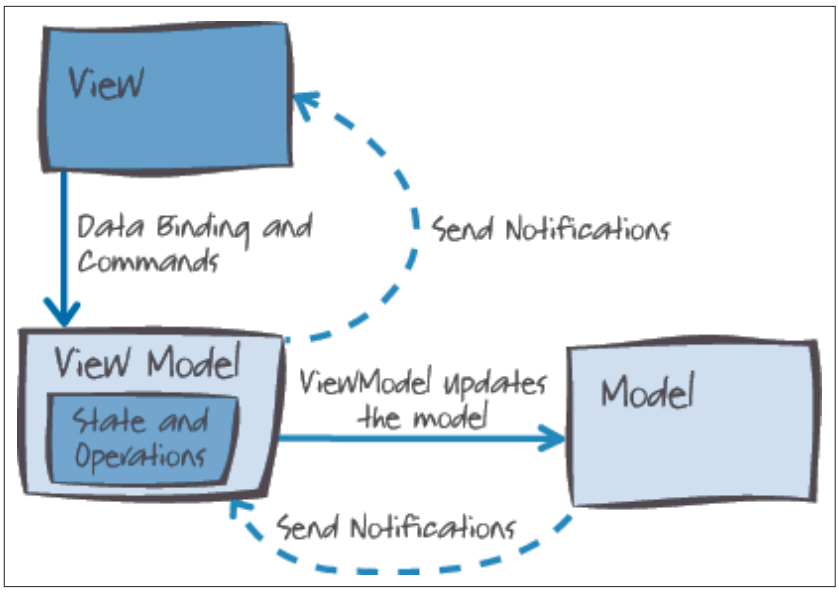
MVVM LIGHT TOOLKIT VERSUS CROSSLIGHT

Musterhafte Hilfe

Frameworks zum Trennen von Daten, Oberfläche und Geschäftslogik einer Anwendung.

Das Entwurfsmuster Model-View-ViewModel, kurz MVVM, hat sich für die Trennung von Daten, Oberflächen und Geschäftslogik in der Entwicklung inzwischen fest etabliert. Es erfreut sich nicht nur in der .NET-Welt einer großen Beliebtheit, sondern inzwischen auch bei vielen JavaScript-Frameworks für Single-Page-Anwendungen, wie zum Beispiel Knockout.js. Selbst für Android und Angular stehen Implementierungen für das MVVM-Pattern zur Verfügung.

Besondere Bedeutung kommt dem Muster beim Entwickeln von systemübergreifenden mobilen Anwendungen zu. Hier stellt sich die Herausforderung, die spezifischen Möglichkeiten der Geräte zu vereinheitlichen und unter einer Programmieroberfläche zu harmonisieren.



Quelle: Microsoft

Die Abhängigkeiten im Entwurfsmuster MVVM (Bild 1)

Warum fällt die Wahl hierbei ausgerechnet auf MVVM? Mit der Implementierung vor allem durch Xamarin für iOS und Android lässt sich der Code deutlich vereinheitlichen und daher auch kompakter gestalten. Hierdurch lassen sich dann Abhängigkeiten zwischen Benutzeroberfläche und Geschäftslogik verringern. MVVM erlaubt eine sehr gute Trennung von User-Interface-Design und User-Interface-Logik. Mithilfe eines MVVM-Frameworks kann ein Entwickler die typischen Fälle einer MVVM-Applikation wesentlich einfacher und effizienter umsetzen als ohne ein solches Framework.

Das MVVM-Pattern

Das Entwurfsmuster Model-View-ViewModel reiht sich in die bekannten Architekturmuster MVC (Model-View-Controller), MVP (Model-View-Presenter) und Presentation Model ein. Da die meisten Anwendungsentwicklungen in der Regel aus Datenbankmodell, Benutzeroberfläche und Geschäftslogik bestehen, ist eine besser strukturierte Kommunikation zwischen Modell und Oberfläche das Ziel. Die bekanntesten Muster hierfür sind, wie schon gesagt, MVC und MVP. Auf diese Weise sollen die Abhängigkeiten zwischen Oberfläche und Geschäftslogik verringert und eine organisierte und modulare Schicht für die Anwendungsentwicklung erstellt werden, die als Brücke zwischen Benutzeroberfläche und Logik fungiert.

Als neuer Lösungsansatz kommt dann noch MVVM hinzu. Das Muster erlaubt eine noch bessere Trennung von UI-Design und der benötigten UI-Logik.

Mit MVVM erfolgt eine lose Kopplung zwischen View und View-Modell. Diese Vorteile gibt es aber nicht umsonst, auch MVVM setzt eine gewisse Infrastruktur voraus, um funktionieren zu können. Das Muster besteht aus drei Hauptkomponenten:

- dem Datenmodell (Model),
- dem UI-Design, oder auch Sicht (View),
- der UI-Logik, oder auch View-Modell (ViewModel).

Bild 1 zeigt die Abhängigkeiten des MVVM-Musters. Die Sicht (View) nutzt das ViewModel und die ViewModel-Schicht das Model. Umgekehrt bestehen keine Abhängigkeiten.

Das heißt, die übergeordnete Komponente kennt nur die untergeordnete. Somit kennt das Datenmodell weder die UI-Logik noch das UI. Die Sicht ist demzufolge für die Darstellung der Informationen zuständig und enthält die entsprechenden grafischen Elemente. Jedem UI wird eine ViewModel-Klasse zugeordnet, welche die darzustellenden Daten und die Logik enthält. Somit werden praktisch alle Benutzereingaben mittels Datenbindung an die UI-Logik weitergeleitet und dort entsprechend weiterverarbeitet.

Das Datenmodell fasst die Klassen zusammen, die sich um das Laden und Speichern der Daten kümmern. Es enthält also auch die Geschäftslogik. Somit ergibt sich über das MVVM-Muster eine strikte Trennung von Oberfläche und UI-Logik. Die grundsätzliche Struktur einer MVVM-Applikation sollte immer folgende Ansätze besitzen:

- Pro View wird ein Interface angelegt, in dem die Funktion definiert wird, die das View-Modell benötigt, um die View steuern zu können.
- Das View-Modell erhält zur Laufzeit ein Objekt, das dieses Interface implementiert.
- Jede View erhält zur Laufzeit eine Instanz seines View-Modells, um so auf die Daten zugreifen zu können.

Mit dieser Struktur lassen sich dann sehr einfach die Interaktionen im MVVM-Muster umsetzen. Der wichtigste Aspekt, der MVVM zu einem sehr guten Muster macht, ist die Infrastruktur der Datenbindung. Durch das Binden von Eigenschaften einer View an ein ViewModel entsteht eine lose Kopplung zwischen den beiden, wodurch in einem ViewModel kein Code geschrieben werden muss, der eine View direkt aktualisiert [1].

Frameworks

Inzwischen gibt es eine Vielzahl von MVVM-Frameworks, wie zum Bespiel CoreMVVM [2], Caliburn [3] oder auch Prism [4]. Aber warum sollte man für dieses Architektur-Muster ein Framework einsetzen?

Bei einem entsprechenden Framework handelt es sich um einen standardisierten Satz von Konzepten, Praktiken und Kriterien für den Umgang mit MVVM, und dieser soll helfen, Probleme und Implementierungen schneller und effektiver zu lösen.

Alle traditionellen MVVM-Frameworks arbeiten sehr stabil, bringen aber leider auch hin und wieder einen gewissen Overhead an Funktionalität und Umfang mit sich. Das sollte aber nicht abschrecken, da generell die Vorteile beim Einsatz eines Frameworks überwiegen.

Somit ergeben sich für den Einsatz eines MVVM-Frameworks folgende Vorteile:

- Standardkonzepte erleichtern Umsetzung und Implementierung,
- schnellere Umsetzung des Entwicklungsprozesses,
- Einsatz von bewährten Praktiken (Best Practices),
- Hilfe in Foren und Communities,
- umfassende Wiederverwendbarkeit der Komponenten,
- regelmäßige Updates.

Die Frage, welches Framework denn nun das beste für die Entwicklung mit dem MVVM-Muster ist, lässt sich dabei nicht so einfach beantworten. Zu diesem Zweck stellt dieser Artikel zwei sehr unterschiedliche MVVM-Frameworks vor. Zum einen das MVVM Light Toolkit, das auf Open-Source-Software basiert, zum anderen Crosslight, entwickelt von Intersoft Solutions, das für den Einsatz von kommerzieller Software steht und ein komplettes, aber auch pragmatisches Framework zur Verfügung stellt.

MVVM Light Toolkit

Das MVVM Light Toolkit ist ein kleines, leichtgewichtiges, quelloffenes und plattformübergreifendes MVVM-Framework von Laurent Bugnion (GalaSoft). Sein Hauptzweck besteht darin, die Entwicklung von MVVM-Anwendungen mit der Windows Presentation Foundation, der Universal Windows Platform (UWP) sowie mit Xamarin.Android und Xamarin.iOS zu beschleunigen.

Mithilfe des Frameworks lässt sich das MVVM-Muster sehr komfortabel implementieren und somit auch für Android und iOS verfügbar machen. Ein weiterer wichtiger Aspekt des Frameworks ist die Anpassbarkeit und Erweiterbarkeit, die sich dadurch ergibt, dass man auch nur die MVVM-Light-Bibliotheken einbinden muss beziehungsweise benötigt. Das ergibt ein sehr schlankes Framework für die Implementierung von MVVM [5].

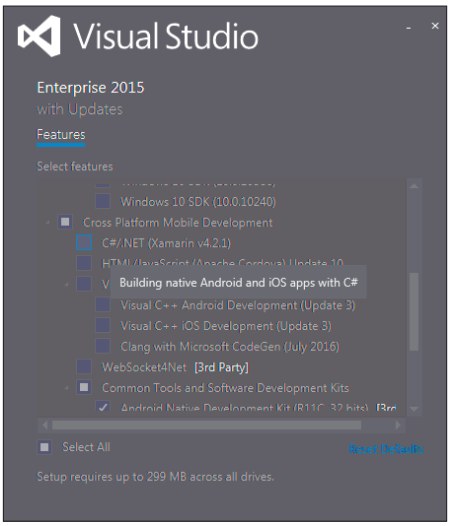
Crosslight

Crosslight ist ein leistungsfähiges Entwickler-Toolset von Intersoft Solutions. Das Unternehmen ist offizieller Partner von Xamarin und baut sein Framework auf Xamarin und .NET auf. Es ermöglicht es somit, unter Verwendung von C# native Anwendungen für Android, iOS und macOS zu entwickeln. Eine Lizenz schlägt allerdings mit 999 US-Dollar zu Buche. Dafür erhält der Entwickler ein Framework mit einer Vielzahl von Projektvorlagen und Komponenten für das Entwickeln von plattformübergreifenden Apps.

Des Weiteren hilft ein leistungsfähiger Form Builder bei Oberflächen mit Optionsfeldern, Auswahllisten und weiteren GUI-Komponenten. Das gilt auch für eine Kartenansicht (Map) und eine Kalenderkomponente. Neben der Einbindung kontextbezogener Hilfe stellt Intersoft Solutions Releases, Support und viele gut dokumentierte Beispiele zur Verfügung, die beim Entwickeln weiterhelfen und Lösungsansätze vorstellen.

Doch im Vordergrund steht – und das ist für Entwickler von besonderer Bedeutung –, dass viele Funktion von Crosslight vor allem den Aufwand bei der Nutzung des MVVM-Patterns verringern sollen.

Schließlich gilt auch für dieses Framework, dass es die bewährten Praktiken für MVVM umsetzt.



Visual Studio um Xamarin erweitern (Bild 2)

Installation der Frameworks

Da beide MVVM-Frameworks für das plattformübergreifende Entwickeln auf Xamarin für .NET setzen, sollte für die nachfolgenden Beispiele Xamarin für Visual Studio installiert sein. Sie können aber auch Xamarin Studio nutzen oder Xamarin für Visual Studio einfach nachinstallieren.

Zum nachträglichen Einrichten öffnen Sie in der *Systemsteuerung* einfach *Programme | Programme und Features*, wählen dann den Eintrag *Microsoft Visual Studio 2015* aus, rufen ▶

das Kontextmenü dazu auf und selektieren *Ändern*. Aktivieren Sie dann im Installationsprogramm der IDE den Punkt *Modify*. Wählen Sie in der Rubrik *Cross Platform Mobile Development* den Eintrag *C#/.NET (Xamarin)* aus (Bild 2). Mit einem Klick auf *Next* installieren Sie Xamarin.

Somit können jetzt auch Anwendungen, die Sie mit den Tools Xamarin.iOS und Xamarin.Android in C# entwickeln, auf die Plattformen iOS und Android gelangen. Die wichtigsten Elemente, um dabei die beiden genannten MVVM-Frameworks einzubinden, sind die Verwendung von portablen Klassenbibliotheken (Portable Class Library, PCL) sowie die geteilten Projekte und geteilter Code („Shared Projects" und „Shared Code"). Mehr hierzu erfahren weiter unten bei den Erläuterungen der verwendeten Beispiele.

Das MVVM Light Toolkit lässt sich sehr komfortabel über die Verwaltungsfunktion von NuGet in neue oder bestehende Projekte einbinden. Hierfür öffnen Sie einfach den Menüpunkt *NuGet Package Manager | Manage NuGet Packages for Solution* im Register *Tools* oder im Kontextmenü der geöffneten Projektmappe. Im Dialog, der daraufhin folgt, können Sie unter *Browse* nach der gewünschten Bibliothek suchen, in diesem Fall *MvvmLight* (Bild 3).

Über *Install* wird die Bibliothek heruntergeladen und dem Projekt hinzugefügt. Die NuGet-Konfiguration befindet sich in der Datei *packages.config* in der Projektmappe.

Das Crosslight-Framework von Intersoft Solutions bekommen Sie als komplette Entwickler-Tool-Edition, und es wird unter der Bezeichnung „Mobile Studio" auch als Trial-Version für 30 Tage als Download angeboten [6]. Nach der Installation über den Installations- und Konfigurationsassistenten der Mobile-Studio-Edition steht das Framework in Visual Studio oder in Xamarin Studio zur Verfügung.

Die Beispielanwendungen

Im Folgenden wird jeweils ein Beispiel für das entsprechende Framework betrachtet. Es zeigt für den jeweiligen Fall die Art und Weise, wie das Framework das MVVM-Muster umsetzt. Beide Beispielprojekte dienen als Testprojekte und stehen als Download zur Verfügung.

Das Beispiel für das MVVM Light Toolkit Framework stammt direkt von Laurent Bugnion, dem Entwickler des Frameworks, und steht in einem OneDrive-Laufwerk für Interessierte zum Download bereit [7].

Nach dem Extrahieren der Datei *XamFormsDemo.zip* können Sie die Projektmappe der Beispielanwendung mit Visual Studio öffnen. Bild 4 zeigt die ausgewählte Mappe, die folgende Ordner und Projekte enthält:

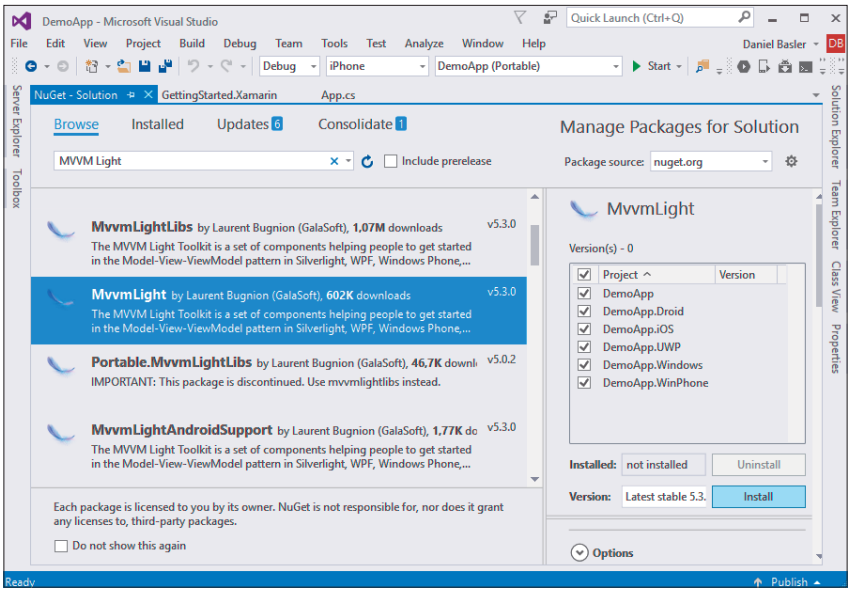
- *XamFormsDemo (Portable)*: Hier befindet sich die portable Klassenbibliothek, in welcher der gemeinsame Code für alle Projekte platziert wird.
- *XamFormsDemo.Android*: Dieser Zweig stellt das Xamarin.Android-Projekt dar.
- *XamFormsDemo.iOS*: Dieses Verzeichnis enthält den Code für das Xamarin.iOS-Projekt.
- *XamFormsDemo.WinPhone*: Hier ist der Code für das entsprechende Windows-Phone-Projekt zu finden.

Somit teilen sich alle drei App-Projekte – für Android, iOS und Windows Phone – denselben Quellcode im Portable-Projekt.

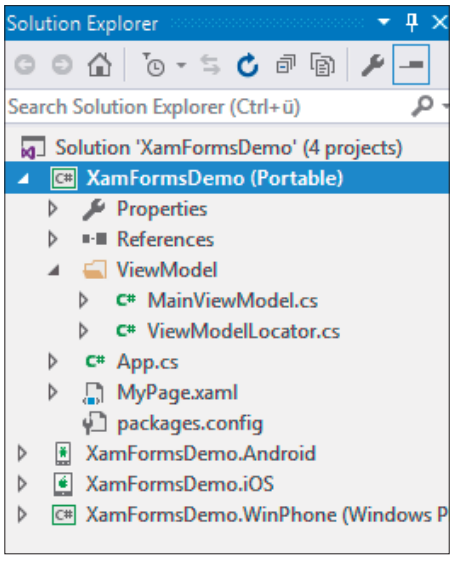
Die Beispielanwendung für das Crosslight-Framework steht im Download-Bereich von Intersoft Solutions zur Verfügung [8].

Wählen Sie dort das Beispiel *MVVM Samples* aus. Nach dem Download können Sie die Datei *basic-mvvm-master.zip* entpacken und die Projektmappe mit Visual Studio öffnen (Bild 5).

Auch hier besteht die Mappe aus einem Projekt *MvvmSample.Core(Portable),* das die gemeinsame Codebasis für alle Projekte enthält, sowie einem zusätzlichen Projekt für Windows-Store-Apps auf Basis von WinRT, das für das Beispiel aber nicht von Bedeutung ist.



Die Installation von MVVM Light Toolkit über NuGet (Bild 3)



Die Programmstruktur der Projektmappe XamFormsDemo (Bild 4)

Nach dem Laden der Projektmappe in Visual Studio müssen Sie noch den Verweis auf das Crosslight-Framework in den einzelnen Projekten einbinden.

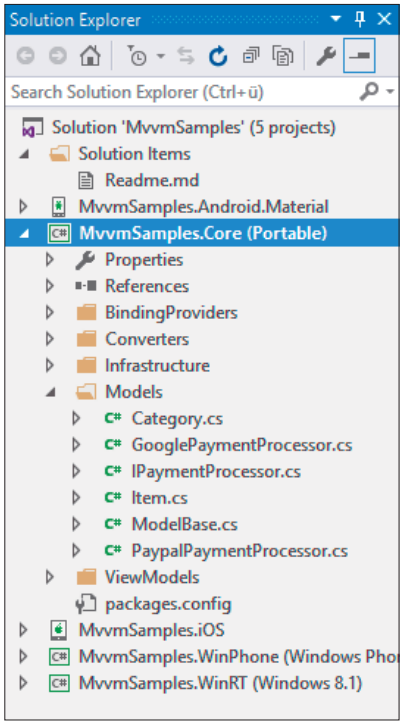
Wie sich am Projektaufbau zeigt, können beide Frameworks bei der Implementierung von plattformübergreifenden Entwicklungen entweder auf PCLs oder geteilte Projekte setzen.

Portable Klassenbibliotheken

Als portable Klassenbibliothek bezeichnet Microsoft eine Assembly, die den kleinsten gemeinsamen Nenner mehrerer Varianten des .NET Frameworks unterstützt und daher ohne Neukompilierung auf den entsprechenden Plattformen läuft. Visual Studio bietet hierfür die Projektvorlage *Portable Klassenbibliothek* (*Portable Class Library*). In den Projekteigenschaften kann man auswählen, welche Plattformen von ihr unterstützt werden sollen.

Zu beachten ist aber, dass hierbei – wie oben erwähnt – das Prinzip des kleinsten gemeinsamen Nenners gilt. Diese Untermenge nennt sich Profil. Somit gibt es ein Profil für Windows Phone, eines für Windows-Store-Apps für Xamarin.Android und eines für Xamarin.iOS. Die Profile stellen jeweils Untermengen des vollen .NET-Profils dar, sind aber nicht deckungsgleich.

Es wird also nur die vorhandene Schnittmenge der Zielplattformen zur Verfügung gestellt. Daher schränken sich mit jeder neuen Auswahl die zur Verfügung stehenden Methoden und Funktionen des vollständigen .NET-Profils ein. Die portable Klassenbibliothek kann darüber hinaus Ressourcendateien aufnehmen, um diese anwendungsübergreifend zu verwalten. Hierfür steht die Klasse *System.Resources.ResourceManager* zur Verfügung.



Die Programmstruktur der Projektmappe „MVVM Sample" von Crosslight (Bild 5)

Beide Frameworks, MVVM Light Toolkit und Crosslight, setzen in ihrem Beispiel jeweils eine PCL ein.

Shared Projects

Neben portablen Klassenbibliotheken bietet Xamarin auch die Möglichkeit, seinen Quellcode in Form von geteilten Projekten oder geteiltem Code auf verschiedenen Plattformen zu verwenden. Hierbei wird der Inhalt des geteilten Projekts automatisch in alle referenzierten Projekte kopiert und dort dann kompiliert. Dadurch entfällt die Ausgabe einer DLL wie bei der portablen Klassenbibliothek. Die referenzierten Projekte sind bei diesen Verfahren in der Regel plattformspezifisch. Somit lassen sich Komponenten einmal entwickeln und dann auf allen zur Verfügung gestellten Plattformen verwenden.

Beim Entwickeln mit geteilten Projekten ist allerdings zu beachten, dass diese keine anderen Projekte referenzieren können. Daher ist eine gründliche Planung der Projekte und der zu entwickelnden Komponenten notwendig.

Frameworks im Einsatz

Bei der Entwicklung einer Applikation ist es immer sehr verführerisch, rein in Benutzeroberflächen zu denken. Dieses Problem lösen die beiden MVVM-Frameworks im Wesentlichen mit der Implementierung des Entwurfsmusters. Es ▶

Listing 1: IoC mit dem MVVM Light Toolkit

```
using GalaSoft.MvvmLight.Ioc;
using Microsoft.Practices.ServiceLocation;

namespace XamFormsDemo.ViewModel
{
  public class ViewModelLocator
  {
    static ViewModelLocator()
    {
      ServiceLocator.SetLocatorProvider(() =>
        SimpleIoc.Default);
      SimpleIoc.Default.Register<MainViewModel>();
    }

    // Gets the Main property.
    [System.Diagnostics.CodeAnalysis.SuppressMessage(
      "Microsoft.Performance",
      "CA1822:MarkMembersAsStatic",
      Justification = "This non-static member is "
        + "needed for data binding purposes.")]
    public MainViewModel Main
    {
      get
      {
        return ServiceLocator.Current.
          GetInstance<MainViewModel>();
      }
    }
  }
}
```

Listing 2: IoC unter Crosslight

```
using System;
using Android.Runtime;
using Intersoft.Crosslight;
using Intersoft.Crosslight.Android.v7;
using MvvmSamples.ViewModels;

namespace MvvmSamples.Android.Fragments
{
  [ImportBinding(typeof(
    DelegateCommandBindingProvider))]
  public class DelegateCommandFragment :
    Fragment<DelegateCommandViewModel>
  {
    #region Constructors

    public DelegateCommandFragment()
    {
    }

    public DelegateCommandFragment(
      IntPtr javaReference,
      JniHandleOwnership transfer)
      : base(javaReference, transfer)
    {
    }

    #endregion

    #region Properties

    protected override int ContentLayoutId
    {
      get { return
        Resource.Layout.delegate_command_view; }
    }

    #endregion

    #region Methods

    #region Method

    protected override void Initialize()
    {
      base.Initialize();

      this.AddBarItem(new BarItem("ClearButton",
        CommandItemType.Cancel));
      this.IconId = Resource.Drawable.ic_toolbar;
    }

    #endregion

    #endregion
  }
}
```

soll den Entwickler nämlich dazu anhalten, nach Möglichkeit immer eine typische Interaktion zwischen Benutzer, View und ViewModell herauszuarbeiten: Der Anwender der Software führt in einer View eine bestimmte Aktion aus, die View gibt daraufhin ein Ereignis an das View-Modell weiter. Somit kann das View-Modell darauf reagieren und der View konkrete Anweisungen geben.

Diese Vorgabe lässt sich mit den MVVM-Frameworks sehr gut umsetzen, da diese schon bekannte Muster und Komponenten zur Verfügung stellen. Dazu zählen unter anderen die Schnittstelle *ICommand*, ein Event-Bus, Inversion of Control (IoC) und die *INotifyPropertyChanged*-Implementierungen.

Die wichtigsten Klassen befinden sich bei beiden Frameworks im jeweiligen *Core*-Projekt: Beim MVVM Light Toolkit im Ordner *ViewModel* sind es die Klassen *ViewModelLocator.cs* (Listing 1) und *MainViewModel.cs*. Die *ViewModelLocator*-Klasse fasst die Definitionen aller View-Modelle zusammen, sodass sie sich auf Anfrage zwischenspeichern und abrufen lassen; das geschieht über Dependency Injection. Unter Crosslight findet sich im Ordner *ViewModels* die Klasse *IocViewModel.cs*, siehe Listing 2. Beide Frameworks arbeiten – erkennbar im Quellcode – mit dem Entwurfsmuster Inversion of Control.

Somit gibt die Anwendung die Steuerung der Ausführung bestimmter Unterprogramme einfach an das Framework ab. Um einen Standardcontainer zu konfigurieren, werden die benötigten Abhängigkeiten entsprechend registriert. Dadurch lässt sich auch die View über ein Attribut in der Codebehind-Datei oder Activity zuordnen. Daher können verschiedene Sichten mit dem View-Modell verknüpft werden.

Da das Datenmodell (Model) in MVVM die Daten abbildet – sehr häufig direkt die Datenquelle –, Sie für die Funktionsweise der Frameworks aber keine Datenbindung über das Datenmodell vornehmen wollen, soll das Model im eigentlichen Sinne hier nicht weiter interessieren. Es lässt sich anhand der beiden Beispiele im Code nachvollziehen.

Kommunikation

Um in einer Applikation die Kommunikation zwischen den Anwendungsbausteinen zu ermöglichen – sprich: einfach Informationen anzuzeigen –, ist unter MVVM die Datenbindung notwendig, um von der View auf das View-Modell zuzugreifen. Das MVVM-Muster stellt dafür vier einfache Prinzipen auf:

- *Observable*-Objekte implementieren die *INotifyPropertyChanged*-Schnittstelle.

Listing 3: RelayCommand implementieren

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;

namespace XamFormsDemo.ViewModel
{
  public class MainViewModel : ViewModelBase
  {
    // The <see cref="ClickCount" /> property's name.
    public const string ClickCountPropertyName =
      "ClickCount";

    private int _clickCount;

    // Sets and gets the ClickCount property.
    // Changes to that property's value raise the
    // PropertyChanged event.
    public int ClickCount
    {
      get
      {
        return _clickCount;
      }
      set
      {
        if (Set(() => ClickCount, ref _clickCount,
          value))
        {
          RaisePropertyChanged(() =>
            ClickCountFormatted);
        }
      }
    }

    public string ClickCountFormatted
    {
      get
      {
        return ClickCount == 0
          ? "Click me please"
          : string.Format(
          "The button was clicked {0} time(s)",
          ClickCount);
      }
    }

    private RelayCommand _incrementCommand;

    // Gets the IncrementCommand.
    public RelayCommand IncrementCommand
    {
      get
      {
        return _incrementCommand
          ?? (_incrementCommand = new RelayCommand(
          () =>
          {
            ClickCount++;
          }));
      }
    }
  }
}
```

- Listen, die im Verlauf einer Anwendung geändert werden können, sollen in *ObservableCollection<T>*-Instanzen gespeichert werden.
- Bei Änderung des Inhalts einer Liste wird das *CollectionChanged*-Ereignis ausgelöst.
- Eigene Objekte müssen die *ICommand*-Schnittstelle implementieren. Diese Schnittstelle definiert eine *Execute()*- und eine *CanExecute()*-Methode.

Unter MVVM Light Toolkit werden diese Schnittstellen von konkreten Klassen implementiert, wie von *ViewModelBase* (implementiert *INotifyPropertyChanged*) und *RelayCommand* (implementiert *ICommand*). Im Beispiel können Sie die Integration in der Klasse *MainViewModel.cs* in Listing 3 sehen.

Hier wird die Schaltfläche mit der Beschriftung *Increment Counter* auf der Hauptseite zum Binding über die Methode *IncrementCommand()* in *MainViewModel* benutzt. Wenn die Methode aufgerufen wird, erhöht dieser Befehl den Zähler und setzt die *WelcomeTitle*-Eigenschaft. Diese löst das *PropertyChange*-Ereignis aus, und das entsprechende Binding sorgt für die Aktualisierung der *TextBox* in der *MainPage* der Anwendung.

Für komplexe Anwendungen bietet das MVVM Light Toolkit noch die *Messenger*-Klasse zur Implementierung an. Hierbei handelt es sich um das Entwurfsmuster des *Mediator*-Musters als direkter Vermittler. Diese Klasse regelt und steuert den Austausch von Nachrichten zwischen einzelnen Objekten. Sie wird hauptsächlich für das Senden von Nachrichten zwischen View-Modellen verwendet.

Die *Messenger*-Klasse verringert die Kopplung zwischen den View-Modellen und ermöglicht die Kommunikation mit anderen View-Modellen ohne ein entsprechendes Binding. Die Kooperation der Objekte erfolgt dann über die *Messenger*-Klasse in Form des *Mediator*-Musters.

Crosslight nutzt für das Binding auch eine eigene *BindingProvider*-Klasse. Listing 4 zeigt den übersichtlichen Aufbau der Klasse im Beispielprojekt.

Das *[ImportBinding]*-Attribut in der Android-Klasse (Listing 5) weist der benötigten Klasse die verbindlichen Defi- ▶

Listing 4: Die BindingProvider-Klasse

```
using Intersoft.Crosslight;

namespace MvvmSamples
{
  public class DelegateCommandBindingProvider :
    BindingProvider
  {
    #region Constructors
    public DelegateCommandBindingProvider()
    {
      this.AddBinding("TipAmountLabel",
        BindableProperties.TextProperty,
        new BindingDescription("TipAmount")
      { StringFormat = "{0:c}" });
      this.AddBinding("TotalAmountLabel",
        BindableProperties.TextProperty,
        new BindingDescription("TotalAmount")
      { StringFormat = "{0:c}" });
      this.AddBinding("ChargeTextField",
        BindableProperties.TextProperty,
        new BindingDescription("MealAmount",
        BindingMode.TwoWay)
      {
        UpdateSourceTrigger =
          UpdateSourceTrigger.PropertyChanged
      });
      this.AddBinding("TipTextField",
        BindableProperties.TextProperty,
        new BindingDescription("TipPercentage",
        BindingMode.TwoWay)
      {
        UpdateSourceTrigger =
          UpdateSourceTrigger.PropertyChanged
      });
      this.AddBinding("ResultView",
        BindableProperties.IsVisibleProperty,
        "IsResultVisible");

      this.AddBinding("CalculateButton",
        BindableProperties.CommandProperty,
        "CalculateCommand");
      this.AddBinding("CalculateButton",
        BindableProperties.CommandParameterProperty,
        "SomeParameter", true); // direct source
      this.AddBinding("ClearButton",
        BindableProperties.CommandProperty,
        "ClearCommand");
    }
    #endregion

  }
}
```

nitionen als Importdatei zu. Wurde das Binding ordnungsgemäß in die Klasse importiert, so wird die Definition automatisch wirksam. Objekte, die einseitig gebunden sind, werden automatisch aktualisiert, wenn sich der Quellcode im *Core*-Projekt ändert; bidirektional gebundene Objekte werden je Projekt aktualisiert.

Auch wenn die Datenbindung in den Frameworks auf unterschiedliche Art und Weise implementiert ist, so stehen dennoch alle für MVVM benötigten Modi der Datenbindung zur Verfügung:

- *OneWay*: Änderungen der Quelle verändern das Ziel.
- *TwoWay*: Änderungen erfolgen in beide Richtungen (Voreinstellung).
- *OneWayToSource*: Änderungen am Ziel führen zu Änderungen der Quelle.
- *OneTime*: Bei der Initialisierung des Ziels werden Daten der Quelle abgefragt, Änderungen werden nicht weitergegeben.

Zu beachten ist dabei, dass bei MVVM die Elemente des GUI immer das Ziel des Bindings darstellen.

Layout und Controls

Das MVVM Light Toolkit setzt bei Layouts und Controls auf Xamarin.Forms und stellt somit im eigentlichen Framework keine Komponenten zur Verfügung. Ganz anders sieht das bei Crosslight aus. Es bietet neben bereits fertigen Diensten für E-Mail, Benachrichtigungen und weitere Funktionen zum Auslesen von Informationen auf Betriebssystemebene auch eine Vielzahl von fertigen Layouts und Controls an. Des Weiteren bietet es auch noch eine große Anzahl von Projektvorlagen für den Datenzugriff, aber auch für erweiterten Service für GPS-Nutzung und Plug-ins für die Entwicklung.

Navigation

Die Navigation in der Anwendung lösen beide Frameworks mithilfe eines Navigationsdienstes. In MVVM Light Toolkit wird er einfach über die *Locator*-Eigenschaft aufgelöst. Hierfür reicht der simple Aufruf in der folgenden Form:

```
App.Locator.NavigationService.NavigateTo(PageKey)
```

PageKey stellt die entsprechende Seite dar. Somit wird *NavigationService* einfach über einen zugeordneten Schlüssel konfiguriert.

Auch bei Crosslight wird die Navigation über einen Dienst geregelt. Crosslight bietet allerdings deutlich mehr Möglichkeiten für die Anpassung der Navigation im Layout-Bereich. Für die gewählte Navigation unter MVVM sind beide Frameworks fast identisch.

Listing 5: Die Android-Klasse zum Binding

```
using System;
using Android.Runtime;
using Intersoft.Crosslight;
using Intersoft.Crosslight.Android.v7;
using MvvmSamples.ViewModels;

namespace MvvmSamples.Android.Fragments
{
  [ImportBinding(typeof(
    DelegateCommandBindingProvider))]
  public class DelegateCommandFragment :
    Fragment<DelegateCommandViewModel>
  {
    #region Constructors

    public DelegateCommandFragment()
    {
    }

    public DelegateCommandFragment(IntPtr
      javaReference, JniHandleOwnership transfer)
      : base(javaReference, transfer)
    {
    }

    #endregion

    #region Properties

    protected override int ContentLayoutId
    {
      get { return
        Resource.Layout.delegate_command_view; }
    }

    #endregion

    #region Methods

    #region Method

    protected override void Initialize()
    {
      base.Initialize();

      this.AddBarItem(new BarItem("ClearButton",
        CommandItemType.Cancel));
      this.IconId = Resource.Drawable.ic_toolbar;
    }

    #endregion

    #endregion

  }
}
```

Fazit

Für die Entwicklung einer plattformübergreifenden Anwendung lohnt sich der Einsatz des Entwurfsmusters Model-View-ViewModel, kurz MVVM, auf jeden Fall. Indem Frameworks für dieses Muster Datenbindung, Dependency Injection, Lokalisierung und Navigation umsetzen und dem Entwickler zur Verfügung stellen, erweisen sie sich in dieser Hinsicht als sehr praktisch und zeitsparend.

Mit dem Framework MVVM Light Toolkit steht ein leichtgewichtiges Tool für den schnellen und effektiven Einsatz des Model-View-ViewModel-Entwurfsmusters zur Verfügung.

Bei dem Framework Crosslight ist die kommerzielle Ausrichtung sofort zu erkennen. Die Vielzahl seiner nützlichen Tools und Vorlagen kann besonders bei zeitkritischen und sehr komplexen Anwendungen den Programmierer sehr gut unterstützen und den Entwicklungsaufwand reduzieren. Einziger Wermutstropfen sind die Kosten für das Framework von fast 1000 Dollar.

Für jeden Entwickler, der es mit plattformübergreifenden Anwendungen zu tun bekommt, lohnt sich ein Blick auf MVVM-Frameworks. Sie erleichtern ihm den Einstieg in dieses Entwurfsmuster und in dessen richtige Implementierung ungemein. ■

[1] Wikipedia: Model View ViewModel, www.dotnetpro.de/SL1709MVVMVergleich1

[2] CoreMVVM – A Simple MVVM Framework, https://coremvvm.codeplex.com

[3] Caliburn.Micro, http://caliburnmicro.com

[4] Prism, https://github.com/PrismLibrary

[5] GalaSoft/Laurent Bugnion, www.galasoft.ch

[6] Crosslight/Mobile Studio, www.dotnetpro.de/SL1709MVVMVergleich2

[7] Beispiel, www.dotnetpro.de/SL1709MVVMVergleich3

[8] Get Started with Intersoft Studio, www.dotnetpro.de/SL1709MVVMVergleich4

Daniel Basler
ist Senior Consultant für Microsoft-Technologien und beschäftigt sich darüber hinaus auch mit Datenbanken und Compiler-Bau.

dnpCode A1709MVVMVergleich