services *StreamStatePersistenceService*, *FileStatePersistenceService* und *IsolatedStorageStatePersistenceService* zur Verfügung.

Service

Ein Service ist im Grunde ein Objekt, welches von vielen anderen Objekten in der Anwendung benutzt wird. Es kann sich dabei zum Beispiel um einen Logger handeln oder einen Service, welcher den Zugriff auf ein Backend-System zur Verfügung stellt. Es gibt jedoch aus Sicht des CAB keine spezifischen Anforderungen an einen Service. Ähnlich wie der *SmartPart* wird ein Objekt zum Service, indem man es der *Service*-Collection des *WorkItems* hinzufügt.

Es gibt drei Wege einen Service hinzuzufügen. Der erste verwendet die Konfigurationsdatei *app.config* der Anwendung.

In Listing 1 wird ein CAB-Service hinzugefügt, den das Framework nicht von Haus aus installiert. Typ und Assembly

spardorado.de

Bei Fachbüchern 50 % und mehr sparen!

Toptitel des Monats

ADO.NET
Grundlagen und Profiwissen

Programmierung von Datenbankanwendungen mit C# und VB.NET
Detlev Wanzke, Lothar Wanzke
Hanser
ca. 770 Seiten, 1 CD
H2157

statt 49,90 € nur
24,95 €

Weitere Datenbanktitel:

Jetzt lerne ich ADO.NET
Ralf Westphal, Christian Weyer
ca. 450 Seiten, 1 CD
P6229
statt 24,95 € nur 12,95 €

SQL lernen
Michael Ebner
ca. 270 Seiten, 1 CD
P2025
statt 24,95 € nur 12,95 €

MySQL Tutorial
Luke Welling, Laura Thomson
ca. 280 Seiten
P2169
statt 29,95 € nur 14,95 €

Ihre erste Adresse für ■ Remittenden ■ Messerückläufer ■ Sonderausgaben ■ Ladenpreisaufhebung

Jetzt schnell ins Internet auf **www.spardorado.de** dem Online-Bookshop in Kooperation mit dotnetpro

Listing 1

Services in der Datei App.config.

```
<configuration>
  <configSections>
    <section name="CompositeUI" type="Microsoft.Practices.CompositeUI.Configuration.
        SettingsSection, Microsoft.Practices.CompositeUI" />
  </configSections>
  <CompositeUI>
    <services>
      <add serviceType="Microsoft.Practices.CompositeUI.Services.IStatePersistenceService,
        Microsoft.Practices.CompositeUI" instanceType="Microsoft.Practices.CompositeUI.
          Services.IsolatedStorageStatePersistenceService, Microsoft.Practices.CompositeUI" />
    </services>
  </CompositeUI>
</configuration>
```

definiert durch *instanceType* sind notwendig, damit das Framework über Reflection eine Instanz des Services erstellen kann. Der *serviceType* ist optional und kommt zum Einsatz, wenn der Service unter einem besonderen Typ publiziert werden soll. Darf der Anwender des Services dessen Implementierung nicht kennen, verwendet man hier üblicherweise ein Interface.

Im Beispiel greift die Anwendung nur mit dem Interface *IStatePersistenceService* auf den Service zu. Damit ist es möglich, den Service über die Konfigurationsdatei auszutauschen, ohne eine Zeile Code zu ändern.

Man kann einen Service auch deklarativ hinzufügen.

```
[Service]
public class CustomerDetailService
```

Findet der CAB einen Typ markiert mit *[Service]*, erstellt er automatisch eine Instanz und fügt sie dem Root-*WorkItem* hinzu. Damit wird der Service allen *WorkItems* zugänglich. Auch hier ist es möglich, den Service unter einem anderen Typ zu publizieren.

```
[Service(typeof(ICustomerDetailService))]
public class CustomerDetailService
```

Der Service kann auch nur nach Bedarf erstellt werden.

```
[Service(AddOnDemand=true)]
public class CustomerDetailService
```

Und schließlich gibt es auch noch den programmgesteuerten Weg.

```
this.Services.AddNew<CustomerDetailService>();
this.Services.AddNew<CustomerDetailService,
   ICustomerDetailService>();
this.Services.AddOnDemand<CustomerDetailService>();
```

Auf diese Art ist es übrigens möglich, einen Service einem anderen *WorkItem* als dem Root-*WorkItem* zuzuordnen.

Nun benötigt der Code Zugriff auf den Service. Gelangt man an ein *WorkItem*, erreicht man so auch den Service.

```
this.Services.Get<ICustomerDetailService>();
```

Das Attribut *ServiceDependency* ermöglicht den Zugriff über Dependency Injection.

```
private CustomerDetailService detailService;
[ServiceDependency]
public CustomerDetailService DetailService
{ set { detailService = value; }}
```

Der CAB kann eine mit *[ServiceDependency]* markierte Property jedoch nur füllen, wenn ihr Objekt einem *WorkItem* hinzugefügt wurde, das unter der Verwaltung des CAB steht.

Der CAB fügt dem Root-*WorkItem* beim Start einige Standard-Services hinzu. Man kann diese jedoch auch gegen eigene Services ersetzen, indem man dem Root-*WorkItem* einen Service hinzufügt, der dasselbe Interface wie ein Standard-Service implementiert.

Module

UserControls werden zu *SmartParts* und Objekte zu Services, wenn sie vom CAB verwaltet werden. Mit Modulen ist es ganz ähnlich. Ein Modul ist eine Assembly, die in einer XML-Katalogdatei eingetragen ist.

Das Framework lädt alle Assemblies in diesem Katalog in die Anwendung. Dann sucht es in der Assembly nach Klassen, die *IModule* implementieren, erstellt Instanzen davon und führt auf diesen Instanzen *AddServices* und *Load* aus. Aber was ist der Sinn eines Moduls?

Normale Assemblies sind üblicherweise eng gekoppelt, wenn eine Assembly eine andere Assembly referenziert. Module hingegen werden vom CAB geladen und greifen nicht direkt aufeinander zu, sondern verwenden dazu ebenfalls den CAB. Das Resultat ist die lose Kopplung von Assemblies. Mit dem richtigen Design kann der Entwickler Funktionalität hinzufügen, ersetzen oder entfernen, ohne eine Zeile Code zu ändern. Stattdessen passt er nur die Katalogdatei an.

Module enthalten ganze *WorkItems*, einzelne *SmartParts* oder Services. Die Services werden in *AddServices* und die anderen Objekte in *Load* hinzugefügt, siehe Abbildung 7.

Ein Modul kann für ein eigenständiges Feature zuständig sein. In diesem Fall muss es nur in die Shell eingebunden werden. Dazu bietet der CAB *UIExtensionSites*. Es kann jedoch auch sein, dass das Modul einen Infrastruktur-Service wie zum Beispiel einen Logger enthält, auf den die Anwendung direkt zugreifen möchte. Dann publiziert man den Service über ein Interface in einer separaten Assembly. So bleibt das Modul lose gekoppelt und kann ausgetauscht werden, solange das Interface gleich bleibt.

Um aus einer Assembly ein Modul zu machen, erstellt man eine XML-Katalogdatei, siehe Listing 2, und fügt den Namen der Assembly hinzu. Der Standardname der Katalogdatei ist *profilecatalog.xml* und ihr Standardort das Hauptverzeichnis. Benötigt das Modul eine Initialisierung muss es eine Klasse enthalten, die *IModule* implementiert. Statt *IModule* zu implementieren, kann man auch die Klasse *ModuleInit* verwenden, die eine leere Implementation des Interfaces bereitstellt. Das Modul erhält einen Namen, indem man das folgende Attribut der *AssemblyInfo*-Klasse hinzufügt.

```
[assembly: Microsoft.Practices.CompositeUI.
  Module("MyModule")]
```

Der Name ist nicht zwingend. Mit dem Namen kann das Modul identifiziert werden, wenn sich die Anwendung zum Beispiel auf den Event *ModuleLoaded* von mehreren Modulen registriert. Außerdem ist der Name notwendig, wenn man Abhängigkeiten zwischen Modulen angeben will.

```
[assembly: Microsoft.Practices.CompositeUI.
  ModuleDependency( "MyModule")]
```

Ein Modul mit diesem Attribut wird immer nach dem Modul mit dem Namen *MyModule* geladen. Der CAB löst eine Exception aus, wenn kein passendes Modul in der Katalogdatei eingetragen ist.

Eine Katalogdatei mit einem eigenen Namen oder an einem besonderen Ort kann in *app.config* konfiguriert werden, siehe Listing 3.

Es ist sogar möglich, ein Modul von einer anderen Datenquelle als einer Datei zu laden, zum Beispiel von einer Datenbank. Dazu implementiert der Entwickler das Interface *IModuleEnumerator* für diese Datenquelle und fügt es in *app.config* hinzu, siehe Listing 4.

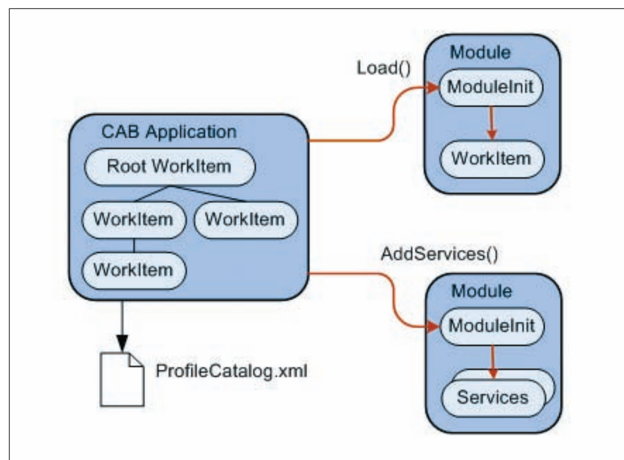


Abbildung 7 Module laden.

UIExtensionSites

Stellen Sie sich vor, ein Entwickler entwirft ein *WorkItem*, welches einen *SmartPart* verwendet um die Zeit anzuzeigen. Um eine bestehende Anwendung damit zu erweitern, schreibt er ein Modul und fügt das *WorkItem* dem CAB-Objektbaum hinzu. Dies geschieht in der *Load*-Methode der Klasse *ModuleInit*.

Nun möchte er in der Menüleiste der Anwendung für das *WorkItem* gerne einen neuen Menüpunkt *Uhr anzeigen*. Die Menüleiste zu erweitern, obwohl das *WorkItem* die Shell gar nicht kennen darf, klappt mit einer *UIExtensionSite*.

```
protected override void AfterShellCreated()
{ base.AfterShellCreated();
  rootWorkItem.UIExtensionSites.
  RegisterSite("MainMenu", Shell.
  MainMenuStrip);}
```

Die Anwendung registriert eine *UIExtensionSite* für die Menüliste im *RootWorkItem*. Nun kann der Entwickler die

spardorado.de

Bei Fachbüchern 50 % und mehr sparen!

Toptitel von Wintellect Know how. bei Microsoft Press

Applied .NET Framework Programming Collection
statt 70,00 € nur 29,95 €

C# Edition - including Training Video and Poster Pack
Jeffrey Richter, Microsoft Press
ca. 640 Seiten, Schuber, 1 CD, Poster
MS1975

Programming ASP.NET
Dino Esposito
MS1903
statt 60,00 € nur 29,95 €

Programming Visual Basic .NET
Francesco Balena
MS1375
statt 60,00 € nur 29,95 €

Building Web Solutions with ASP.NET and ADO.NET
Dino Esposito
MS1578
statt 50,00 € nur 24,95 €

Applied .NET Framework Programming in VB.NET
Jeffrey Richter, Francesco Balena
MS1787
statt 50,00 € nur 19,95 €

Applied XML Programming for Microsoft .NET
Dino Esposito
MS1801
statt 50,00 € nur 24,95 €

Ihre erste Adresse für ■ Remittenden ■ Messerückläufer ■ Sonderausgaben ■ Ladenpreisaufhebung

Jetzt schnell ins Internet auf **www.spardorado.de** dem Online-Bookshop in Kooperation mit dotnetpro

Listing 2

ProfilCatalog.xml.

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

Listing 3

Namen und Ort der Katalogdatei überschreiben.

```
<services>
  <add serviceType="Microsoft.Practices.CompositeUI.Services.IModuleEnumerator,
     Microsoft.Practices.CompositeUI" nstanceType="Microsoft.Practices.CompositeUI.Services.
       FileCatalogModuleEnumerator, Microsoft.Practices.CompositeUI"
         filePath="MyCatalogFile.xml"/>
</services>
```

Listing 4

Modulkatalog aus Datenbank.

```
<services>
  <add serviceType="Microsoft.Practices.CompositeUI.Services.IModuleEnumerator,
     Microsoft.Practices.CompositeUI" instanceType="Microsoft.Practices.CompositeUI.Services.
       MyDBCatalogModuleEnumerator, Microsoft.Practices.CompositeUI" connectionString="..."/>
</services>
```

Menüleiste von jedem *WorkItem* im CAB-Objektbaum ansprechen, ohne die Shell selbst zu referenzieren.

```
ToolStripMenuItem clockMenu = new
   ToolStripMenuItem("Zeit");
clockWorkItem.UIExtensionSites["MainMenu"].
   Add(clockMenu);
ToolStripMenuItem menuItem = new
   ToolStripMenuItem("Uhr anzeigen");
clockMenu.DropDownItems.Add(menuItem);
```

Die *UIExtensionSite* lockert die Kopplung zwischen einem *WorkItem* und einem Element der Benutzeroberfläche, das nicht direkt unter der Kontrolle des *WorkItems* steht. Mit der *UIExtensionSite* kann ein *WorkItem* die Shell oder auch fremde *SmartParts* dynamisch erweitern, ohne sie direkt zu kennen.

UIExtensionSites können in jedem *WorkItem* registriert werden, das visuelle Komponenten hat. Zurzeit unterstützt der CAB jedoch nur *UIExtensionSites* für *ToolStrips*, also für Menüleisten, Werkzeugleisten und Statuszeilen. Es ist jedoch möglich andere Arten von *UIExtensionSites* zu unterstützen, indem der Entwickler einen eigenen Adapter (*IUIElementAdapter*) und eine eigene Factory (*IUIElementAdapterFactory*) implementiert.

Command

Eine Benutzeroberfläche bietet häufig mehr als einen Weg, um eine bestimmte Aktion auszuführen. Die Aktion *Save* findet man üblicherweise in der Menüleiste, in der Werkzeugleiste oder sogar in einem Kontextmenü. Der CAB bietet dafür *Commands*, die die verschiedenen Auslöser derselben Aktion zuordnen.

```
[CommandHandler("EditCustomer")]
public void OnCustomerEdit(object sender,
  EventArgs args)
```

Eine so markierte Methode wird ausgeführt, wenn *EditCustomer* ausgelöst wird. Wie immer gilt, dass das Framework das Objekt mit dem *Command Handler* kennen muss. Der *Command Handler* muss also zu einem *WorkItem* oder zu einem in einem *WorkItem* enthaltenen Objekt gehören, siehe Abbildung 8. Nun benötigt der *Command* einen oder mehrere Auslöser. Das *WorkItem* besitzt dazu eine *Command*-Collection.

```
Commands["EditCustomer"].AddInvoker(
  editCustomerMenuItem, "Click");

Commands["EditCustomer"].AddInvoker(
  editCustomerToolbarButton, "Click");
```

Im Beispiel sind die *Invoker* Events der Menü- und der Werkzeugleiste. Ein *Command* kann auf beliebige Art ausgelöst werden, vorausgesetzt eine entsprechende Adapterklasse ist beim *CommandAdapterMapService* registriert. Der CAB selbst registriert den *ControlCommandAdapter* und den *ToolStripItemCommandAdapter*, die beide von *EventCommandAdapter* ableiten. Das bedeutet, dass die Typen *Control* und *ToolStripItem* Auslöser sein können und dass zur Auslösung der Event-Mechanismus von .NET verwendet wird. Der *EventCommandAdapter* registriert einen Event Handler für jeden durch *AddInvoker* angegebenen Event. Auf der anderen Seite werden alle *Command* Handler als Event Handler bei einem Event des Adapters registriert. Wenn ein *Invoker* seinen Event auslöst, wird dieser vom entsprechenden Event Handler im Adapter empfangen. Der Adapter löst dann seinen eigenen Event aus, was zur Ausführung aller *Command* Handlers führt. Dies erlaubt eine *n-m*-Beziehung zwischen *Invoker* und *Command* Handler.

Der Einsatz von *Commands* hebt eine weitere enge Kopplung zwischen der Shell und den *WorkItems* oder den Modulen auf. Da sich das *WorkItem* beim Adapter für den *Command* registriert, muss es den Auslöser selbst nicht kennen. *Commands* beschränken sich jedoch nicht auf die Shell. Sie können überall eingesetzt werden, wo man einen Event und einen Event Handler lose koppeln will. *Commands* kann man auch programmgesteuert auslösen.

```
Commands[("MyCommand"].Execute();
```

Events

Komponenten einer CAB-Anwendung kommunizieren über Events. Der normale Event-Mechanismus von .NET genügt jedoch nicht, wenn die Komponenten lose gekoppelt sein sollen. Darum verwendet der CAB einen Event Broker. Als Erstes wird ein Event publiziert.

```
[EventPublication("topic://Customer/
  UpdateAddress")]
public event EventHandler<DataEventArgs<string>>
  UpdateAddressEvent;
```

Dann kann man sich als Empfänger dieses Events registrieren.

```
[EventSubscription("topic://Customer/
  UpdateAddress")]
public void OnUpdateCustomerAddress(object sender,
  DataEventArgs<string> e)
{ ... }
```

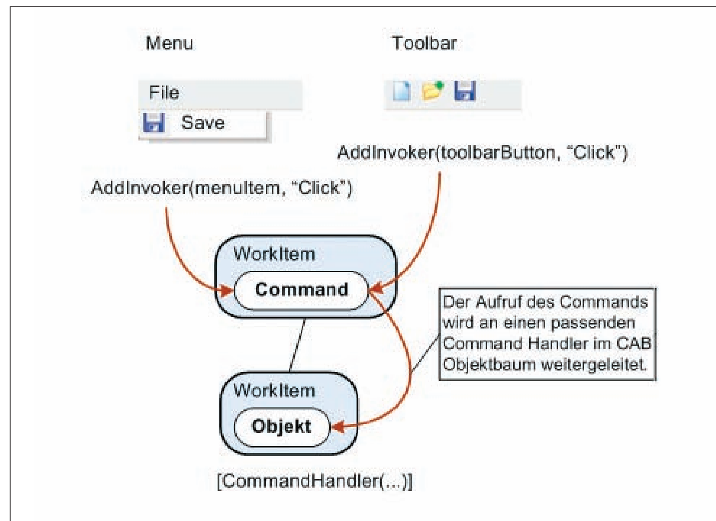
Der Event wird durch eine Bezeichnung, genannt *EventTopic*, identifiziert. Die Bezeichnung kann ein beliebiges Format haben. Das URI-Format wird jedoch empfohlen. Es ist allgemein bekannt und kann kontextabhängige Informationen enthalten, wie *Customer* in dem Beispiel. Im Programm sieht das so aus:

```
EventTopics["topic://Customer/
  UpdateAddress"].AddPublication(...)

EventTopics["topic://Customer/
  UpdateAddress"].AddSubscription(...)
```

Auch für die CAB-Events gilt: Der Publisher und der Subscriber müssen unter der Verwaltung des CAB stehen. Das übliche Objekt im CAB-Objektbaum ist nur für übergeordnete Objekte sichtbar. Der CAB-Event ist etwas flexibler.

Abbildung 8
Commands.



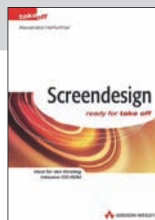

```
[EventPublication("topic://Customer/
  UpdateAddress", PublicationScope.Global)]
```

Dieser Event ist ausdrücklich für alle *WorkItems* sichtbar, unabhängig davon, wo sie im CAB-Objektbaum angeordnet sind. Benutzt man stattdessen *PublicationScope.Descendants*, ist der Event nur für das publizierende *WorkItem* und dessen Kinder sichtbar. Mit *PublicationScope.WorkItem* kann man sich nur innerhalb desselben *WorkItems* für den Event

spardorado.de

Bei Fachbüchern 50 % und mehr sparen!

Praxiswissen Projektmanagement
Oliver Gassmann
Hanser
ca. 220 Seiten
H2809
statt 24,90 € nur **12,95 €**



Screendesign
Alexandra Herfurtner
Addison-Wesley
ca. 250 Seiten
P2065
statt 19,95 € nur **9,95 €**

Vertragsrecht für IT-Fachleute
Christoph Zahrnt
Hüthig
ca. 430 Seiten
MI5026
statt 49,90 € nur **19,95 €**

Keine Angst vor der Akquise!
Jürgen Ratzkowski
Hanser
ca. 230 Seiten
H2700
statt 19,90 € nur **9,95 €**

Wissensmanagement
Bettina Trauner
Hanser
ca. 130 Seiten
H1729
statt 9,95 € nur **4,95 €**

Das Webpflichtenheft
Claudia Lettau
mitp
ca. 500 Seiten
MI1349
statt 49,00 € nur **24,95 €**

Ihre erste Adresse für ■ Remittenden ■ Messerückläufer ■ Sonderausgaben ■ Ladenpreisaufhebung

Jetzt schnell ins Internet auf **www.spardorado.de** dem Online-Bookshop in Kooperation mit dotnetpro

DataEventArgs

DataEventArgs ist ein generischer Typ des Frameworks. Wenn es nur darum geht, mit den EventArgs ein Objekt von einem bestimmten Typ zu übermitteln, lässt sich DataEventArgs mit diesem Typ deklarieren. Der Zugriff erfolgt über die Property Data.

registrieren. Mit CAB-Events ist es auch sehr einfach zu bestimmen, in welchem Thread der Event Handler ablaufen soll.

```
[EventSubscription("topic://Customer/
  UpdateAddress", ThreadOption.Publisher)]
```

Dies ist das Standardverhalten des .NET-Event-Mechanismus. Der Event Handler wird im selben Thread ausgeführt, in dem auch der Event ausgelöst wurde.

```
[EventSubscription("topic://Customer/
 UpdateAddress", ThreadOption.Background)]
```

Der Event wird in einem Background-Thread ausgeführt.

```
[EventSubscription("topic://Customer/
 UpdateAddress", ThreadOption.UserInterface)]
```

Der Event wird im UI-Thread ausgeführt, praktisch, wenn ein Nicht-UI-Thread einen *SmartPart* aktualisieren muss.

Der CAB-Event hat also verglichen mit dem *Command* einige Vorteile. Wieso verwendet man dann nicht nur Events? *Command* kann auf einen Event einer Systemkomponente reagieren.

```
Commands["EditCustomer"].
  AddInvoker(editCustomerMenuItem, "Click");
```

Für einen CAB-Event ist das schwieriger, da man nicht direkt den *Click*-Event publizieren kann.

Aber natürlich gibt es trotzdem Entwickler, die gerne die Vorteile der *Commands* und der CAB-Events nutzen möchten.

Dafür gibt es den *EventTopicCommand*.

```
Commands.AddNew<EventTopicCommand>("EditCustomer");
Commands["EditCustomer"].AddInvoker(
  editCustomerMenuItem, "Click");
```

Mit einem *EventTopicCommand* ist es möglich, einen *Command* mit einem CAB-Event zu verknüpfen. Die Bezeichnung des Events muss dazu nur ein bestimmtes Format aufweisen: *topic://EventTopicCommand/...*

```
[EventSubscription("topic://EventTopic
  Command/EditCustomer")]
```

Woher kommen die Instanzen?

Die Objekte eines *WorkItems* kennen sich untereinander und auch alle übergeordneten *WorkItems* und deren Objekte. Den Zugriff erhalten sie mit Hilfe von Dependency Injection. Der Artikel ist bereits auf den Zugriff auf *States* mit dem *State* Attribut und auf *Services* mit dem *ServiceDependency*-Attribut eingegangen. Um an andere Objekte zu gelangen, muss man sich direkt an den ObjectBuilder wenden.

Der ObjectBuilder ist zuständig für Instanzen im CAB. Der eine oder andere Entwickler kennt ihn wahrscheinlich bereits von der Enterprise Library. Das Thema ObjectBuilder reicht für einen eigenen Artikel und wurde bereits in einer vorangegangenen Ausgabe [2] angeschnitten. Darum beschränkt sich dieser Artikel auf einige Informationen zum Injection-Mechanismus.

Mit dem Attribut *Dependency* kann eine Property dem ObjectBuilder mitteilen, dass es mit der nächsten passenden Instanz im CAB-Objektbaum gesetzt werden möchte.

```
[Dependency]
public WorkItem MyWorkItem
{ set { workItem = value; }}
```

Für den Fall, dass keine passende Instanz vorhanden ist, kann man definieren, wie der ObjectBuilder reagieren soll.

```
[Dependency(NotPresentBehavior=
  NotPresentBehavior.CreateNew)]
[Dependency(NotPresentBehavior=
  NotPresentBehavior.ReturnNull)]
[Dependency(NotPresentBehavior=
  NotPresentBehavior.Throw)]
```

Der ObjectBuilder kann auf zwei Arten nach dem Objekt suchen: Nur lokal oder den Objektbaum aufwärts.

```
[Dependency(SearchMode=SearchMode.Local)]
[Dependency(SearchMode=SearchMode.Up)]
```

Wenn der Typ nicht ausreicht, um das Objekt zu identifizieren, muss es einen Namen haben, den der ObjectBuilder in seiner Suche verwenden kann.

```
[Dependency(Name="MyObject")]
```

Model-View-Controller

Im CAB repräsentieren der State das Model und der SmartPart den View. Für den Controller gibt es sogar eine eigene Klasse Controller. Diese ist jedoch nicht besonders spektakulär ausgefallen. Alles was sie bietet, sind zwei Properties für den direkten Zugriff auf das WorkItem und den State. Die Umsetzung des MVC-Patterns liegt beim Entwickler.

Der ObjectBuilder kann auf Wunsch auch neue Instanzen erstellen.

```
[CreateNew]
public MyController Controller
{ set { controller = value; }}
```

Diese Beispiele verwenden alle *Property Injection*. Der ObjectBuilder unterstützt jedoch auch *Constructor Injection* und *Method Injection*. Hier noch ein Beispiel mit *Constructor Injection*.

```
public MyClass
{ private WorkItem workItem;
  public MyClass([Dependency] WorkItem workItem)
  { this.workItem = workItem; }
}
```

Man darf jedoch nicht vergessen, dass ein Objekt, das durch den ObjectBuilder mit Instanzen versehen werden soll, bereits unter der Verwaltung des ObjectBuilders stehen muss.

Fazit

Der Composite UI Application Block setzt das Konzept der losen Kopplung konsequent um und ist darum mehr als nur ein Framework für modulare Benutzeroberflächen. Wer den CAB einsetzt, hat zwar kein Anrecht auf die offizielle Unterstützung von Microsoft, der Sourcecode ist aber beinahe ebenso wertvoll. Und selbst wenn der CAB nicht zum Einsatz kommt, bietet er doch eine Fülle von Ansätzen zur losen Kopplung, welche man in das eigene Design übernehmen kann. ||||||

[1] Martin Fowler, Inversion of Control Containers and the Dependency Injection Pattern, www.martinfowler.com/articles/injection.html
[2] Stephan Volmer, Verstecktes Juwel, Der Object Builder Application Block, dotnetpro 9/2006, Seite 130 ff.