NET Architecture CAMP

Jörg Neumann

Flexible Smart-Client-Architekturen für .NET-Desktop-Entwickler

Die Clientseite



Jörg Neumann

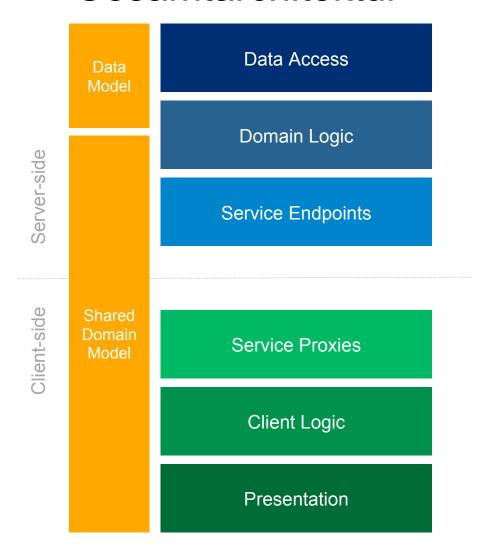


- Principal Consultant bei Acando
- Associate bei Thinktecture
- MVP "Windows Platform Development"
- Beratung, Training, Coaching
- Buchautor, Speaker
- Themen: Xamarin, Windows Store Apps, WPF
- Mail: <u>Joerg.Neumann@Acando.de</u>
- Blog: <u>www.HeadWriteLine.BlogSpot.com</u>



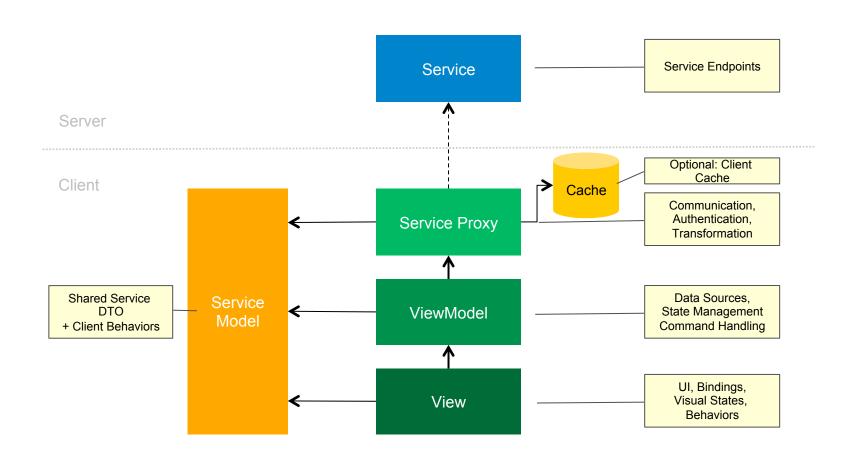


Gesamtarchitektur





Client-Architektur



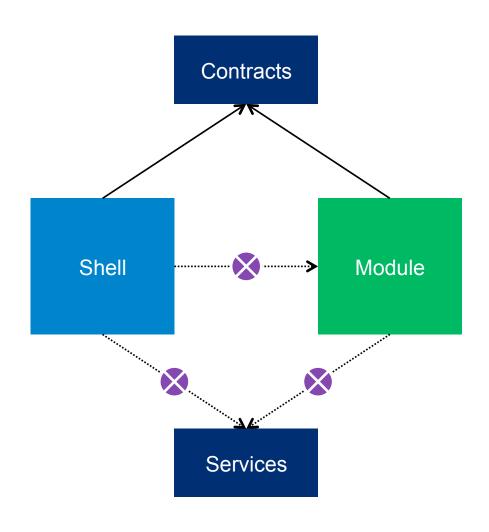


Atomare Einheiten

- Alle Teile der Anwendung verhalten sich autonom
 - Keine direkten Abhängigkeiten zwischen Shell und Modulen
- Nutzung von gemeinsamen Ressourcen
 - Bereitstellung von gemeinsamer Funktionalität in Pools
 - Zugriff über Interfaces
 - Späte Bindung zur Laufzeit
- Vorteile
 - Änderungen an einem Modul wirken sich nicht auf andere Teile aus
 - Funktionalität kann zur Testzeit ausgetauscht werden
 - Alle Teile können getrennt voneinander versioniert werden



Modularisierung





Techniken & Patterns

- Testbarkeit
 - Modul-View-ViewModel
 - Dependency Injection
- Modularisierung
 - Module Management
 - Service Pooling
 - Message Aggregation
- UI-Separation
 - Resource Management
 - State Management
 - Behaviors

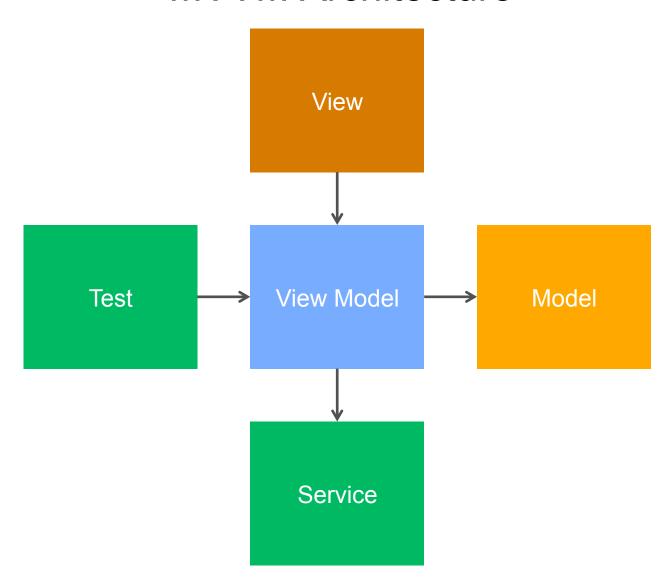


Das MVVM-Pattern

- Model-View-ViewModel
- Trennung von UI und Logik
- Ermöglicht bessere Testbarkeit
- Bessere Portierbarkeit
- Basiert auf Data Binding und Commands

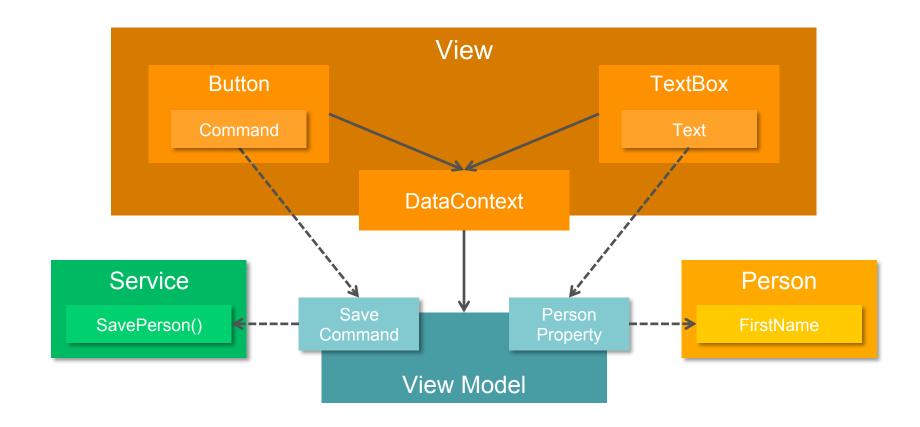


MVVM Architecture





Funktionsweise





Voraussetzungen

- Synchronisieren von UI & Model
 - INotifyPropertyChanged-Implementierung
 - Basisklasse für Model und ViewModel
- Commands kommunizieren
 - ICommand-Implementierung



DEMO

MVVM



Implementierung

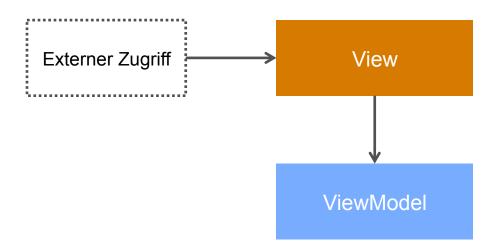
- Implizite ViewModel-Binding
 - Die View wird instanziiert
 - ViewModel wird über XAML instanziiert und gebunden
- Explizite ViewModel-Binding
 - Das ViewModel wird instanziiert
 - View wird vom ViewModel instanziiert und gebunden



Implizite ViewModel-Bindung

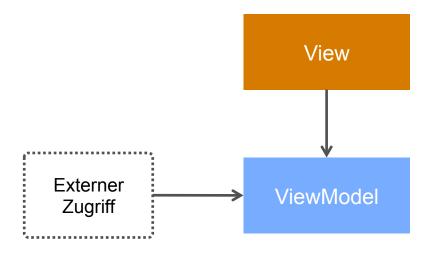
```
<Window x:Class="MyApp.Views.MainView"
   xmlns:vm="clr-namespace:MyApp.ViewModels" ... >

<Window.Resources>
   <vm:MainViewModel x:Key="viewmodel" />
   </Window.Resources>
   <frid DataContext="{StaticResource viewmodel}">
        ...
   </Grid>
   </Window>
```





Explizite ViewModel-Bindung





Anforderungen an Datenklassen

- Synchronisation
 - INotifyPropertyChanged
- Transaktionssteuerung / Undo
 - IEditableObject
- Validierung
 - IDataErrorInfo, INotifyDataErrorInfo



Anforderungen an Datenlisten

- Synchronisieren von UI & Model
 - INotifyCollectionChanged
 - ObservableCollection<T>
- Navigieren, Sortieren, Filtern
 - ICollectionView
 - ListCollectionView
- Editieren
 - IEditableCollectionView
 - ListCollectionView



DEMO

UMGANG MIT LISTEN



Zustände der Oberfläche steuern

- Zustände werden über XAML definiert
 - Zustände werden über VisualStateManager definiert
 - Ein State enthält ein Storyboard
 - Eigenschaften werden über Animationen verändert
- Umschalten von States
 - Steuerung über eine State-Eigenschaft das ViewModels
- Herausforderung
 - VisualStateManager benötigt ein UI-Element für die Umschaltung



DEMO

STATE MANAGEMENT



UI-Interaktionslogik

- Verhalten von Controls beeinflussen
 - Auswahl eines Bildes über einen Dateidialog
 - Focus-Steuerung
 - Tastatur- und Mausereignisse
 - **—** ...



UI-Interaktionslogik

- Zuweisung
 - Kann nicht im ViewModel erfolgen
 - Sollte per Markup zugewiesen werden
- Behaviors
 - Attached Properties
 - Expression Blend Behaviors



DEMO

UI-INTERACTION LOGIC



Modularisierung

- Modularisierung
 - Anwendung besteht aus einer generischen Shell
 - Fachliche Module werden als atomare Einheiten entwickelt
 - Dynamisches Nachladen der Module und Services
- Das Managed Extensibility Framework
 - Teil von .NET 4.0/4.5 & Silverlight 4/5
 - Basiert auf Attributen
 - Kann flexibel erweitert werden.



MEF

- Das Managed Extensibility Framework
 - Teil von .NET 4.0/4.5 & Silverlight 4/5
 - System.ComponentModel.Composition
 - Dynamisches Laden und Instaziieren
 - Basiert auf Attributen
 - Kann flexibel erweitert werden



Begriffsdefinition

- Export
 - Exportierte Typen (Plug-Ins)
- Imports
 - Verweis auf exportierte Typen
- Parts
 - Imports und Exports
- Composition
 - Verbinden von Parts



Imports und Exports

Exports werden geladen, instanziiert und mit Import verbunden

```
[Export]
public class MyModule
{
    ...
}
```

```
public class Program
{
    [Import]
    public MyModule Module { get; set; }
}
```



Interface-basierte Exports

Interface definieren

```
public interface IModule
{
  void Initialize();
  FrameworkElement GetView();
}
```

Exports definieren

```
[Export(typeof(IModule))]
public class Module1 : IModule
{
    ...
}
```

```
[Export(typeof(IModule))]
public class Module2 : IModule
{
    ...
}
```

Import definieren

```
public class Program
{
   [ImportMany]
   public IEnumerable<IModule> Modules { get; set; }
}
```



Composition

- Das Zusammenführen von Import und Exports zur Laufzeit
- Wird von CompositionContainer vollzogen
 - Option 1: Import/Export-Attribute
 - Option 2: Programmatische Composition



Catalogs

- Zum Auffinden von Parts werden Catalogs verwendet
 - Ermitteln alle Exports in einem bestimmten bereich
 - Stellen Metadaten f
 ür die Instanziierung bereit
- Standardkataloge
 - AssemblyCatalog
 - DirectoryCatalog
 - TypeCatalog
 - AggregatingCatalog



DEMO

MEF



Lazy Loading

- Instanz wird erst erzeugt wenn ein Zugriff erfolgt
- Verwendung der Klasse Lazy<T>

```
public class Program
{
   [ImportMany]
   public IEnumerable<Lazy<IModule>> Modules { get; set; }
}
```

Zugriff:

```
IModule module = this.Modules.First().Value;
```



Part Lifetime

- Standardmäßig cached der CompositionContainer die Instanzen
 - Keine automatische Garbage Collection
 - Die Lebensdauer von Parts kann explizit gesteuert werden

```
[ImportMany(
    RequiredCreationPolicy = CreationPolicy.NonShared)]
    public IEnumerable<Lazy<IModule>> Modules { get; set; }
```

```
[Export(typeof(IModule))]
[PartCreationPolicy(CreationPolicy.NonShared)]
public class Module1
{ ... }
```



DEMO

LAZY LOADING



Metadaten

- Exports können Metadaten bereitstellen
- Definition erfolgt über das ExportMetadata-Attribut

```
[Export(typeof(IModule))]
[ExportMetadata("DisplayName", "Buchungen")]
[ExportMetadata("DisplayIndex", 0)]
public class Module1 { ... }
```

Metadaten importieren

```
[ImportMany]
public IEnumerable<Lazy<Imodule, IDictionary<string, object>> Modules { get; set; }
```

Zugriff

```
string displayName = this.Modules.First().Metadata["DisplayName"];
```



Metadaten

- Nachteile
 - Untypisierter Zugriff
 - Keine Garantie, dass Metadaten vom Export bereitgestellt werden
- Alternative: Benutzerdefiniertes Metadaten-Attribut erstellen
 - Erstellung eines Interface zur Beschreibung der Metadaten
 - Ableitung von ExportAttribute-Klasse
 - Bereitstellung der Metadaten über den Konstruktor



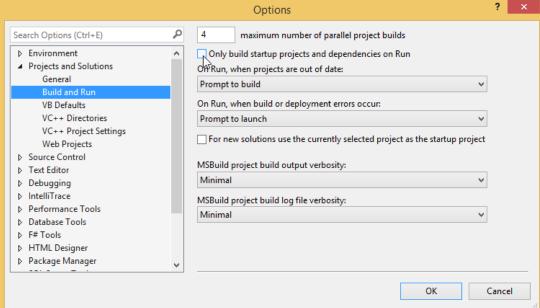
MEF METADATA



Debugging von MEF-Projekten

- Standardmäßig wird nur das Startprojekt kompiliert
 - Es kann vorkommen, dass zur Laufzeit keine Module vorhanden sind
- Visual Studio-Build-Option abschalten

Only build startup projects and dependencies On Run"



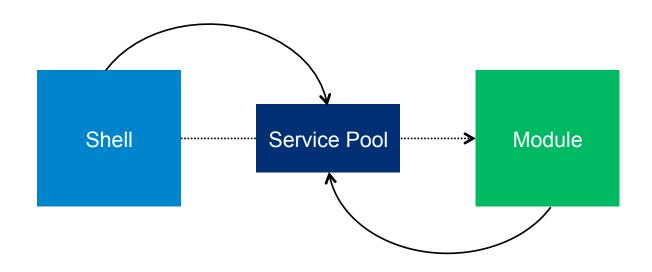


Dynamische Service-Bindung

- Shell und Module nutzen gemeinsame Services
 - Fachliche Services / Service Proxies
 - Technische Services der Shell
- Erstellung sollte über Dependency Injection erfolgen
 - Keine direkte Instanziierung
 - Abstraktion durch Interfaces
 - Nutzung von Service-Mocks zur Testzeit
 - Module können Services unterschiedlicher Versionen nutzen.
- Lösungsansatz
 - Shell ermittelt zur Laufzeit vorhandene Services und stellt diese den Modulen über einen Pool zu Verfügung



Architektur





SERVICE POOLING

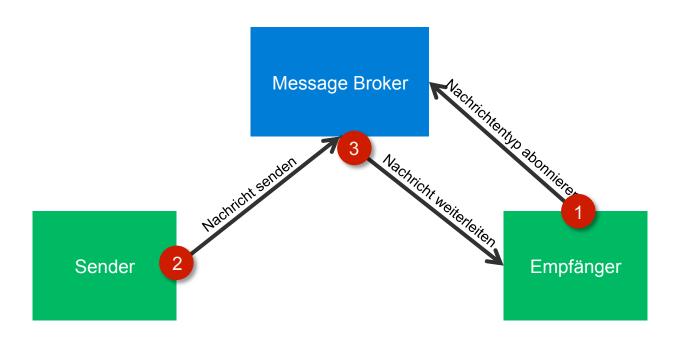


Message Aggregation

- Shell und Module müssen miteinander kommunizieren
 - z.B. Anzeige von Statusmeldungen
- Messaging nach dem Publisher-/Subscriber-Prinzip
 - Ein Teil abonniert einen Nachrichtentyp
 - Ein anderer Teil sendet eine Nachricht
 - Das Messaging-System vermitteln
 - Zur Kommunikation werden Message-Typen verwendet



Funktionsweise





MESSAGING



Umgang mit UI-Ressourcen

- UI-Ressourcen sollten zentral verwaltet werden
 - Werden von Shell und Modulen verwendet
 - Ermöglicht ein einheitliches Look & Feel
 - Erleichtert die Rollentrennung zwischen Entwickler und Designer
- Ressource-Typen
 - Farben, Fonts, Images, Größen
 - Styles & Templates
 - Custom Controls, Value Converter
- Fachliche Ressourcen werden auf Modulebene definiert
 - Data Templates, Data Template Selectors, User Controls
- Organisation
 - Definition der Assets in separaten Resource Dictionaries

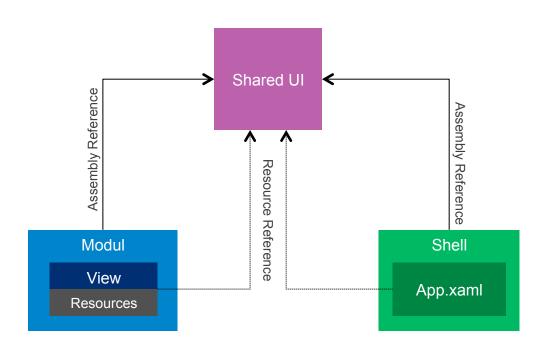


Lösungsansatz

- Ressourcen werden in separater Assembly hinterlegt
 - Shell und Module referenzieren diese Assembly
- XAML-Ressourcen müssen eingebunden werden
 - Über Merged Resource Dictionaries
- Shell
 - app.xaml
- Module
 - Resource Dictionary der View

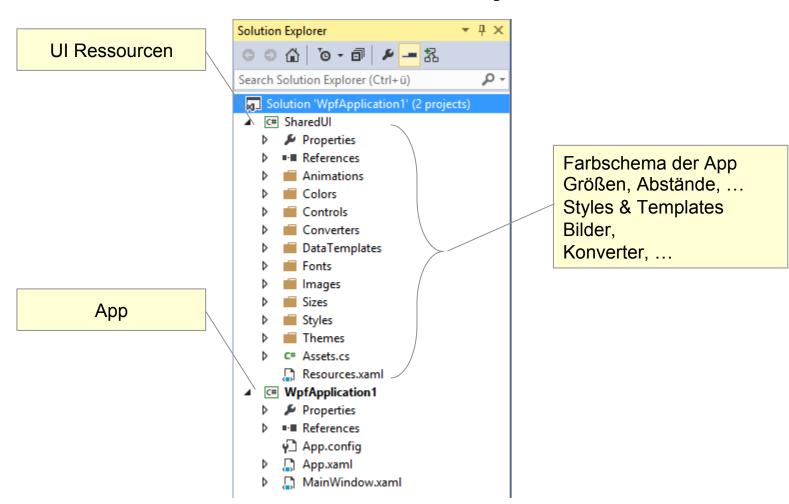


Umgang mit UI-Ressourcen





Aufbau des UI-Projekts





UI SEPARATION



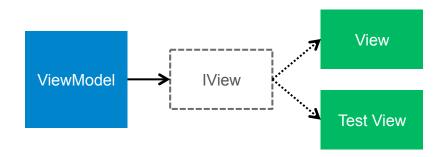
Tipps & Tricks

- Automatische View-Bindung
- Asynchrone Validierung



View-Bindung in MVVM

- Herausforderung
 - View-Implementierungen verschleiern
 - Kein UI-Zugriff in Unit Tests
 - Kein direkter Zugriff auf die View aus dem ViewModel
- Lösung
 - Factory-Klasse findet autom. die zugehörige View
 - Konventionsbasierte Ermittlung mit MEF 2.0





AUTOMATISCHE VIEW-BINDUNG



Asynchrone Validierung

- INotifyDataErrorInfo-Interface
 - Nachfolger von IDataErrorInfo
 - Verfügbar in WPF & Silverlight
- Ermittelt eine Liste von Fehlern
- Ermöglicht asynchrone Validierung



INotifyDataErrorInfo

HasErrors

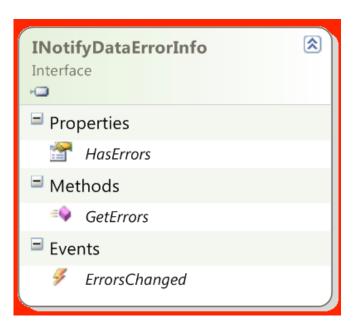
Zeigt an, ob das
 Objekt Fehler enthält

GetErrors()

 Gibt eine Liste der Fehler für eine Eigenschaft zurück

ErrorsChanged

 Wird ausgelöst, wenn sich der Fehlerstatus ändert





INotifyDataErrorInfo

- Datenklasse implementiert das Interface
- Validierung am Binding aktivieren
 - ValidatesOnNotifyDataErrors=True



EINGABEVALIDIERUNG







Sie brauchen Unterstützung?

- Training, Coaching, Reviews, Entwicklungsunterstützung
 - joerg.neumann@acando.de