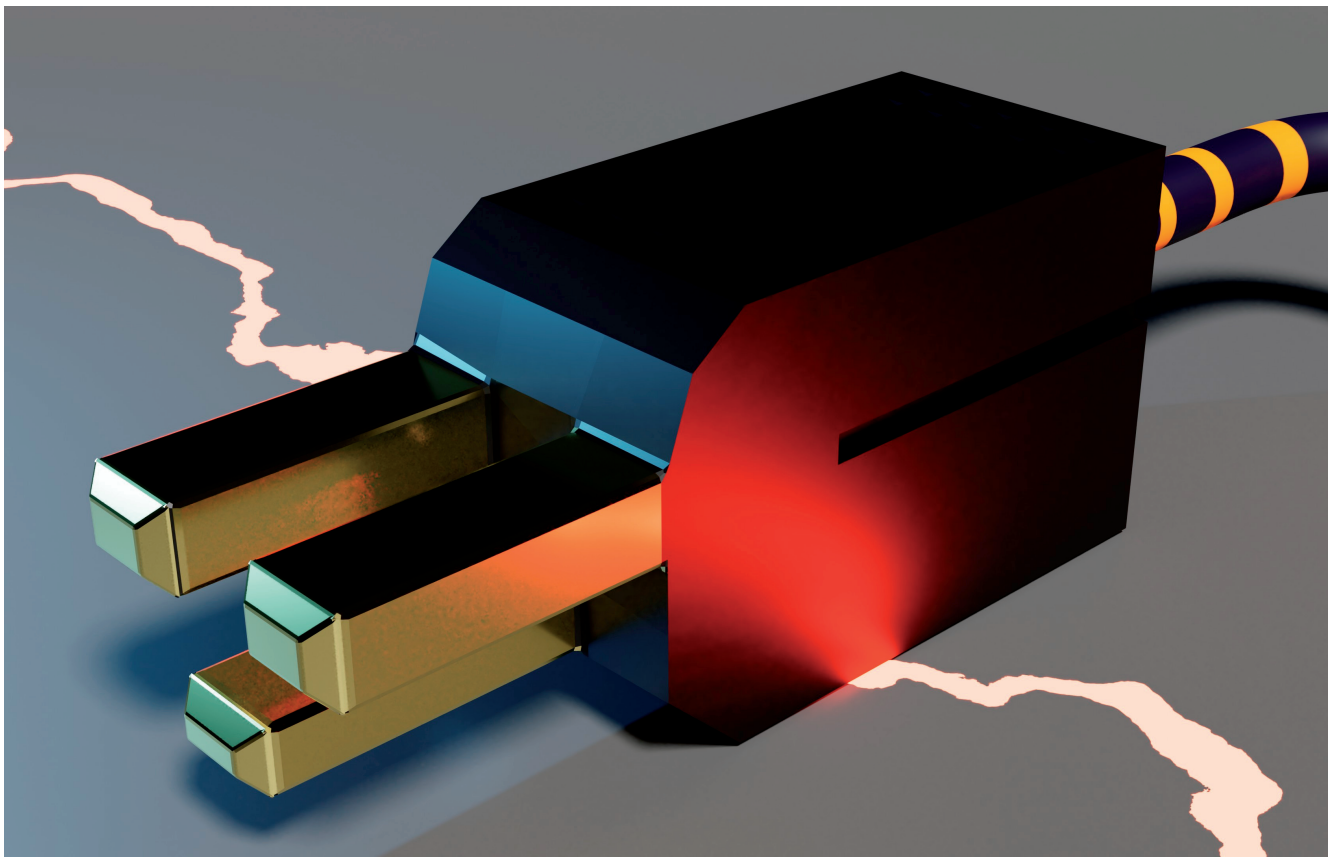


Bild: dotnetpro

DEPENDENCY INJECTION UND CO. IN WPF NUTZEN

WPF modernisiert

Seit .NET Core 3.x lassen sich viele aus Webanwendungen bekannte Vorgehensweisen auch in Desktop–Anwendungen nutzen, wenn Sie diese entsprechend einrichten.

Schon seit langer Zeit ist es in ASP.NET-Webanwendungen Stand der Technik, die Anwendung beim Start modular zusammenzusetzen. Typische Funktionen, wie sie immer wieder benötigt werden, sind zum Beispiel Logging oder der Zugriff auf Konfigurationsdaten. Diese und beliebige weitere Dienste können über Dependency Injection bereitgestellt werden, sodass sie später beispielsweise in den Controller-Klassen wieder abgerufen werden können.

Wollte man solche Möglichkeiten in der Vergangenheit auch in WPF-Anwendungen nutzen, musste man gewöhnlich auf Frameworks wie Prism zurückgreifen oder zumindest ein Dependency-Injection-Framework mit einbinden. Mit .NET Core 3.x hat Microsoft nun endlich die notwendige Funktionalität aus dem ASP-Framework extrahiert und auch für Desktop-Anwendungen verfügbar gemacht. Mit ein paar Handgriffen kann man sich die Technik in einer WPF-Anwendung zunutze machen.

Die klassische Vorlage für WPF-Anwendungen in Visual Studio 2019 basiert auch bei .NET Core 3.1 darauf, dass das in *App.xaml* über *StartupUri* festgelegte Fenster automatisch instanziert und als Hauptfenster angezeigt wird. Der erste Schritt besteht daher darin, diesen Eintrag zu entfernen. Dann lässt sich die Anwendung zwar starten, zeigt aber nichts an. Das Einrichten der Anwendung, des Hauptfensters sowie eines ViewModels erfolgt nun im C#-Code in *App.xaml.cs* (Listing 1). Vorab ist noch die Installation eines NuGet-Pakets erforderlich: Microsoft.Extensions.Hosting. Über dieses NuGet-Paket werden die Klasse *Host* sowie eine Reihe von Erweiterungsmethoden eingebunden. Hierüber werden Konfiguration, Logging et cetera initialisiert.

Der Code zum Einrichten des Hosts lässt sich auf verschiedene Weisen ausführen. Neben der hier verwendeten Überschreibung der Methode *OnStartup* kann beispielsweise auch der *Startup*-Event der *Application*-Klasse eingesetzt werden.

Prinzipiell würden sich alle Details einzeln und individuell einrichten lassen. Die Methode *Host.CreateDefaultBuilder* fasst jedoch bereits einige der gebräuchlichen Konstruktio-

nen zusammen und vereinfacht so das Vorgehen (siehe Details hierzu in [1]).

Zu den Standardfunktionen gehören zum Beispiel das Lesen von Konfigurationsdaten aus der Kommandozeile, aus Windows-Umgebungsvariablen oder der *appsettings.json*-Datei sowie die Bereitstellung von Logging-Mechanismen inklusive der Log-Ausgabe auf die Konsole beziehungsweise das Debug-Fenster. *CreateDefaultBuilder* liefert ein Objekt zurück, welches das Interface *IHostBuilder* implementiert. Auf diesem Interface können weitere Methoden aufgerufen werden, mit denen die Standardeinrichtung ergänzt werden kann. Hier wird über *ConfigureServices* der Container für die Dependency Injection gefüllt. Zunächst wird hier lediglich eine Instanz der Klasse *MainViewModel* (Listing 2) bereitgestellt. Der abschließende Aufruf der Methode *Build* generiert das fertige Host-Objekt. Gestartet wird dieses dann über *StartAsync* beziehungsweise die Erweiterungsmethode *RunAsync*, je nachdem, ob das von dem Aufruf zurückgegebene *Task*-Objekt sofort nach dem Start oder erst nach Beendigung des Hosts beendet werden soll. Im Beispiel wird nur der Start abgewartet.

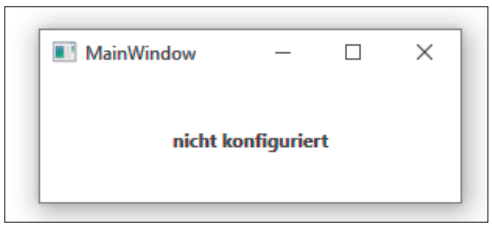
Listing 2: Lesen von Konfigurationsdaten ohne eine spezifische Quelle

```
public class MainViewModel : INotifyPropertyChanged
{
  ...
  public string Title { get; set; }

  private readonly IConfiguration configuration;

  // Bereitstellen von Services über Constructor Injection
  public MainViewModel(IConfiguration configuration)
  {
    this.configuration = configuration;

    // Laden des Titels aus Konfigurationsdaten
    this.Title = configuration["title"] ?? "nicht konfiguriert";
  }

}
```

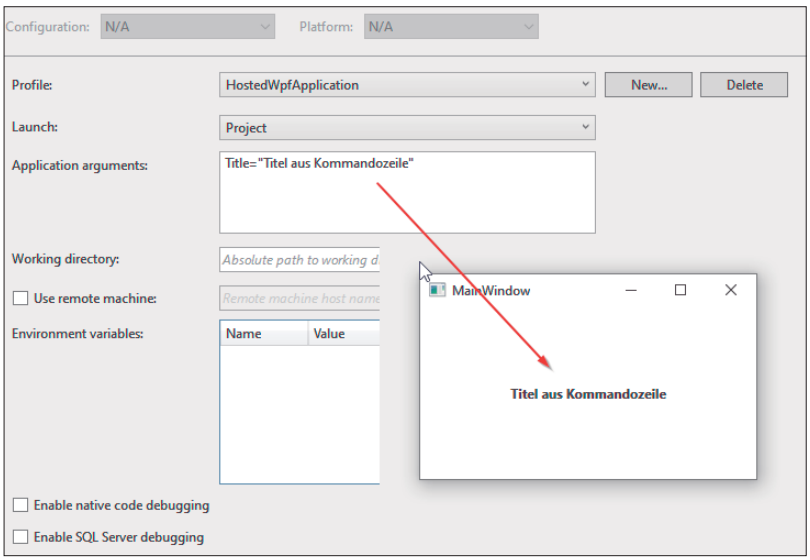
Danach wird das Hauptfenster instanziert und eingerichtet. Über *AppHost.Services.GetService<MainViewModel>* wird hierbei die Instanz des zuvor im Container eingerichteten ViewModels abgerufen. Warum ist der Umweg sinnvoll ▶

Listing 1: .NET-Core-Dependency-Injection in die WPF-Anwendung einbinden

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.DependencyInjection;
using System.Windows;

namespace HostedWpfApplication
{
  /// <summary>
  /// Interaction logic for App.xaml
  /// </summary>
  public partial class App : Application
  {
    public static IHost AppHost;

    protected override async void
        OnStartup(StartupEventArgs e)
    {
      // Host konfigurieren
      AppHost = Host.CreateDefaultBuilder(e.Args)
       .ConfigureServices(ConfigureServices)
       .Build();

      // Host starten
      await AppHost.StartAsync();

      // Hauptfenster einrichten
      var mainWindow = new MainWindow();

      // ViewModel aus DI-Container laden
      mainWindow.DataContext =
        AppHost.Services.GetService<MainViewModel>();
      mainWindow.Show();
    }

    private void ConfigureServices(HostBuilderContext
        context, IServiceCollection services)
    {
      // Hier werden die Services für Dependency
      // Injection bereitgestellt
      services.AddSingleton<MainViewModel>();
    }

    protected override async void OnExit(ExitEventArgs e)
    {
      // Beim Beenden Host stoppen und alles aufräumen
      using (AppHost) await AppHost.StopAsync();
    }
  }
}
```



Der erste Start der gehosteten WPF-Anwendung (Bild 1)



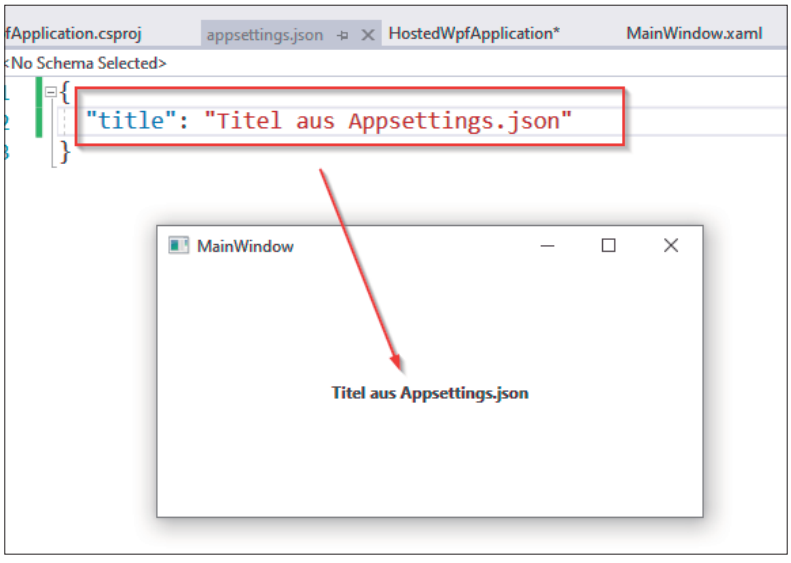
Konfiguration über Kommandozeilenparameter (Bild 2)

und warum instanziert man stattdessen die *ViewModel*-Klasse nicht direkt an dieser Stelle? Nun, der Schlüssel dazu liegt in der Möglichkeit, innerhalb der *ViewModel*-Klasse auf weitere per Dependency Injection bereitgestellte Objekte zugreifen zu können. Die Technik, die hierbei üblicherweise zum Einsatz kommt, nennt sich Constructor Injection. Der Konstruktor der Klasse kann Parameter enthalten, die bei einer Instanzierung über Dependency Injection dann automatisch gefüllt werden. Im Beispiel in Listing 2 ist dies der Parameter vom Typ *IConfiguration*. Dieser Typ wurde zuvor über *CreateDefaultBuilder* hinzugefügt und ermöglicht den Zugriff auf Konfigurationsdaten.

Ein an die Eigenschaft *Title* gebundener TextBlock zeigt die unter dem Key *title* vorgegebene Konfigurationsinformation an. Zunächst einmal gibt es diese jedoch nicht, und die Anzeige beim Start der Anwendung ergibt sich wie in Bild 1 gezeigt. Doch die Konfiguration lässt sich leicht einrichten. Als erste Möglichkeit können Sie die Daten an die Kommandozeile übergeben (Bild 2). Groß- und Kleinschreibung spielen beim Schlüssel keine Rolle. Auch gibt es verschiedene Syntaxvarianten bei der Übergabe über die Kommandozeile (siehe [2]).

Alternativ kann die Konfiguration zum Beispiel auch in Form einer JSON-Datei bereitgestellt werden. *CreateDefaultBuilder* richtet hier automatisch den Zugriff auf eine Datei mit dem Namen *appsettings.json* ein. Ebenfalls berücksichtigt werden spezifische Namen wie *appsettings.Development.json*, die je nach eingestellten Werten von Umgebungsvariablen des Projekts zum Tragen kommen. Weitere Quellen können bei Bedarf explizit hinzugefügt werden. Ein Beispiel für die Konfiguration über *appsettings.json* zeigt Bild 3.

Beim Beenden der Anwendung muss der Host wieder gestoppt werden. Auch sollte zur Sicherheit dessen *Dispose*-Methode aufgerufen werden, damit auch für alle verbleibenden Service-Instanzen *Dispose* aufgerufen wird. Das erfolgt wie in Listing 1 zu sehen innerhalb der überschriebenen *OnExit*-Methode. Diese wird aufgerufen, nachdem das Hauptfenster geschlossen wurde.



Konfiguration über die Datei appsettings.json (Bild 3)

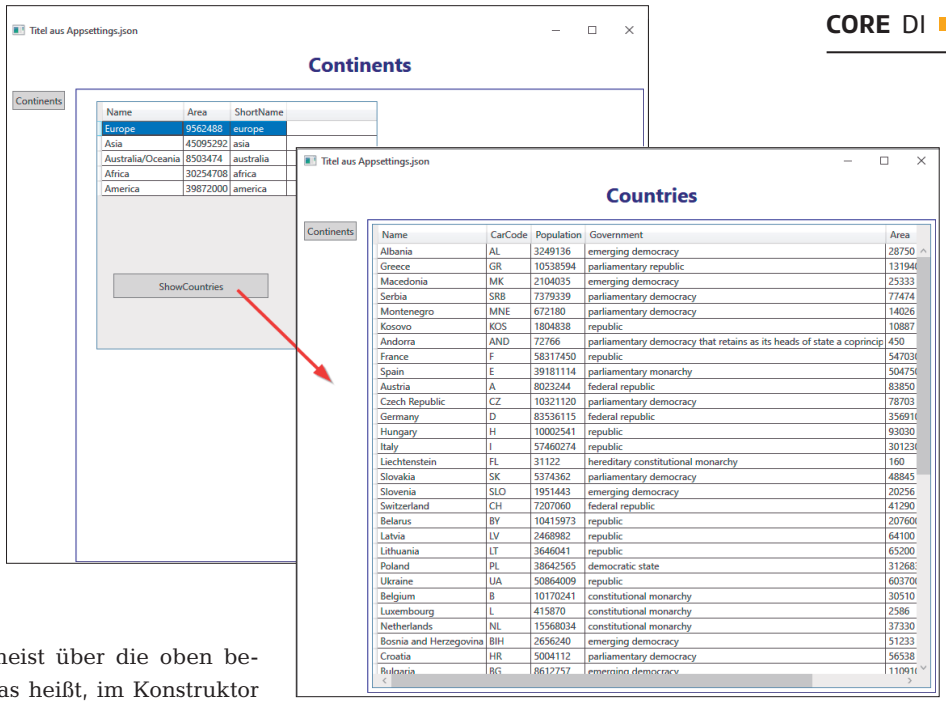
Dependency Injection (DI)

Für das Einrichten des DI-Containers stehen eine Reihe von Funktionen zur Verfügung. Im Wesentlichen sind dies hier *AddSingleton* sowie *AddTransient* in verschiedenen Überladungen.

AddSingleton sorgt dafür, dass bei späteren Abrufen immer wieder die Referenz auf ein und dieselbe Instanz zurückgegeben wird; *AddTransient* hingegen bewirkt, dass bei jedem Abruf eine neue Instanz angelegt wird.

Implementiert ein bereitgestellter Datentyp (auch *Service* genannt) das Interface *IDisposable*, dann wird der Container später auch die *Dispose*-Methode aufrufen. So können vom Service benutzte Ressourcen sicher wieder freigegeben werden.

Dank der verschiedenen Überladungen von *AddSingleton* beziehungsweise *AddTransient* können je nach Bedarf entweder einfach die Datentypen angegeben werden, oder man gibt eine Factory-Methode mit, die die Instanzierung selbst übernimmt.



Das Umschalten der Views und die Übergabe der Daten erfolgen via Dependency Injection (Bild 4)

Aus ASP.NET-Anwendungen ist noch eine weitere Variante bekannt, nämlich *AddScoped*. Diese wird gerne eingesetzt, um während der gesamten Abarbeitung einer Webanfrage immer wieder dasselbe Objekt vom DI-Container zu erhalten, also ähnlich wie beim Singleton. Nach Abschluss der Abfrage wird dieses dann verworfen (und *Dispose* aufgerufen). Diese Variante steht für Desktop-Anwendungen genauso zur Verfügung, wird aber vermutlich eher selten zum Einsatz kommen.

Der Abruf der Services erfolgt meist über die oben beschriebene Constructor Injection, das heißt, im Konstruktor werden Parameter des gewünschten Typs definiert. Die Objektreferenzen werden dann in der Regel in einem privaten Feld für den späteren Zugriff gespeichert.

Alternativ kann man aber auch über den Container die Objekte imperativ abrufen. Ein Beispiel dazu haben Sie bereits in Listing 1 gesehen, nämlich den Aufruf von

```
AppHost.Services.GetService<MainViewModel>()
```

Im Grunde verbirgt sich hinter dem Dependency-Injection-Container nur eine Key-Value-Liste. Hierbei ist der Key ein Typ beziehungsweise ein Typ-Objekt, zum Beispiel *typeof(MainViewModel)*. Oft wird als Typ hier ein Interface verwendet, sodass der Aufrufer nicht von der Implementierung des spezifischen Objekttyps abhängig ist.

In diesem Zusammenhang kommt es auch vor, dass für einen Interfacetyp mehrere Aufrufe von beispielsweise *AddSingleton* implementiert werden. In solchen Fällen bietet sich der Aufruf von *GetServices* an, der eine Auflistung der registrierten Objekte zurückgibt.

Logging

Auch das Logging ist durch den obigen Aufruf von *CreateDefaultBuilder* bereits vorbereitet. Über Dependency Injection lässt sich ein Logger abrufen. Hierbei wird meist der Typ eingesetzt, innerhalb dessen die Log-Ausgaben erfolgen sollen. Ein Beispiel dafür zeigt Listing 3. Verschiedene Methoden erlauben hierüber Ausgaben zu unterschiedlichen Log-Leveln (etwa *Debug*, *Warning*, *Error* …). Die Voreinstellungen, ab welchem Level die Ausgaben tatsächlich erfolgen sollen, können wiederum in der Konfiguration festgelegt werden. Listing 4 zeigt ein Beispiel für die angepasste *appsettings.json*.

In dieser Konstellation kann es besonders hilfreich sein, für unterschiedliche Umgebungen (Entwicklung, Produktion) auch unterschiedliche Konfigurationsdateien anzulegen, sodass beispielsweise *Debug*-Ausgaben während der Entwicklungsphase im Log landen, im Produktionsbetrieb aber nur noch Fehlermeldungen.

MVVM-Pattern (Model-View-ViewModel)

Das MVVM-Pattern ist bei der WPF-Programmierung weit verbreitet im Einsatz und wurde schon oft in der dotnetpro beschrieben. Es lässt sich hervorragend mit den oben beschriebenen Möglichkeiten kombinieren. An manchen Stellen sind allerdings ein paar Besonderheiten zu beachten, sodass sie hier an einem Beispiel gezeigt werden sollen.

In der Beispielanwendung dient das Hauptfenster als Container für beliebige View-ViewModel-Paare. Eine Button-Leiste ermöglicht über Commands das Instanzieren be- ▶

Listing 3: Log-Ausgaben-Injection implementieren

```
public MainViewModel(IConfiguration configuration,
    ILogger<MainViewModel>logger)
{
  ...
  logger.LogDebug("Konstruktor MainViewModel");
}
```

Listing 4: Beispiel für appsettings.json

```
{
  "title": "Titel aus Appsettings.json",

  "Logging": {
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

Listing 5: Imperativer Zugriff auf DI-Services in einer Basisklasse für ViewModels

```
public class ViewModelBase : NotificationObject
{
  public IConfiguration Configuration { get; }
  public ILogger Logger { get; }

  public string Title { get; set; }

  public ViewModelBase()
  {
    var services = App.AppHost.Services;
    Configuration = services.
      GetService<IConfiguration>();

    Type tlogger = typeof(ILogger<>);
    Type t = tlogger.MakeGenericType(this.GetType());
    Logger = (ILogger)services.GetService(t);
  }
}
```

stimmter ViewModels, die dann an die *Content*-Eigenschaft eines als Platzhalter dienenden ContentControls gebunden werden. Damit an dieser Stelle nicht *ToString* des ViewModels angezeigt wird, sondern die dazugehörige View, werden in einem Resource Dictionary entsprechende Data Templates eingerichtet. Aus Platzgründen kann der vollständige Code hier nicht wiedergegeben werden. Sie finden ihn aber in den Downloads zu diesem Artikel.

Eine kleine, zusätzlich eingebundene Bibliothek stellt Geografiedaten aus einer XML-Struktur bereit und liefert so eine Liste der Kontinente sowie zu jedem Kontinent eine Liste der Länder. Zur Laufzeit ergibt sich dann eine Anwendung wie in Bild 4 zu sehen.

Die bereits beschriebene Constructor Injection zeigt dann ihre Grenzen, wenn es um Vererbungen geht. Denn nur die konkrete Klasse, die instanziert wird, kann über den Konstruktor Services entgegennehmen. Konstruktoren der Basisklasse müssen von dieser ja entsprechend aufgerufen und deren Parameter explizit übergeben werden. In solchen Fällen muss man entweder in der konkreten Klasse alle benötigten Parameter formulieren und an die Basisklasse durchreichen oder wieder zu den oben genannten Methoden wie *GetService* greifen.

Im Fall des Loggers, bei dem ein generischer Typ über Dependency Injection angefordert wird, ist noch ein Zwischenschritt notwendig, wie in Listing 5 zu sehen. Das Listing zeigt eine Basisklasse für ViewModels. Hier wird erst mittels Reflection das benötigte Typ-Objekt für den Logger geholt und dieses anschließend an *GetService* übergeben. So lässt sich das Äquivalent zu *ILogger<T>* formell generieren.

Ein ViewModel soll später die Möglichkeit erhalten, den Inhaltsbereich des Hauptfensters auszutauschen (zum Beispiel soll die View der Kontinente ersetzt werden durch die der Länder). Dazu muss das ViewModel natürlich eine konkrete Methode, in diesem Fall des MainViewModels, aufrufen. Meist möchte man allerdings dem unabhängigen View-

Listing 6: Implementierungsdetails mithilfe von Interfaces verbergen

```
public interface IMainController
{
  void DisplayViewModel(ViewModelBase viewModel);
  void DisplayViewModel<T>() where T : ViewModelBase;
}

public class MainViewModel : NotificationObject,
    IMainController
{

  ...

  public MainViewModel(IConfiguration configuration,
    ILogger<MainViewModel>logger, IServiceProvider
    services)
  { ... }

  private void ShowContinents()
  {
    DisplayViewModel<ContinentsViewModel>();
  }

  // IMainController-Implementierung
  public void DisplayViewModel(ViewModelBase viewModel)
  {
    SelectedViewModel = viewModel;
  }

  public void DisplayViewModel<T>() where T :
      ViewModelBase
  {
    var t = typeof(T);
    var vm = (T)services.GetService(t);
    DisplayViewModel(vm);
  }
}
```

Listing 7: Weitere Services wurden in der Application-Klasse hinzugefügt

```
private void ConfigureServices(HostBuilderContext
    context, IServiceCollection services)
{
  // Hier werden die Services für Dependency
  // Injection bereitgestellt

  // Instanz des MainViewModels registrieren
  services.AddSingleton<MainViewModel>();

  // Registrieren derselben Instanz für das Interface
  // IMainController
  services.AddSingleton<IMainController>(p =>
    p.GetRequiredService<MainViewModel>());

  // Instanz für World-Bibliothek einrichten
  string path = context.Configuration["MondialPath"];
  services.AddSingleton<World>(new World(path));

  // ViewModels
  services.AddSingleton<ContinentsViewModel>();
  services.AddSingleton<CountriesViewModel>();

  // Hilfsobjekt für Datenaustausch
  services.AddSingleton<WorldState>();
}
```

Model keine Implementierungsdetails des MainViewModels aufzwingen und daher typischerweise die betreffenden Methoden über ein Interface entkoppeln. Diesen Ansatz sehen Sie in Listing 6.

Im Konstruktor des betreffenden ViewModels (hier *ContinentsViewModel*) wird dann ein Parameter vom Typ des Interfaces (hier *IMainController*) angelegt. Das führt jedoch zu einer neuen Herausforderung: Jetzt muss nämlich ein und dasselbe Objekt, das über *AddSingleton* dem Dependency-Injection-Container hinzugefügt wird, zweimal registriert werden. Nämlich einmal über den eigentlichen Typ und dann auch noch über das Interface.

Die Lösung hierzu sehen Sie in Listing 7. Sie ist einem Vorschlag aus [3] entnommen worden. Konkret kommt hier eine Factory-Methode zum Einsatz, in der über *IServiceProvider.GetRequiredService* die benötigte Instanz zur Laufzeit später abgerufen werden kann.

Darüber hinaus sehen Sie im Listing das Bereitstellen der *World*-Bibliothek-Instanz (der Dateipfad kommt wieder aus der JSON-Konfigurationsdatei), der beiden ViewModels sowie eines Hilfsobjekts (*WorldState*), das weiter unten noch genauer beschrieben wird.

An dieser Stelle müssen wir noch einmal auf die Implementierung der Methode *DisplayViewModel<T>* aus Listing 6 zurückkommen. Diese muss ja auch auf Basis eines vorgegebenen Typs das betreffende ViewModel aus dem DI-Container abrufen. Statt über das Host-Objekt auf den ServiceProvider zuzugreifen, wird dieser hier ebenfalls über Constructor Injection übernommen (*IServiceProvider*-Parameter). In der Methode selbst wird dann darüber das ViewModel-Objekt abgerufen.

Das ViewModel für die Kontinente nimmt nun seinerseits die Geschäftslogik (*World*), *IMainController* und so weiter im Konstruktor entgegen (Listing 8). Über eine Command-Methode ruft es die oben beschriebene Interface-Methode *DisplayViewModel* auf und sorgt so dafür, dass das ViewModel für die Darstellung der Countries beziehungsweise dessen zugehörige View zur Anzeige gebracht wird.

Doch wie können dem anderen ViewModel jetzt die Informationen mitgegeben werden, die dieses benötigt? *CountriesViewModel* sollte zumindest erfahren, um welchen Kontinent es sich handelt. Leider ist an dieser Stelle keine Instanz greifbar, über die sich die entsprechenden Eigenschaften setzen ließen. ▶

Listing 8: Geschäftslogik und Objekte für den Datenaustausch per DI einfügen

```
public class ContinentsViewModel : ViewModelBase
{
  ...
  public ContinentsViewModel(World world,
      IMainController mainController,
      WorldState worldState,
      ILogger<ContinentsViewModel> logger)
  {
    this.Continents = world.GetContinents();
    ...
    ShowCountriesCommand =
      new ActionCommand(ShowCountries);
  }

  private void ShowCountries()
  {
    mainController.DisplayViewModel<
     CountriesViewModel>();
  }
}
```

Listing 9: Service-Klasse für den Datenaustausch zwischen ContinentsView und CountriesView

```
public class WorldState : NotificationObject
{
  private Continent selectedContinent;

  public Continent SelectedContinent
  {
    get { return selectedContinent; }
    set { selectedContinent =
      value; OnPropertyChanged(); }
  }
}
```

Listing 10: Zugriff auf die ausgetauschten Informationen im CountriesViewModel

```
public class CountriesViewModel : ViewModelBase
{
  ...
  public CountriesViewModel(World world,
      WorldState worldState)
  {
    ...
    GetCountries();

    // Falls beide Views parallel angezeigt werden,
    // Änderungen verfolgen
    this.worldState.PropertyChanged +=
      WorldState_PropertyChanged;
  }

  private void WorldState_PropertyChanged(
      object sender,
      System.ComponentModel.PropertyChangedEventArgs e)
  {
    GetCountries();
  }

  private void GetCountries()
  {
    Countries = world.GetCountriesOnContinent(
      worldState.SelectedContinent.ShortName);
  }
}
```

Zur Lösung wird ein gängiges Pattern herangezogen, nämlich, ein Hilfsobjekt über Dependency Injection einfach durchzureichen. Hier kommt nun die bereits genannte Hilfsklasse *WorldState* zum Einsatz (Listing 9). Sie ist wiederum als Singleton angelegt, wird vom *ContinentsViewModel* eingebunden und dort initialisiert, und zwar über eine Datenbindung mit der Information zum ausgewählten Kontinent. Im *CountriesViewModel* (Listing 10) wird dieses Objekt ebenfalls eingebunden und die übergebene Information für die Darstellung verwendet. Auf diese Weise lassen sich beliebig komplexe Informationen zwischen zwei ViewModels über Dependency Injection austauschen, ohne dass die ViewModel-Instanzen voneinander Kenntnis haben müssen.

Fazit

Dependency Injection gewinnt in der Softwareentwicklung zunehmend an Bedeutung. Es ermöglicht eine modulare Programmierung, vermeidet Abhängigkeiten und schafft so nebenbei auch die Voraussetzungen für den Einsatz automatischer Tests.

Mit ein wenig zusätzlicher Infrastruktur lassen sich, wie am Beispiel gezeigt, die in .NET Core vorhandenen Implementierungen gewinnbringend nutzen. Abhängigkeiten von externen Tools und Frameworks wie Prism, Unity Application Block oder Autofac können vermieden werden. Das Lesen von Konfigurationsdaten aus unterschiedlichen Quellen sowie das Ausgeben von Log-Informationen sind bereits implementiert und für Erweiterungen vorbereitet.

Für moderne WPF-Anwendungen, ganz gleich ob sie gerade neu aufgesetzt oder aus dem klassischen .NET-Framework nach .NET Core portiert werden, spricht nichts dagegen, die gezeigten Techniken einzusetzen. ■

[1] CreateDefaultBuilder, www.dotnetpro.de/SL2010DI1
[2] Konfiguration über die Kommandozeile, www.dotnetpro.de/SL2010DI2
[3] How to register a service with multiple interfaces in ASP.NET Core DI, www.dotnetpro.de/SL2010DI3
[4] Quellcode zum Artikel, www.dotnetpro.de/SL2010DI4

Dr. Joachim Fuchs
ist begeisterter Anhänger von Microsofts .NET-Philosophie. Er arbeitet als Softwarearchitekt, Berater und Dozent im Expertennetzwerk www.it-visions.de. Seine Schwerpunkte liegen derzeit bei XAML- und Web-UI-Technologien.
dnp@fuechse-online.de

dnpCode A2010DI