

XAMARIN IN DER PRAXIS

Einer für alle

Mit der Xamarin Platform native Apps für die wichtigsten Plattformen erstellen.

Am 18. März 2016 schloss Microsoft die Übernahme von Xamarin offiziell ab (siehe Kasten Wer oder was ist Xamarin?). Keine zwei Wochen später, am 31. März, wurde bekannt gegeben, dass die Xamarin-Produkte ab sofort kostenlos in Visual Studio verfügbar sind. Doch welche Möglichkeiten bieten die Xamarin-Produkte den .NET Entwicklern? Dieser Artikel betrachtet Xamarin.Forms näher, das am 28. Mai 2014 angekündigt wurde.

Licht ins Dunkel

Sieht man sich die Geschichte von Xamarin und den Xamarin-Produkten an, verwundert es nicht, dass viele nicht durchblicken, was Xamarin nun tatsächlich für Möglichkeiten auf welcher Plattform bietet. Dass mit den Xamarin-Produkten irgendwie native Apps für Apple iOS und Android auf Basis von C# und .NET erstellt werden können, ist vielen noch bekannt. Doch wie das genau geht, welche Voraussetzungen man dafür braucht und welche Rolle Xamarin Studio und Xamarin.Forms dabei spielen, ist oft unklar. Der Kasten Produkte von Xamarin bietet hier einen ersten Überblick. Wie zu sehen, steht neben C# lediglich F# zur Verfügung. VB.NET wird von Xamarin nicht unterstützt.

	MAC OS X	WINDOWS	
Development Environment	XAMARIN STUDIO	VISUAL STUDIO	XAMARIN STUDIO
Xamarin.iOS	Yes	Yes (with Mac computer)	No
Xamarin.Android	Yes	Yes	Yes
Xamarin.Forms	iOS & Android only	Android, Windows Phone, Windows (iOS with Mac computer)	Android only
Xamarin.Mac	Yes	No	No

Die Tabelle zeigt, welche Technologien sich mit welchen Tools und Plattformen einsetzen lassen (Bild 1)

Durch die rasante Entwicklung der Xamarin-Produkte ist ebenfalls oft unklar, was konkret benötigt wird, um ein Ergebnis zu erzielen.

Grundsätzlich haben alle Xamarin-Produkte das Ziel, dem Entwickler dabei zu helfen, schneller und produktiver native Apps für verschiedene Plattformen zu entwickeln und dabei so viel Code wie möglich gemeinsam nutzen zu können. Da bei allen Xamarin-Produkten auf die Programmiersprachen C# oder F# gesetzt wurde, ist dies grundsätzlich auch kein Problem. Entscheidend für die Auswahl der richtigen Xamarin-Technologie und Plattform sind – wie so oft – die Anforderungen an die zu erstellende App.

● Wer oder was ist Xamarin?

Der Weg bis zur Gründung der Firma Xamarin und der letztendlichen Übernahme durch Microsoft im März 2016 ist sehr interessant und führt über mehrere Stationen. Xamarin selbst wurde erst im Mai 2011 von Nat Friedman und Miguel de Icaza gegründet. Miguel de Icaza, der als begeisterter Linux-Entwickler unter anderem das GNOME-Projekt mit ins Leben gerufen hat und sich intensiv am Wine-Projekt beteiligte, war von Anfang an der Überzeugung, dass das .NET Framework die ideale Entwicklungsplattform auch für andere Betriebssysteme als Windows sei, und so entstand Mono. Seine Firma Ximian, die er 1999 ebenfalls mit Nat Friedman gründete, startete Mono bereits im Juli 2001 als Open-Source-Projekt und als freie .NET-Alternative, um C# auf Linux, Mac OS X und verschiedene Unix-Derivate zu bringen.

2003 erschien neben Mono mit MonoDevelop auch eine eigene Entwicklungsumgebung, basierend auf SharpDevelop, die neben Windows auch unter Linux und Mac OS X läuft.

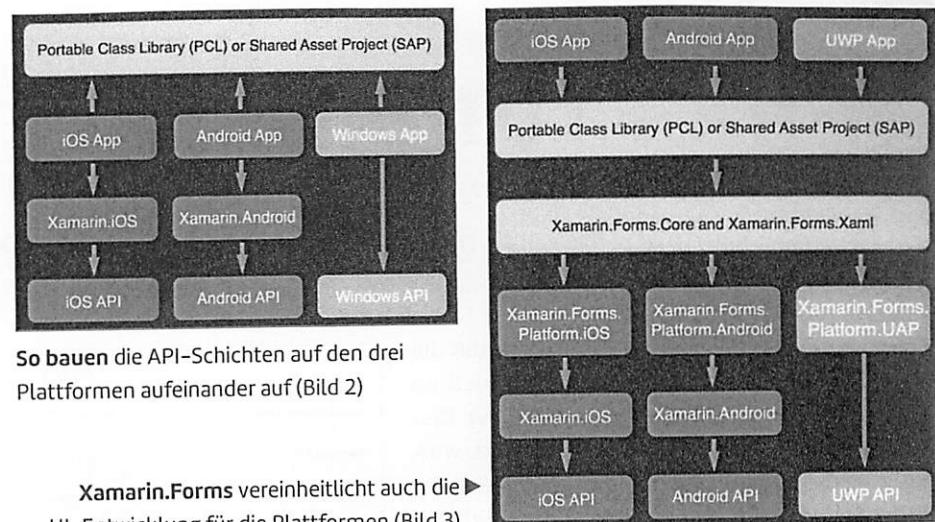
2003 übernahm Novell neben Suse auch die Firma Ximian, um ihre Präsenz im Linux-Bereich auszubauen. 2009 kündigte Novell MonoTouch an, mit dem echte, native Apps für das iPhone mittels C# und .NET entwickelt werden konnten. 2011 folgte mit Mono for Android das entsprechende Pendant für Android. Beide Produkte wurden kommerziell vermarktet. Als Novell 2011 von Attachmate gekauft wurde, hatte man dort kein Interesse an Mono und seinen mobilen Ablegern und entließ kurzerhand den größten Teil der Mono-Entwickler, darunter auch die beiden Ximian-Gründer. Diese machten aus der Not eine Tugend und gründeten Xamarin. Die Firma kümmerte sich fortan um die Weiterentwicklung der Mono-basierten Produkte und baute sie so erfolgreich aus, dass Microsoft im März dieses Jahres schließlich das Unternehmen übernahm. Das hatte zwischenzeitlich mehr als 350 Mitarbeiter und 15.000 Kunden. Laut Xamarin arbeiten derzeit mehr als 1,4 Millionen Entwickler mit Xamarin-Produkten.

Die Qual der Wahl

Die erste Entscheidung, die der angehende Xamarin-App-Entwickler zu treffen hat, ist die Wahl der Betriebssystemplattform für das Produktionssystem und damit auch des einzusetzenden Tools. Fällt die Entscheidung auf den Apple Macintosh, so bleibt als Entwicklungsumgebung lediglich Xamarin Studio übrig. Fällt die Wahl auf Windows, so ist Visual Studio das Tool der Wahl.

Die Auswahl der entsprechenden Plattform bedingt auch die jeweiligen Zielplattformen, denn mit Xamarin Studio auf dem Mac können zum Beispiel keine Windows-Versionen der App erstellt werden – dafür aber Apps für iOS, Mac OS X und Android.

Ein verbreiteter Mythos ist, dass eine App für iOS ausschließlich auf dem Mac mittels Xamarin Studio entwickelt werden kann. Dies stimmt nicht, denn auch mit Visual Studio lässt sich eine App für iOS entwickeln. Nutzt man Xamarin.iOS, kann sogar ein grafischer Designer in Visual Studio für das Design der Oberfläche eingesetzt werden (Bild 1).



● Produkte von Xamarin

Die Xamarin-Produkte sind in vier Hauptgruppen aufgeteilt:

- Xamarin Platform,**
- Xamarin Test Cloud,**
- Xamarin Insights,**
- Xamarin University.**

Die Xamarin Platform wiederum, die in diesem Artikel behandelt wird, beinhaltet folgende Produkte:

- Xamarin.iOS** (ehemals MonoTouch): Zur Entwicklung von nativen iOS-Apps in C# und F#.
- Xamarin.Android** (ehemals Mono for Android): Zur Entwicklung von nativen Android-Apps in C# und F#.
- Xamarin.Mac**: Zur Entwicklung von nativen Mac-OS-X-Anwendungen in C# und F#.
- Xamarin.Forms**: Zur Entwicklung von nativen iOS-, Android-, Windows-Phone-8.1-, Windows-Desktop-8.1- und Universal-Windows-Platform-Apps in C# und F#.
- Xamarin Studio**: Entwicklungsumgebung, die auf Mac OS X läuft. Mit Xamarin Studio können native Apps basierend auf den oben genannten Produkten entwickelt werden.
- Xamarin for Visual Studio**: Integration der Xamarin-Produkte in Microsoft Visual Studio. So ist es beispielsweise möglich, in Visual Studio iOS-Apps zu entwickeln und sogar die Benutzeroberfläche per grafischem Designer direkt in Visual Studio zu entwerfen.

Soll diese App dann getestet und veröffentlicht werden, ist allerdings in der Regel ein Apple-Rechner mit Mac OS X notwendig, der jedoch lediglich im Hintergrund als sogenannter Xamarin Mac Agent aktiv sein muss – doch dazu später mehr.

Wird Visual Studio unter Windows zur App-Entwicklung eingesetzt, stehen ungleich mehr Zielplattformen zur Verfügung. Neben Apple iOS und Android kann die App für Windows 8.1, Windows Phone 8.1, sowie ab Windows 10 und Visual Studio 2015 auch für die Universal Windows Platform (UWP) erstellt werden. Dank Letzterem ist sie auf beliebigen Windows-10-Geräten inklusive Smartphones und Xbox lauffähig. Die einzige Plattform, die unter Windows nicht zur Verfügung steht, ist Mac OS X.

Xamarin.Forms

Bis zum 28. Mai 2014 konnte eine App mittels Xamarin.iOS und Xamarin.Android zwar in C# entwickelt und die Logik mittels Code Sharing und/oder Portable Class Libraries plattformübergreifend wiederverwendet werden, jedoch musste die Oberfläche der App pro Zielplattform neu erstellt werden. Wenngleich dies bereits in Visual Studio möglich war, verursachte es dennoch Mehraufwand (Bild 2).

Den Xamarin-Entwicklern ging dies nicht weit genug, und so entstand Xamarin.Forms mit dem Ziel, auch die Oberflächenentwicklung stark zu vereinfachen und noch mehr Code für alle Plattformen gemeinsam nutzen zu können. Dabei setzt Xamarin.Forms nicht nur auf C#, sondern auch auf XAML. Im Idealfall entwickelt man im zentralen, portablen Projekt die XAML-Dateien und kann diese dann auf allen Plattformen verwenden (Bild 3). Beim Build werden die Xamarin-Steuerelemente dann auf den Plattformen nativen Steuerelementen zugeordnet.

Informationen ohne Ende

Wer schon einmal eine App unter Windows entwickelt hat, kann sich sicherlich vorstellen, dass die Entwicklung für mehrere, auch noch vollkommen verschiedene Plattformen nicht gerade nebenbei geschieht. Die Xamarin-Produkte können dem Entwickler viel an Arbeit abnehmen – trotzdem bleibt ►

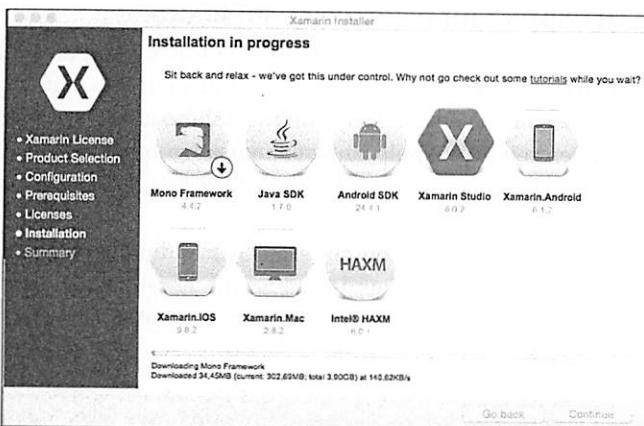
noch genug zu tun. Mit bestimmten Grundlagen der Android- und/oder Apple-iOS-Entwicklung muss man sich dennoch auseinandersetzen.

Auch hier lässt Xamarin aber die Entwickler nicht allein und bietet in zahlreichen Anleitungen Schritt-für-Schritt-Unterstützung an, zum Beispiel dazu, wie man ein Apple Development Certificate bekommt, um eine App auch auf einem Apple-Gerät testen zu können – inklusive dem Thema Free Provisioning [1].

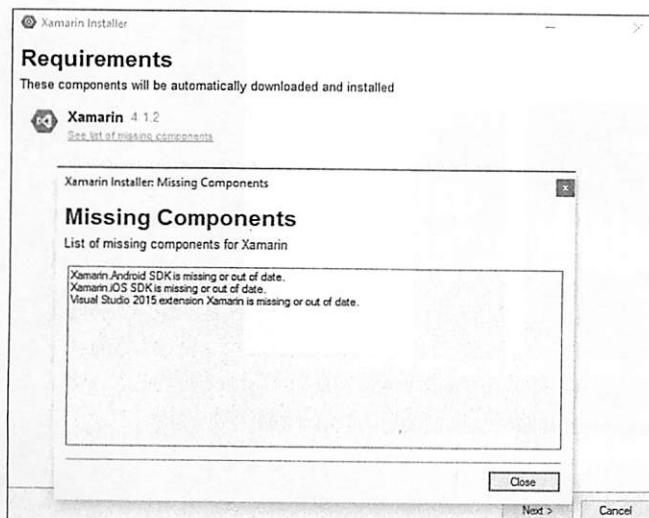
Unter [2] findet sich ein 1200 Seiten starkes, kostenloses Microsoft Press E-Book, in dem Autor Charles Petzold detailliert auf Xamarin.Forms eingeht. Wer darüber hinaus noch Fragen oder Probleme hat, findet im Xamarin-eigenen Forum [3] oder bei StackOverflow [4] Hilfe oder kann dort seine Fragen stellen. Auch gibt es ein kostenloses zweiwöchiges Seminar bei edX [5]. Zahlreiche Beispiele unter [6] sowie fertige App-Templates [7] runden das Angebot ab.

Ran an die Apps

Wie eingangs erwähnt, stellte Microsoft die Xamarin-Produkte kurz nach der Übernahme des Unternehmens kostenlos allen Entwicklern zur Verfügung. Das bedeutet, dass Sie auch



Installation der Xamarin-Produkte auf einem Mac ... (Bild 4)



... und auf einem Windows-10-Rechner (Bild 5)

● Unterschiede zwischen PCL- und SAP-Projekten

Wer schon mit Universal Apps unter Windows 8.x beziehungsweise der Universal Windows Platform ab Windows 10 zu tun hatte, kennt das Konzept der Shared Asset Projects, kurz SAP. Hierbei handelt es sich nicht wirklich um Projekte, sondern um Code, der in einem Ordner in der Projektmappe gespeichert wird und zur Umwandlungszeit in die einzelnen Unterprojekte kopiert wird. Das heißt, bei einem SAP erspart sich der Entwickler das manuelle Kopieren des gemeinsamen Codes in die einzelnen Unterprojekte.

Da der gleiche Code in alle Unterprojekte kopiert wird, muss innerhalb des Codes in der Regel mittels Compiler-Symbolen auf die jeweilige Plattform abgefragt werden, wenn man spezifische Funktionen wie etwa den Zugriff auf das jeweilige Dateisystem einer Plattform nutzen will. Das heißt, der Code wird dann mit zahlreichen `#if __IOS__ #elif __ANDROID__` et cetera gespickt sein, was der Lesbarkeit nicht unbedingt förderlich ist.

Bei einer Portable Class Library, kurz PCL, hingegen handelt es sich tatsächlich um ein eigenständiges Projekt, das als DLL in die Unterprojekte eingebunden wird. Dabei muss bei der Erstellung einer PCL immer angegeben werden, welche .NET-Framework-Versionen die PCL unterstützen soll, und dementsprechend stehen auch die verschiedenen Namensräume zur Verfügung.

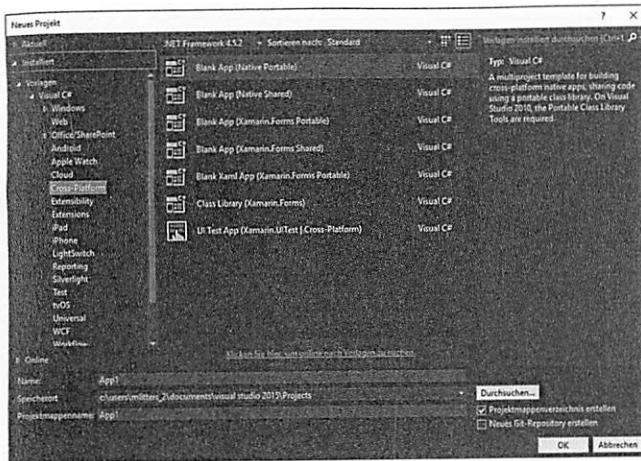
In Xamarin.Forms wird automatisch die Unterstützung für .NET Framework 4.5, ASP.NET Core 1.0, Windows 8, Windows Phone 8.1, Xamarin.Android, Xamarin.iOS und Xamarin.iOS (Classic) voreingestellt. Beide Varianten haben ihre Vor- und Nachteile, die es wie immer gegeneinander abzuwägen gilt. Die gute Nachricht ist, dass in einem SAP-Projekt selbstverständlich jederzeit eine oder mehrere PCLs hinzugefügt werden können, sodass man sich nicht unbedingt für eine Methode entscheiden muss.

mit der Visual Studio Community Edition ohne finanziellen Aufwand Apps für die drei gängigsten Plattformen erstellen können. Auch die Entwicklungsumgebung Xamarin Studio für Mac OS X steht in der Community Edition kostenlos zur Verfügung.

Der einfachste Weg, die aktuelle Version der Xamarin-Produkte zu installieren, geht über den Xamarin Installer. Dieser lässt sich unter [8] herunterladen und prüft den Rechner zunächst auf fehlende Komponenten. Bild 4 zeigt die Installation auf einem Apple Mac, Bild 5 auf einem Windows-10-Rechner mit Visual Studio 2015.

Let's Get Cross Platform

Hat man die Installation geschafft, kann in Visual Studio direkt mit einem neuen Projekt losgelegt werden. Hierzu gibt es einige neue Vorlagen, wie in Bild 6 zu sehen. Als Beispiel wird eine Cross-Platform-App auf Basis der *Blank Xaml App (Xamarin.Forms Portable)* erstellt. Da in Xamarin.Forms nicht unbedingt mit XAML gearbeitet werden muss, sondern die

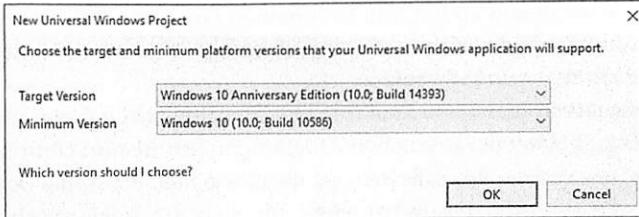


Neue Vorlagen für die Cross-Plattform-Entwicklung in Visual Studio (Bild 6)

Oberfläche auch komplett über den Code erzeugt werden kann, gibt es weitere Vorlagen ohne XAML.

Wie im Bild zu sehen, gibt es die Möglichkeit, ein Projekt auf Basis einer Portable Class Library (PCL) zu erstellen oder als sogenanntes Shared Asset Project (SAP) (siehe dazu auch den Kasten Unterschiede zwischen PCL- und SAP-Projekten). Da sich in Verbindung mit Xamarin.Forms jedoch Projekte auf Basis der PCL durchgesetzt haben und diese auch von Xamarin als Projekttyp empfohlen werden, gibt es für die Blank Xaml App wohl auch nur diese Vorlage.

Im nächsten Schritt kann dann die UWP Minimum- und Target-Version ausgewählt werden (Bild 7). Hierbei gilt es abzuwegen, welche Zielgruppe man ansprechen möchte. Durch die Windows-10-Zwangsupdates besteht jedoch kein so großes Problem wie zum Beispiel bei Android. Nach erfolgter Auswahl generiert Visual Studio insgesamt sechs Projekte



Auswahl der Minimum- und der Target-Version (Bild 7)

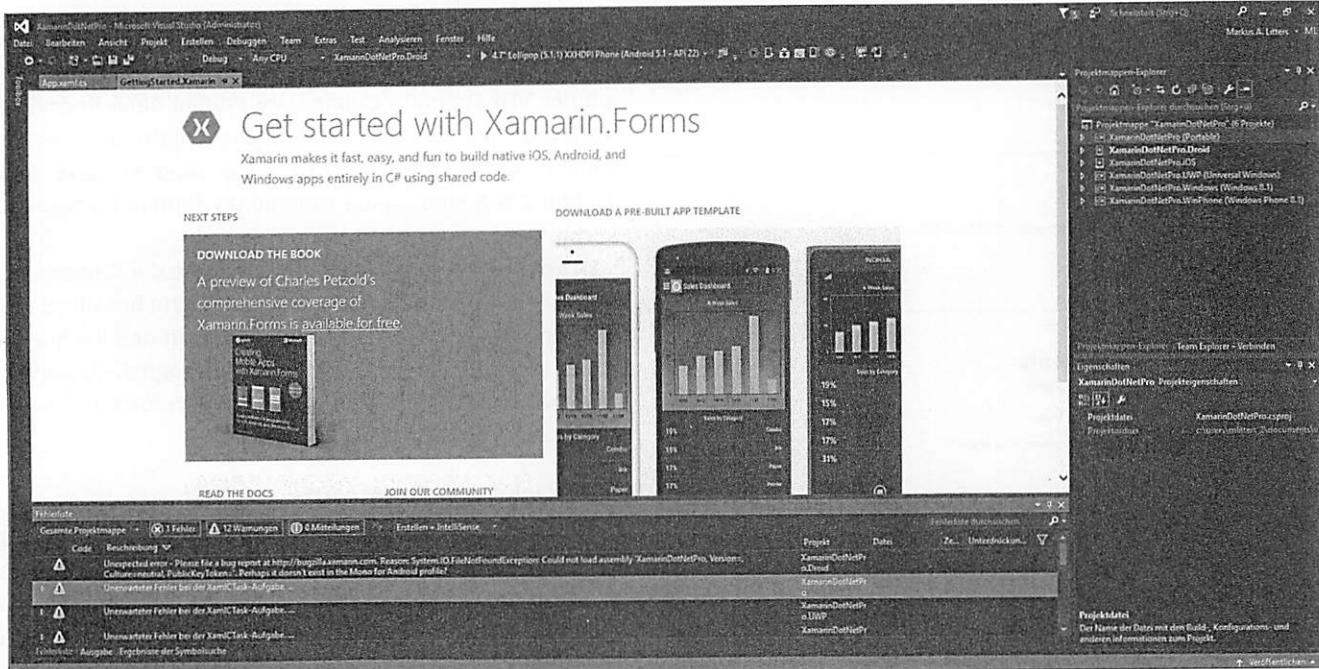
und erstellt eine Ausgabe, wie in Bild 8 zu sehen. Darin ist ebenfalls zu erkennen, dass es schon am Anfang einige Warnungen und sogar einen Fehler gibt, obwohl ein neues Projekt angelegt wurde. Diese Fehler und Warnungen lassen sich in der Regel leicht beseitigen, indem man zunächst per Rechtsklick auf die Projektmappe den Punkt *NuGet-Pakete wiederherstellen* auswählt und nach Abschluss dieser Arbeit die Projektmappe noch einmal komplett neu erstellt.

Nach einem Klick auf *Debug* sollte sich dann auch der Android-Emulator von Microsoft öffnen und dieser eine Ausgabe anzeigen, wie sie in Bild 9 zu sehen ist. In Visual Studio stehen allerhand Geräte und Android-Versionen (ab 4.0.3 API 15) zur Emulation bereit.

Der Apfel fällt nicht weit vom Ästen

Soll die neu erstellte App unter iOS getestet werden, ist das schon etwas aufwendiger. Zunächst muss ein Mac mit OS X ab Yosemite (Version 10.10) und installiertem Xcode 7 vorhanden sein. Die Apple-Entwicklungsumgebung Xcode gibt es für Mac-Anwender kostenlos. Außerdem muss noch Xamarin Studio installiert sein.

Der Mac muss für die entfernte Anmeldung freigegeben sein und kann dann aus Visual Studio heraus via Xamarin Mac Agent verbunden werden (Bild 10). Dabei spielt es ►



Die neue Solution mit Warnungen und einem Fehler (Bild 8)

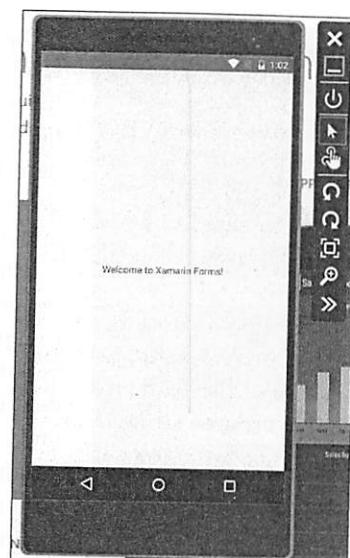
keine Rolle, ob Visual Studio auf einem Windows-Rechner innerhalb einer virtuellen Umgebung auf dem Mac läuft oder auf einem eigenständigen Windows-PC. Die Kombination aus Mac und virtuellem Windows-Rechner hat sich in der Vergangenheit vor allem deshalb bewährt, weil die Simulation von Apple-iOS-Geräten rein auf dem Mac innerhalb des iOS-Simulators von Xcode abgelaufen ist. Wollte man auf einem externen Windows-PC entwickeln, musste man sich Fernsteuerungs-Tools wie VNC oder TeamViewer bedienen, was nicht gerade komfortabel war.

Dank eines neuen Apple-iOS-Simulators für Windows, den Xamarin derzeit in einer Preview unter [9] zur Verfügung stellt, soll dieses Problem jedoch bald der Vergangenheit angehören. Zwar wird nach wie vor ein Mac als Umwandlungsrechner im Hintergrund benötigt, die Ausgabe des Simulators erfolgt jedoch auf dem Windows-Rechner und unterstützt sogar Touch-Bildschirme. Leider ist es dem Autor bisher noch nicht gelungen, den Simulator unter Windows zum Laufen zu bringen, trotz zahlreicher Versuche auf mehreren Rechnern und verschiedenen Xamarin-Versionen. Die oben genannten Foren zeigen sehr viele unterschiedliche Probleme, die aktuell noch mit dem Simulator auftreten. Diese werden hoffentlich bald behoben.

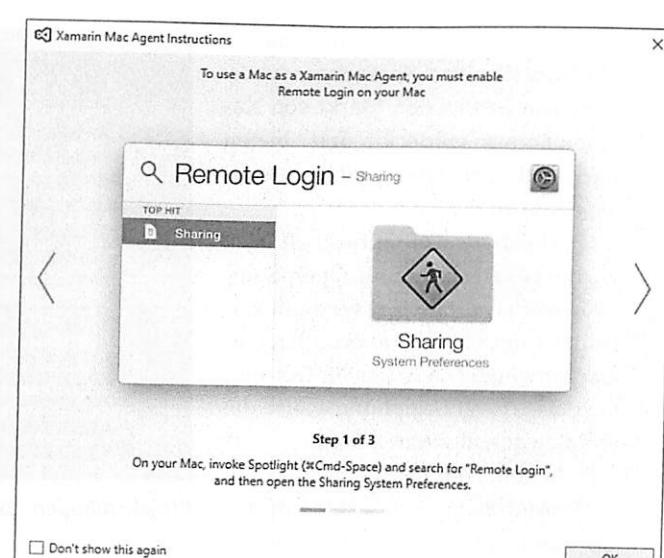
Läuft der iOS-Simulator klassisch auf dem Mac, sieht das Ergebnis wie in Bild 11 dargestellt aus. Auch auf dem Mac stehen verschiedenste Geräte mit unterschiedlichen iOS-Versionen zur Verfügung. Hier hängt es unter anderem davon ab, welches Betriebssystem und welche Xcode-Version auf dem Mac installiert sind, da neuere Versionen von Xcode und den iOS-SDKs auch nur auf neueren Versionen von Mac OS X laufen. Die gleiche Aktion ließe sich nun auch noch mit dem UWP-Projekt sowie den beiden Windows-8.1-Projekten ausführen, worauf hier jedoch verzichtet wird, da über UWP bereits an anderer Stelle in der dotnetpro berichtet wurde.

Mehr als Hallo Welt

Wie geht es aber nach dieser Hello-World-App weiter? Um



Die neue App im Android-Emulator (Bild 9)

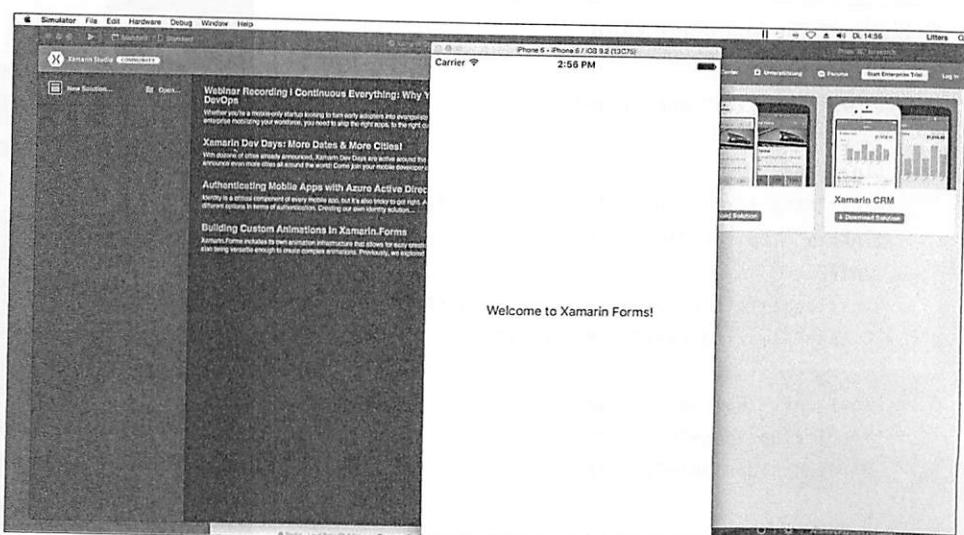


Fernmeldung am Mac für die Kompilierung einer iOS-App (Bild 10)

mittels Xamarin.Forms effektiv zu entwickeln, ist es sinnvoll, so viel Code wie möglich in das PCL-Projekt zu packen, in dem unter anderem bereits die *MainPage.xaml* zu finden ist (Bild 12). Der enthaltene Code ist der aus Listing 1 und ist dem XAML, das man aus anderen .NET-Projekten kennt, schon sehr ähnlich. Aufgrund der Tatsache, dass dieses XAML jedoch beim Umwandeln den nativen Steuerelementen der jeweiligen Zielplattform zugeordnet werden muss, handelt es sich um einen eigenen Xamarin-Dialekt. Als Nächstes fällt auf, dass es keinen grafischen Designer für XAML gibt. Unter [10] kann aber der Status des XAML-Previewers beobachtet werden. Er funktioniert in der Alpha-Version in Xamarin Studio auf dem Mac, eine VS-Version ist gerade erschienen

Zur Tat

Will man der neu erstellten App mehr Inhalt spendieren, so gibt es von Xamarin zahlreiche User Controls, die direkt ein-



Simulator mit leerem Projekt auf dem Mac (Bild 11)

gesetzt werden können. Auch haben die Hersteller von Komponentenbibliotheken bereits den Markt von Xamarin.Forms entdeckt und bieten nach und nach immer mehr Steuerelemente an.

Glücklicherweise entwickelt sich auch das Ökosystem im Open-Source-Bereich. Im Beispiel wird ein Calendar Control von Rebecca Groote laar verwendet, das per NuGet einfach ins Projekt beziehungsweise die Projekte geholt werden kann.

Dazu geht man per Rechtsklick auf die Projektmappe und wählt dort *NuGet-Pakete für Projektmappe verwalten*. Nun kann man das gewünschte Paket direkt in alle Projekte installieren lassen (Bild 13). Die Entwicklerin hat auf der GitHub-Seite [11] neben dem Quellcode einige gute Beispiele sowie ein Demo-Projekt und Beschreibungen des Steuerelements veröffentlicht, sodass einem schnellen Einsatz nichts im Weg steht. Im PCL-Projekt kann nun via XAML oder Code das neue Steuerelement verwendet werden. Listing 2 zeigt die MainPage.xaml des erweiterten Hallo-Welt-Beispiels. Hierin befinden sich neben dem Kalender noch eine Eingabekontrolle sowie ein Picker-Steuerelement. Diese sind standardmäßig in Xamarin vorhanden. Der Namensraum des Kalenderelements muss hingegen, wie man dies von XAML gewohnt ist, über einen Eintrag bei den Namensräumen im Rootelement hinzugefügt werden:

```
xmlns:controls="clr-namespace:XamForms
.Controls;assembly=XamForms.Controls.Calendar"
```

Ansonsten wird hier ein Grid verwendet, das den Bildschirm entsprechend in Zeilen einteilt. Dies funktioniert auf allen Plattformen.

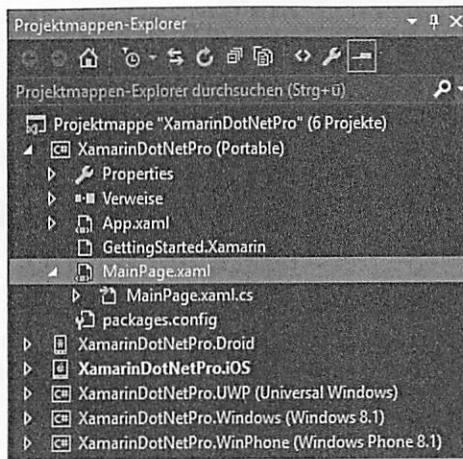
● Listing 1: Die MainPage.xaml

```
<?xml version="1.0" encoding="utf-8" ?>

<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/
    xaml"
    xmlns:local="clr-namespace:XamarinDotNetPro"
    x:Class="XamarinDotNetPro.MainPage">

    <Label Text="Welcome to Xamarin Forms!">
        VerticalOptions="Center"
        HorizontalOptions="Center" />

</ContentPage>
```



Projektmappen-Explorer der Solution (Bild 12)

Das Picker-Element wird im Beispiel verwendet, um die Tastaturart für die Eingabebox festzulegen. Das heißt, je nach Auswahl wird eine andere Bildschirmtastatur gezeigt, wie in Bild 14 auf allen drei Emulatoren/Simulatoren zu sehen ist.

Listing 3 zeigt die zugehörige Code-behind-Datei mit zwei Methoden, die Ereignisse aus den Steuerelementen verarbeiten. Im Konstruktor wird auf das Kalender-Element zugegriffen, und über die *SpecialDates*-Eigenschaft werden bestimmte Tage eingefärbt sowie über die *MinDate*-Eigenschaft die auswählbaren Tage festgelegt. Wird im Kalender auf einen Tag

geklickt, gibt die Methode *OnDateClick* eine Nachricht auf dem Bildschirm mit dem ausgewählten Tag aus.

Klickt der Benutzer auf die Picker-Auswahl unten im Bild und wählt dort einen anderen Tastaturtyp aus, springt die Methode *OnPickerSelectedIndexChanged* an. Sie ordnet den ausgewählten Tastaturtyp dem Entry-Box-Element zu, sodass bei Auswahl von *Numeric* nur Zahlen in der Bildschirmtastatur erscheinen, bei *E-Mail* auf ein gültiges E-Mail-Format geachtet wird et cetera. Damit das Kalendersteuerelement in den verschiedenen Projekten funktionieren kann, muss ►

Saxonix Systems
So geht Software!

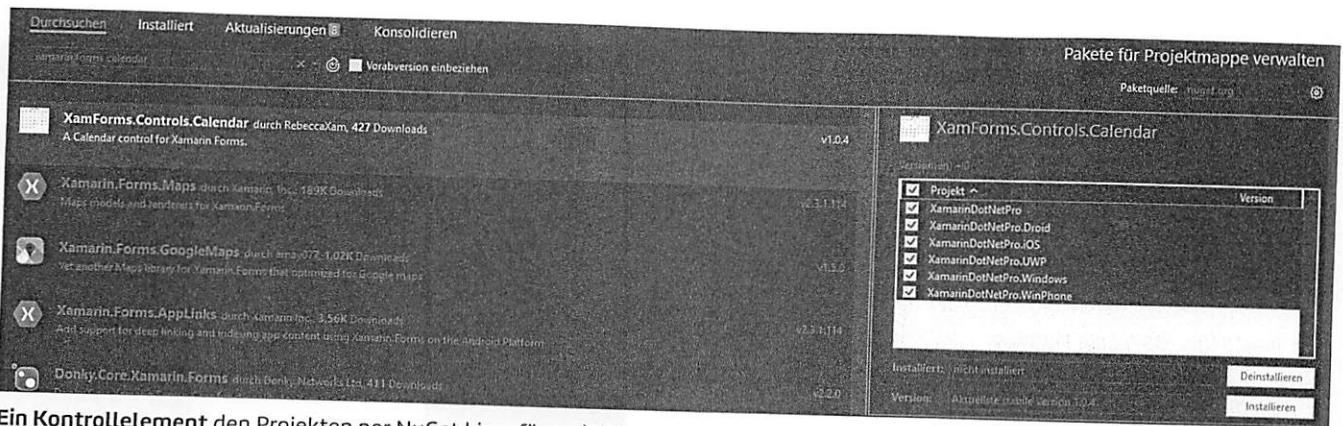
5.-7. DEZEMBER
LIVE-SESSIONS
AUF DER DDC
IN KÖLN!

ASP.NET CORE - MODULE STATT MONOLITH

So geht Web mit .NET.

Langjährige, ganzheitliche Erfahrung...
Wir beraten, entwickeln, schulen
und migrieren/portieren im
.NET-Umfeld.

sogehtsoftware.de/aspdotnetcore



Ein Kontrollelement den Projekten per NuGet hinzufügen (Bild 13)

noch eine Initialisierungszeile in den jeweiligen Einstiegscode eingefügt werden. Beim Android-Projekt ist dies die *MainActivity.cs*. Die Zeile

```
XamForms.Controls.Droid.Calendar.Init();
```

muss direkt hinter die *Forms.Init*. Bei iOS ist dies die *AppDelegate.cs* und bei UWP die *App.xaml.cs*.

Databinding und MVVM

Im Beispiel wurde mit Code-behind und *x:Name* gearbeitet. Auch kann die Oberfläche komplett oder teilweise im Code erzeugt werden, was gerade bei Android-Entwicklern sehr beliebt zu sein scheint.

Selbstverständlich unterstützt Xamarin.Forms auch Databinding und das Model-View-ViewModel-Muster. Neben ei-

genen Umsetzungen stehen auch MVVM-Frameworks wie Prism und MVVM Light zur Verfügung und sorgen für noch mehr Transparenz und Wiederverwendbarkeit.

Fazit

Xamarin.Forms erleichtert die Entwicklung plattformübergreifender nativer Apps enorm. Dank der einheitlichen Programmiersprache, der gleichen Umgebung und der XAML-Unterstützung können im günstigsten Fall 100 Prozent des Codes für alle Plattformen verwendet werden. Nichtsdestotrotz können spezifische Anpassungen pro Plattform integriert werden.

Derzeit muss der Entwickler jedoch noch einige Kompromisse eingehen, denn so richtig rund läuft die Sache noch nicht in allen Bereichen. Updates von NuGet-Paketen führen in der Regel zu Problemen, die man manuell umgehen muss,

● Listing 2: Beispiel zu Calendar Control

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamarinDotNetPro"
    xmlns:controls="clr-namespace:XamForms.Controls;assembly=XamForms.Controls.Calendar"
    x:Class="XamarinDotNetPro.MainPage">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="10" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Label Grid.Row="1"
            Text="Welcome to Xamarin Forms!"
            VerticalOptions="Center"
            HorizontalOptions="Center"/>
        <controls:Calendar Grid.Row="2" x:Name="xCalendar"
            StartDay="Monday" SelectedBorderWidth="4"
            DisabledBorderColor="Black"
            DateClicked="OnDateClick"/>
        <Entry Grid.Row="3" x:Name="xEntry"
            Placeholder="Tippen Sie irgendetwas ein" />
        <Picker Grid.Row="4" Title="Tastaturtyp"
            SelectedIndexChanged=
                "OnPickerSelectedIndexChanged">
            <Picker.Items>
                <x:String>Default</x:String>
                <x:String>Text</x:String>
                <x:String>Chat</x:String>
                <x:String>Url</x:String>
                <x:String>Email</x:String>
                <x:String>Telephone</x:String>
                <x:String>Numeric</x:String>
            </Picker.Items>
        </Picker>
    </Grid>
</ContentPage>
```

● Listing 3: Verarbeitung der Ereignisse

```

using System;
using System.Collections.Generic;
using System.Reflection;
using Xamarin.Forms;
using XamForms.Controls;

namespace XamarinDotNetPro
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
            xCalendar.SpecialDates = new List<SpecialDate>{
                new SpecialDate(DateTime.Now.AddDays(4)) {
                    BackgroundColor = Color.Red,
                    TextColor = Color.Accent,
                    BorderColor = Color.Maroon, BorderWidth=8
                },
                new SpecialDate(DateTime.Now.AddDays(6)) {
                    BackgroundColor = Color.Green,
                    TextColor = Color.Blue, Selectable = true
                }
            };
            xCalendar.MinDate = DateTime.Now.AddDays(-1);
        }

        async void OnDateClick(object sender, EventArgs e){
            DateTimeEventArgs _auswahl =
                (DateTimeEventArgs)e;
            await App.Current.MainPage.DisplayAlert(
                "Kalenderinfo", "Hier kommen die Daten vom " +
                auswahl.DateTime.Date.ToString(), "Alles klar"
            );
        }

        void OnPickerSelectedIndexChanged(object sender,
            EventArgs args)
        {
            if (xEntry == null) return;
            Picker picker = (Picker)sender;
            int selectedIndex = picker.SelectedIndex;
            if (selectedIndex == -1) return;
            string selectedItem = picker.Items[selectedIndex];
            PropertyInfo propertyInfo = typeof(Keyboard)
                .GetRuntimeProperty(selectedItem);
            xEntry.Keyboard = (Keyboard)propertyInfo
                .GetValue(null);
        }
    }
}

```



Das Date-Picker-Kontrollelement auf allen drei Emulatoren
(Bild 14)

genau wie der fehlende XAML-Designer oder der noch nicht richtig funktionierende iOS-Simulator für Windows.

Auch tauchen während der Entwicklung immer wieder Fehlermeldungen auf, die beim Umwandeln jedoch keine Rolle spielen und wieder verschwinden. Intellisense funktioniert nicht immer richtig und anderes mehr. Hier fehlt definitiv noch der Feinschliff.

Jedoch bekommt man auch sehr viel, denn der Aufwand, sich in die einzelnen Plattformen mit deren spezifischen Programmiersprachen, Entwicklungs-Tools und Frameworks einzuarbeiten, ist ungemein größer.

Durch die Übernahme Xamarins durch Microsoft dürfen die genannten Probleme bald der Vergangenheit angehören, denn für Microsoft ist die Technologie von Xamarin die große Hoffnung, im mobilen Bereich nicht ganz auf der Strecke zu bleiben. ■

- [1] <https://developer.xamarin.com/guides>
- [2] E-Book *Creating Mobile Apps with Xamarin.Forms*,
www.dotnetpro.de/SL1611XamarinPraxis1
- [3] <http://forums.xamarin.com>
- [4] www.dotnetpro.de/SL1611XamarinPraxis2
- [5] www.dotnetpro.de/SL1611XamarinPraxis3
- [6] www.dotnetpro.de/SL1611XamarinPraxis4
- [7] www.xamarin.com/prebuilt
- [8] www.xamarin.com/download
- [9] www.dotnetpro.de/SL1611XamarinPraxis5
- [10] www.dotnetpro.de/SL1611XamarinPraxis6
- [11] www.dotnetpro.de/SL1611XamarinPraxis7



Markus A. Litters

arbeitet seit 1990 mit IBM-Midrange-Systemen und seit 2005 mit dem Microsoft .NET Framework. Aktuell verlagert sich sein Schwerpunkt hin zu Universal Apps, die mit den verschiedenen Backend-Systemen integriert sein sollen.

mlitters@insassl.de

dnpCode

A1611XamarinPraxis



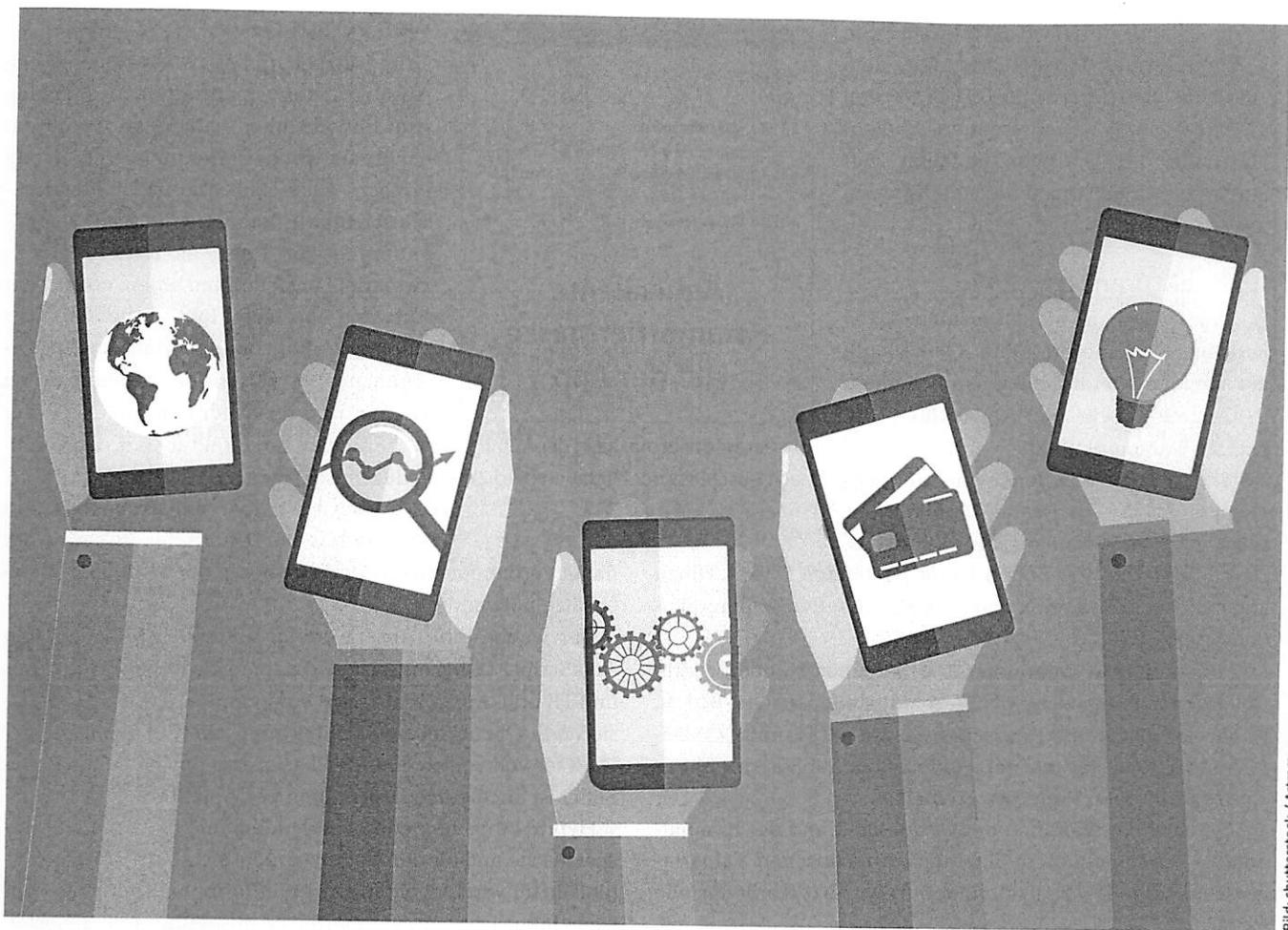


Bild: shutterstock / Attram

XAMARIN IM PRODUKTIVEINSATZ, TEIL 1

Mit .NET zu den Apps

Das Ziel: Mobile Apps mit größtmöglicher gemeinsamer Codebasis. **Der Weg:** Xamarin.

Kunden wünschen sich immer häufiger mobile Apps – auch für bereits existierende, klassische, Windows-basierte Unternehmenssoftware. Dieser Wunsch ist für das Entwicklerteam mit einigen Herausforderungen verbunden. Diese sind unter anderem die Unterstützung mehrerer Plattformen wie zum Beispiel iOS, Android und Windows, größtmögliche Wiederverwendung der existierenden Codebasis und Funktionalität zur Reduzierung von Entwicklungs- und Testaufwänden, Verwendung der geeigneten Entwicklungstechnologien sowie Anpassung der Entwicklungsinfrastruktur und -prozesse. Dieser Artikel gibt einen Einblick in die Erfahrungen des wetcon-Entwicklerteams bei der Migration von einer Windows-basierten Prozess- und Industrieautomatisierungssoftware hin zu iOS- und Android-Apps.

FDT-Komponentenstandard

In der Prozess- und Factory-Automatisierung wird Sensorik von unterschiedlichen Herstellern und unter Verwendung

unterschiedlichster Bustechnologien wie Hart, Profibus und CAN in Produktionsanlagen eingesetzt.

Der IDE-Standard FDT (Field Device Tool) ermöglicht dabei die Integration von gerätespezifischer Logik in die Engineering-Systeme der jeweiligen Systemanbieter.

Dazu stellt ein Gerätehersteller einen DTM (Device Type Manager) zur Verfügung, der es ermöglicht, Feldgeräte (wie Stellglieder, Ventile oder Messumformer) im Engineering-System beziehungsweise Servicetool (siehe Bild 1) des Systemherstellers zu visualisieren, zu parametrieren und zu überwachen. FDT definiert dazu ein Komponentenmodell und entsprechende Schnittstellen für Windows-Desktop-Betriebssysteme.

Intelligente Feldgeräte verfügen dabei zusätzlich über eigene Displays zur Überwachung und Konfiguration. Diese sollen nun durch Apps ersetzt werden, die die in DTMs implementierte Gerätefunktionalität wie Parametriermasken oder Messwertdarstellungen beinhalten.

Vergleichen Sie dazu Bild 1 und Bild 2. Sie kommunizieren dabei direkt mit dem Feldgerät über Bluetooth. DTMs werden mithilfe von .NET programmiert. Es liegt also nahe, die Gerätefunktionalität wiederzuverwenden – in einer auf Xamarin basierenden App.

Architekturansatz

Der vom Gerätehersteller erstellte DTM implementiert die gesamte Gerätelogik mit allen Parametern, die entsprechenden UI-Controls sowie die Abwicklung der Feldbuskommunikation.

Der FDT-Host (Rahmenanwendung) implementiert die generelle Bedienerführung, den Gerätekatalog und die Instantierung der zum jeweiligen Gerät passenden DTM. Hinzu kommen noch Persistenz und Management der Kommunikations-Topologie.

DTM und Host interagieren dabei über standardisierte COM-beziehungsweise .NET-Schnittstellen (siehe Bild 3). Dabei differenziert FDT zwischen Geräte-DTM und COMM-DTM zur Realisierung der Feldbuskommunikation mittels eines spezifischen Windows-Treibers.

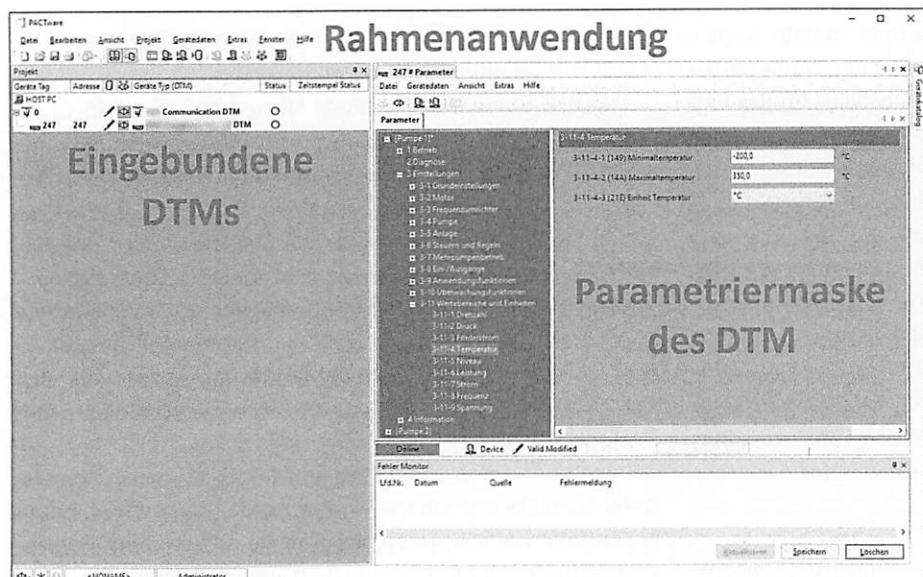
Zur Wiederverwendung der Gerätelogik und der Kommunikationsfunktionalität in einer gerätespezifischen Xamarin-App muss die FDT-Laufzeitumgebung ersetzt werden (siehe Bild 4).

Ein zu FDT analoges Komponentenmodell mit dynamischem Laden von gerätespezifischer Funktionalität scheidet aus, da dies bei iOS generell unterbunden wird.

Stattdessen wird ein App-Generator verwendet, der aus der Gerätebeschreibung, die typischerweise in Form von XML oder EDD (Electronic Device Description) vorliegt, eine gerätespezifische App generiert.



Die Parametriermaske in der App nach Abschluss der Migration (Bild 2)



Rahmenanwendung und DTM-Parametriermaske (Bild 1)

Web versus nativ

Grundsätzlich gibt es zwei technologische Ansätze der plattformübergreifenden App-Entwicklung. Anfänglich wurde zunächst ein webbasiertes und vereinfachtes Prototyp entwickelt. Die nachfolgend beschriebenen Erkenntnisse dienen als Orientierung zur Entscheidungsfindung in zweierlei Hinsicht: Zum einen, ob sich der Aufwand überhaupt lohnt, einen Prototyp zu entwickeln, zum anderen, um den geeigneten Lösungsansatz bestimmen zu können und das Migrationspotenzial der vorhandenen Architektur und des Quellcodes einschätzen zu können.

Zunächst ist aber ein grundsätzliches Verständnis der Unterschiede sowie der damit verbundenen Vor- und Nachteile der beiden Lösungsansätze notwendig.

Der webbasierte Ansatz erfolgt mittels HTML, CSS und JavaScript. Dabei unterscheidet man zwischen Web-Apps und Hybrid-Apps. Web-Apps sind von Natur aus HTTP-basierende Client-Server-Architekturen, die der Benutzer per Browser von verschiedenen Plattformen aus bedienen kann. Somit ist auch keine Distribution per App Store notwendig.

Hybrid-Apps hingegen verwenden eine native, plattformspezifische und für den Benutzer nicht sichtbare Browser-Engine. In ihr wird der eigentliche plattformübergreifende App-Code inklusive Benutzeroberfläche ausgeführt und gehostet. Die Engine wird samt webbasiertem Quellcode in ein natives App-Paket geschnürt und in dem jeweiligen App Store bereitgestellt.

Möglich macht dies das PhoneGap-Framework (auch bekannt als Apache Cordova). Im Unterschied zu Web-Apps ermöglicht die Lösung mit PhoneGap den Zugriff per JavaScript-Schnittstelle auf Gerätefunktionalitäten wie zum Beispiel Kamera oder GPS-Sensor.

Eine Wiederverwendung von bestehendem C#-Quellcode ist hierbei natürlich nicht möglich. Um existierende Programmfunktionalität in einer Hybrid- oder Web-App nutzen zu können, müsste diese per Server und Webservice zur Verfügung gestellt werden.

Hierbei sind weitere Herausforderungen wie Offline-Verhalten und -Datenhaltung zu meistern. Wenn die Daten dann noch in einem gewissen Format vorgegeben sind, wie beispielsweise die zugrunde liegenden Gerätebeschreibungen, dann geraten browserbasierte Apps an ihre Grenzen.

Diese zu überwinden erfordert dann einen entsprechend großen Aufwand. ►

In der Regel sehen die Benutzeroberflächen web-basierter Lösungen auf allen Plattformen gleich aus. Dies kann von Vorteil sein, wenn man sich bewusst für ein einheitliches Aussehen auf allen Plattformen entscheidet. Im Gegenzug kann man damit aber nicht allen Richtlinien zur Gestaltung der UIs aller Plattformhersteller gerecht werden.

Der native Lösungsansatz der plattformübergreifenden Entwicklung mit Xamarin erfolgt unter Zuhilfenahme von C#, Visual Studio und großen Teilen des .NET-Solution-Stacks. Zusätzlich werden im Hintergrund die spezifischen SDKs und Werkzeuge der jeweiligen Plattformhersteller verwendet, um aus C#-Quellcode ausführbare, native und binäre App-Images als Ergebnis zu erhalten.

Das bedeutet, dass am Ende ein App-Store-fähiges App-Paket vorliegt, so, als hätte man dies ausschließlich mit den plattformspezifischen Werkzeugen von Apple für iOS oder von Google für Android erzeugt. Die mit Xamarin entwickelten Apps können somit das von der jeweiligen Plattform zur Verfügung gestellte Spektrum an grafischen Bedienelementen, Systemdiensten und Hardware-Ressourcen voll ausschöpfen.

Dadurch lassen sich Benutzeroberflächen entwickeln, die den Programmierrichtlinien und Konventionen des Plattformherstellers entsprechen. Es sind aber auch eigene Anpassungen der UI-Elemente möglich, um beispielsweise ein Corporate Design zu implementieren. Hierfür heißt das Stichwort Custom Renderer, wenn man die eigene Anwendung mit dem UI-Framework Xamarin.Forms entwickelt.

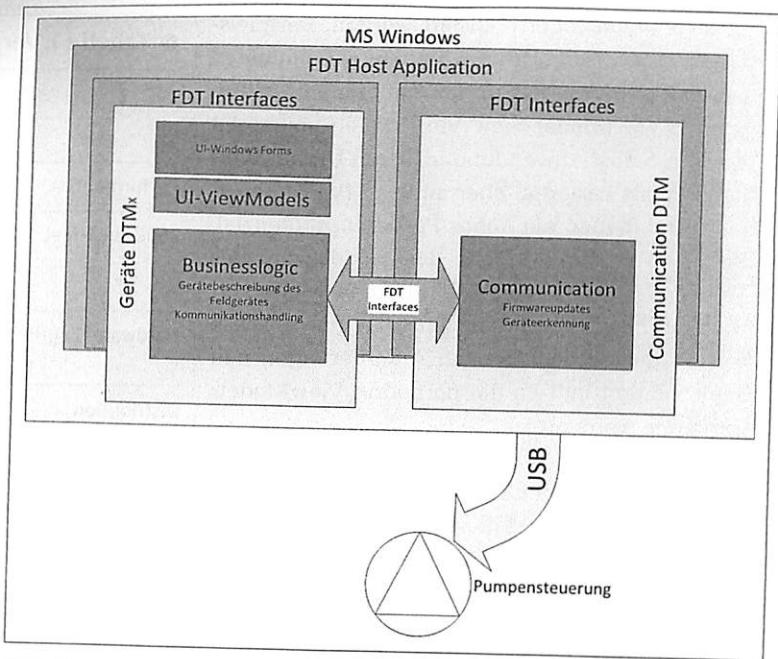
Eine bestehende C#-Codebasis kann entweder in die mobilen C#-Entwicklungsprojekte eingebunden oder als PCL (Portable Class Library) in eine wiederverwendbare Zwischensprache kompiliert und als DLL referenziert werden. Existierender Code für Benutzeroberflächen und Windows-spezifischer Programmcode, wie zum Beispiel UserControls oder Registry-Zugriffe, sind nicht portierbar.

Nach diesem groben Technologieüberblick (siehe auch Tabelle 1) und aus den Erkenntnissen des auf PhoneGap basierenden Prototyps können folgende Schlüsse gezogen werden: In Sachen Bedienbarkeit und Performance konnte das gewünschte native Look-and-feel der Bedienelemente nicht erzielt werden.

Die App vermittelt dem Benutzer dadurch ein Gefühl von fehlender Wertigkeit und Qualität. Aber genau diese Eigenschaften sind Benutzer von Unternehmenssoftware gewohnt und erwarten diese auch von ihr.

Wie bereits erwähnt, ist ein weiteres Problem der Weblösung die Einbindung der existierenden Gerätebeschreibungen aufgrund deren Größe und Formats. Dies ist in webbasierten Apps nur schwer oder kaum umsetzbar. Möglicherweise ändern sich die genannten Probleme in unbestimmter Zukunft, da Webtechnologien stetig weiterentwickelt und verbessert werden.

Für das angestrebte Projekt und dessen Anforderungen war dies aber keine Option. Der native Ansatz von Xamarin



FDT-Komponentenmodell und gerätespezifische Funktionalität (Bild 3)

erfüllt ein höheres Maß der Qualitätsansprüche und der Anforderungen des Kunden. Er ermöglicht nicht nur eine hohe Wiederverwendung funktionierenden und getesteten Codes, sondern auch die vom Windows-basierten Servicetool bekannten Offline-Use-Cases und deren Dateneingabeformat.

Migrationspotenzial analysieren

Um den möglichen Funktionsumfang der mobilen Anwendung, den Projektverlauf und dessen Aufwände besser abschätzen zu können, ist eine Analyse folgender Punkte notwendig:

- Welche Anwendungsfälle und Anforderungen der bestehenden Software sind zu portieren? Sind diese überhaupt portierbar?
- Welche neuen Anwendungsfälle und Anforderungen gibt es?
- Prüfung der bestehenden Architektur der Anwendung.
- Prüfung der Portierbarkeit des bestehenden Codes mit Xamarin .NET Mobility Scanner.

Grundlage der Analyse ist, die Wünsche und Erwartungen des Kunden an die zukünftige App so gut wie möglich zu kennen. Je klarer hier das Bild ist, umso genauer lassen sich die Anwendungsteile bestimmen, die portiert werden müssen. Sollten manche Anwendungsfälle für die mobile Welt erst gar nicht relevant sein und können die betroffenen Komponenten oder Codeteile von der Migration ausgeschlossen werden, so kann dies die Aufgabe erleichtern und mögliche Aufwände ersparen.

Abhängig ist dies wiederum von der Architektur der bestehenden Anwendung. Von Vorteil ist es, wenn diese in entsprechend lose gekoppelte Komponenten aufgeteilt ist. Eventuell notwendige Ersetzungen oder Erweiterungen verschiedener Anwendungsteile, die eine Portierung mit sich bringt,

können so einfacher realisiert werden. Beispielsweise lassen sich UI-relevante Funktionalitäten sehr gut portieren, wenn diese mit einem Pattern wie MVVM (Model-View-ViewModel) umgesetzt wurden. Selbst Anwendungen, deren UIs auf Windows Forms basieren, aber mittels MVVM konzipiert sind, haben ein hohes Portierungspotenzial. So lassen sich große Teile des Models und des ViewModels auf anderen Plattformen wiederverwenden. Nicht portierbar sind die Views. Diese müssen gemäß den mobilen Anforderungen neu implementiert und an die portierten ViewModels gebunden werden.

Nun ist noch zu prüfen, inwieweit der Code zu Xamarins Plattformen für iOS oder Android kompatibel ist. Das Ausmaß dieser Kompatibilität hängt davon ab, welche Base-Class-Library-Typen oder Drittanbieter-Bibliotheken der Code verwendet.

Dies eigenhändig ohne entsprechendes Werkzeug zu analysieren wäre sehr aufwendig. Aus diesem Grund bietet Xamarin unter [2] die dafür benötigte Unterstützung an. Dort können Sie die vorhandenen .NET-Assemblies (DLL- oder EXE-Dateien) prüfen lassen, inwieweit der darin enthaltene Code von den ausgewählten mobilen Plattformen unterstützt wird. Als Ergebnis des Scans erhalten Sie einen Bericht (siehe Bild 5), der aufzeigt, wie viel Prozent des Codes auf den jeweiligen Plattformen lauffähig sind, ohne Modifikationen vornehmen zu müssen. Im Detail erhält man auch Informationen darüber, welche Methoden auf welcher Plattform unterstützt werden. Dabei basiert der Bericht auf einer technischen Analyse der Cross-Plattform-Kompatibilität der verwendeten Typen und Methoden.

Die Analyse der desktopbasierten Assemblies, die weitestgehend Businesslogik enthalten, ergab 98 Prozent Code-

• Tabelle 1: Vergleich von webbasierten Apps und Xamarin-Apps

	Hybrid-/Web-App	Native Xamarin-App
UI	Rendering durch Browser	Rendering nativ
Performance	Eingeschränkt	Nativ
Look-and-feel	Maximal flexible Gestaltung, Natives Look-and-feel beschränkt nachahmbar	Nativ, anpassbar
Hardware-Zugriff	Ja (evtl. mit Einschränkungen bei Web-Apps)	Ja
Distribution	Web / App-Store	App-Store
Offline-fähig/ Offline-Datenhaltung	Beschränkt	Ja
Programmierung	HTML5, JavaScript, CSS	C#
Referenzierung existierenden Codes	Implizit per Web-Service-Architektur	Ja

Kompatibilität zu Xamarins iOS- und Android-Plattformen. Die anfängliche Skepsis gegenüber diesem sehr hohen Wert konnte nach ersten Kompilierungsversuchen und der letztendlichen Portierung widerlegt werden.

Entwicklungswelt von Xamarin

Möchte man mit Xamarin Apps für Android entwickeln, so hat man die Wahl, ob man dazu einen Windows-Rechner oder einen Mac verwendet. Bei der Entwicklung mit Xamarin für iOS benötigt man hingegen unbedingt Hard- und Software von Apple.

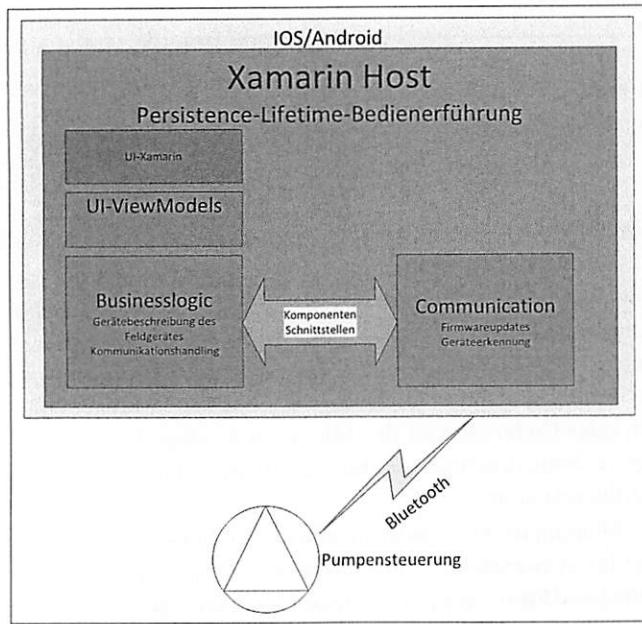
Dennoch ist es mit Xamarin möglich, mit einem Windows-System für iOS zu entwickeln. Hierfür muss sich ein Mac-Rechner (zum Beispiel Mac mini oder MacBook) mit Mac OS X, Xcode (Entwicklungsumgebung von Apple) und Xamarin-Installation im lokalen Netzwerk des Windows-Entwicklungsrechners befinden. Der Mac wird benötigt, um den iOS-spezifischen, binären Maschinencode und das iOS App Store Package (IPA) zu erzeugen.

Diesen Vorgang kann der Entwickler von Visual Studio aus anstoßen, den Rest erledigen die Xamarin- und Apple-Tools im Hintergrund. Als Entwicklungsumgebung dient Visual Studio auf einem Windows-System, das entweder mit einem physikalischen Rechner oder mit einer virtuellen Windows-Maschine auf dem Mac betrieben wird.

Des Weiteren kann man die App mit Apples iOS-Simulator oder mit einem am Mac per USB angeschlossenen Gerät, wie zum Beispiel dem iPhone, testen und debuggen. Siehe dazu mehr weiter unten.

Alternativ zu obigem Einsatzszenario ist die gesamte Entwicklung auch ohne Windows-Umgebung möglich. Hierfür ist die von Xamarin eigens entwickelte IDE Xamarin Studio für Mac OS vorgesehen. Damit kann man sowohl für Android als auch für iOS entwickeln.

Die Android-Entwicklung kann ebenfalls mit Visual Studio unter Windows oder Xamarin Studio unter Mac OS erfolgt.



Wiederverwendung der DTM-Geräetelogik in einer Xamarin-App
(Bild 4)

gen. Die Xamarin- beziehungsweise Visual-Studio-Installation stellt auch hier im Hintergrund alle benötigten Tools und SDKs für Android bereit. Wird ein Build angestoßen, so ist das Ergebnis ein Android Application Package (APK), das unter anderem den C#-Code als IL-Kompilat und eine .NET-basierte Laufzeitumgebung (Mono) enthält.

Hier lässt sich die App ebenfalls mit angeschlossenen Android-Geräten oder -Simulatoren debuggen. Auch hierzu später mehr.

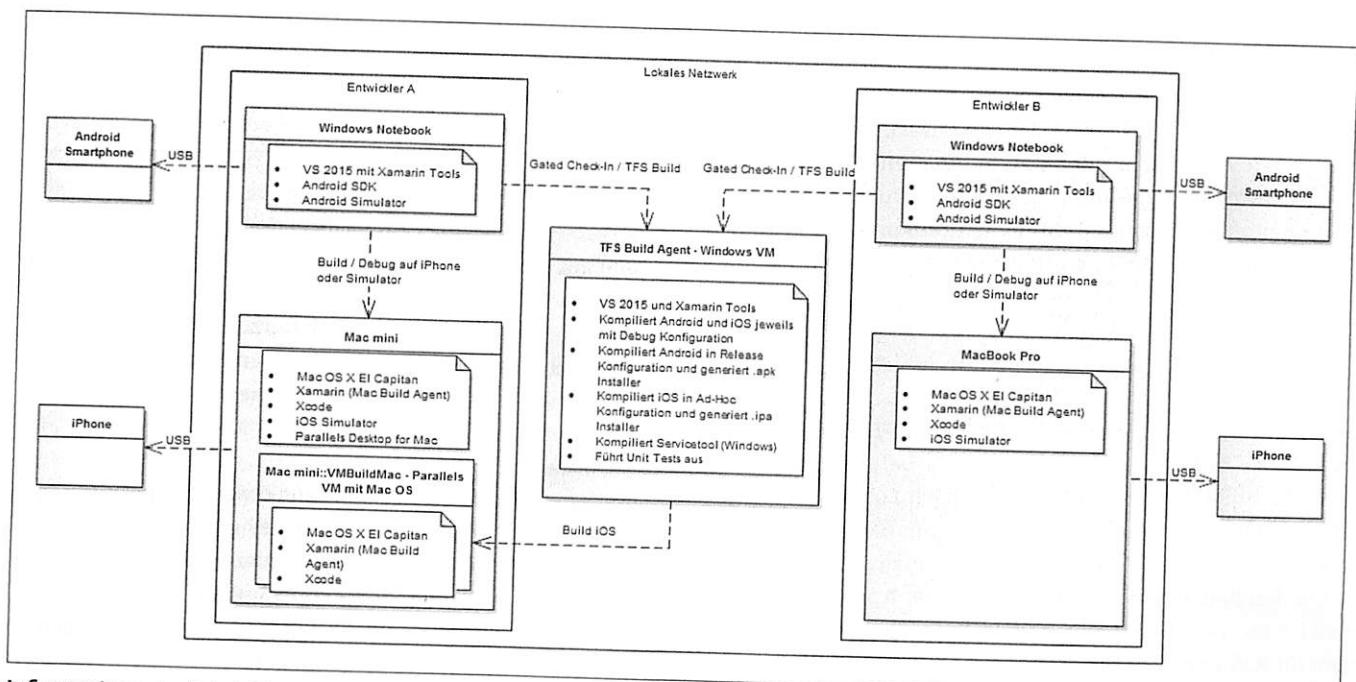
Entwicklungs- und Build-Management

Ursprünglich entwickelte wetcon klassische Windows-Desktop-Anwendungen. Dementsprechend setzen die bisherige IT-Infrastruktur und die verwendeten Entwicklungstechnologien zu großen Teilen auf Microsoft-Systemen und Anwendungen auf. Dazu zählen:

- Windows Server 2008 R2 mit TFS 2010 SP1 Installation,
- vorhandene Team Project Collection mit angepasstem XAML-Buildprocess-Template und angepassten Work Item Templates (Bug, Build Info),
- Builddefinitionen für Continuous Integration mit Gated Check-In, sowie Build und Deployment der bisherigen Produkte,
- verschiedene Entwicklersysteme unter Windows 7, 8 und 10 mit Visual Studio 2015 Update 3.

Der angestrebte Entwicklungsprozess beziehungsweise die Infrastruktur mit Xamarin ist zum Teil in Bild 6 veranschaulicht und sieht unter Berücksichtigung der genannten Ausgangslage wie folgt aus:

- Parallele Entwicklung und TFS Continuous Integration (CI) Builds: Mehrere Entwickler können gleichzeitig und unabhängig voneinander iOS- und Android-Apps kompilieren und auf entsprechenden Smartphones debuggen.



Infrastruktur der Entwicklungsumgebung mit Xamarin (Bild 6)

Scan Results

98% of your code is ready for mobilization!
Taking this code mobile will save 5 months, 1 week, 1 day of platform-specific development effort.



Beispielbericht des Xamarin .NET Mobility Scanners (Bild 5)

- Wiederverwendung des vorhandenen TFS mit angepassten Work Items, selbst entwickelten Build Process Templates und Build Tools (unter anderem wegen automatischer Versionierung, Release Notes und Branch-Erstellung),
- CI-Build und Unit-Test-Ausführung der mobilen Xamarin-Projekte und der bisherigen Windows-Desktop-Projekte,
- Deployment-Builds neuer Testversionen, jeweils für iOS und Android, mit Auslieferung per Microsoft HockeyApp.

Um parallele iOS-Entwicklung zu ermöglichen, verwendet jeder Entwickler einen eigenen physikalischen Mac. Separate Macs sind aus Gründen der räumlichen Trennung der einzelnen Entwickler und deren Gerätetests notwendig. Der auf dem Mac installierte Xamarin-Build-Agent erlaubt zwar

mehrere Verbindungen unterschiedlicher Benutzer beziehungsweise Visual-Studio-Instanzen, diese können aber nicht gleichzeitig genutzt werden.

Deshalb ist auch eine weitere Maschine mit Mac OS für TFS-Builds notwendig. Hierfür ist keine separate Hardware nötig, eine virtuelle Maschine reicht vollkommen aus, da damit keine Entwicklung, Geräte- oder Simulatortests durchgeführt werden. Xamarin for Mac genügt, eine vollständige Installation mit Xamarin Studio und Android ist nicht nötig.

Des Weiteren ist auf diesem virtuellen Rechner Mac OS und Xcode installiert. Die virtuelle Maschine befindet sich auf einem der Entwickler-Macs, der dauerhaft im Netzwerk betrieben wird und somit für CI-, Nightly- und Deployment-Builds immer zur Verfügung steht.

Um TFS-Builds einzusetzen zu können, sind folgende Voraussetzungen zu erfüllen:

- Visual Studio 2015 muss auf TFS Build Server installiert sein.
- Xamarin muss sowohl auf TFS Build Server als auch auf Mac Build VM installiert sein.
- Initial muss einmal eine Verbindung mit Visual Studio des TFS Build Servers und dem Xamarin Build Agent auf Mac hergestellt werden.
- Verbindungsparameter des Xamarin Mac Agents sind in der Builddefinition anzugeben.

Damit der TFS-Build-Vorgang auch die Build-Tools von Visual Studio 2015 und unter anderem aktuelle Compiler-Features verwendet und Shared Projects interpretieren kann, muss das TFS 2010 Build Process Template angepasst werden, sodass die MSBuild-Activities auf die MSBuild.EXE von Visual Studio 2015 verweisen. Verwendet man TFS 2015, ist dies nicht notwendig.

Des Weiteren muss Visual Studio bei der Einrichtung der Verbindung zum Xamarin Build Agent mit dem Benutzer ausgeführt werden, der auch zur Ausführung der TFS-Builds verwendet wird.

Die CI-Build-Definition ist wie folgt konfiguriert: Sie übersetzt und testet die bisherige Windows-Desktop-Solution und die mobile Solution mit den darin enthaltenen Projekten für iOS, Android und plattformunabhängige Unit-Tests. Wichtig ist die Angabe folgender MSBuild-Argumente, die so aktuell bei Xamarin nicht dokumentiert sind:

```
/p:"ContinueOnDisconnected=false" /p:"ServerAddress= myBuildMacIpOrDNS" /p:"ServerUser=myBuildMacUserName".
```

Fehlen diese Argumente, so lässt der Build-Bericht glauben, dass der iOS-Build erfolgreich war. Es befinden sich zwar DLLs im Build-Ausgabeordner, jedoch wird ohne diese Argumente keine Verbindung zum Mac aufgebaut und das gewünschte iOS App Store Package nicht erzeugt.

Für den Bau und die Auslieferung neuer App-Versionen an den Kunden existieren jeweils zwei separate Build-Definitionen für iOS und Android, die ähnlich wie die CI-Build-Definition konfiguriert sind. Die resultierenden App-Installationspakete werden per HockeyApp distribuiert. Dies ist ein von

Microsoft bereitgestellter Dienst (siehe [3]), um Apps verschiedenster Plattformen beziehungsweise unterschiedlichen Formats (zum Beispiel IPA, APK, MSI) verteilen zu können. Der Dienst ist auf die Auslieferung von Testversionen beschränkt.

Die finalen Releases müssen wie von den Plattformherstellern vorgegeben per App Store oder Play Store zur Verfügung gestellt werden. Außerdem bietet der Dienst das Verwalten von Testteams, Bereitstellung von Nutzerstatistiken und Fehlerberichterstattung an.

Ist eine neue App-Version an den Kunden auszuliefern, wird ein Build der entsprechenden Build-Definition angestoßen. Nach erfolgreichem Build kann das Installationspaket entweder automatisiert per HockeyApp-Konsolenanwendung oder manuell per Browser in das Portal von HockeyApp hochgeladen werden. Anschließend können die Tester den Download der neuen Version über HockeyApp anstoßen.

Fazit

Neben einer vorhandenen Desktop-Anwendung sollen Apps für Mobilgeräte entstehen. Der Artikel hat dabei bislang erklärt, welche Technologien hier zum Einsatz kommen können. Die Voraussetzungen legten schließlich nahe, native Anwendungen zu bauen, die über die Plattformgrenzen hinweg möglichst viel Code teilen.

Aus dem Grund wurde die Xamarin-Lösung präferiert. Geklärt wurde außerdem das Infrastrukturumfeld, das neben der Desktop-Anwendung auch für die Entwicklung der Mobilanwendungen dienen soll.

Im zweiten Teil des Artikels wird es um die Implementierung gehen. Dabei ist erst einmal eine wichtige Entscheidung zu fällen: Soll der gemeinsame Code über die sogenannten Portable Class Libraries oder über Shared Projects verwaltet werden? ■

[1] www.fdtgroup.org

[2] <https://scan.xamarin.com>

[3] www.hockeyapp.net



Christian Fahrenholz

ist Dipl. Informatiker (FH) und arbeitet als Software Architect bei der wetcon GmbH in Senden bei Neu-Ulm. Er ist dort spezialisiert auf die Konzeption und Umsetzung von Desktop- und Cross-Plattform-Apps im .NET-Umfeld.
cfahrenholz@wetcon.net



Mathias Hartner

arbeitet als Softwareentwickler für die wetcon GmbH in Senden. Seine Aufgaben umfassen Konzeption und Umsetzung von .NET-Desktop-Apps und die Cross-Plattform-Entwicklung für Android und iOS mithilfe von Xamarin.
mhartner@wetcon.net

BASISKENNTNISSE ZU ANDROID-APPS

Von Xamarin beflügelt

Mit Visual Studio können Entwickler nun Apps für Googles Betriebssystem entwickeln.

Als Scott Guthrie am 31. März 2016 auf der Bühne der Microsoft Build 2016-Konferenz sprach, hätte die Reaktion der Teilnehmer kaum begeisterter ausfallen können. Schließlich kündigte Guthrie die freie Verfügbarkeit der Xamarin-Plattform für alle Inhaber einer gültigen Visual-Studio-2015-Lizenz an und erfüllte somit einen der größten Wünsche vieler .NET-Entwickler.

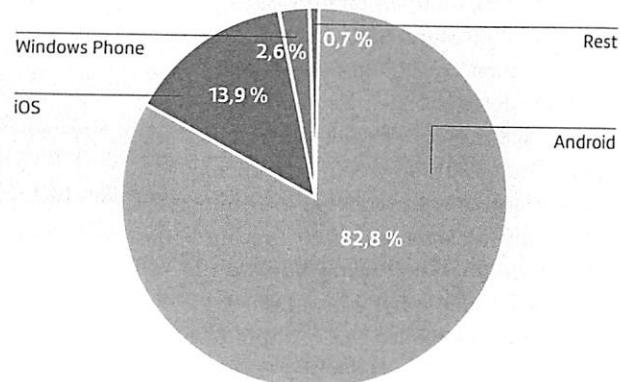
Mit der Ankündigung der kostenlosen Werkzeuge fiel für viele Programmierer der Startschuss zur plattformübergreifenden App-Entwicklung. Da für die Entwicklung von iOS-Apps jedoch zusätzlich zur Xamarin-Plattform und Visual Studio ein Mac nötig ist, wenden sich die meisten Entwickler zunächst Android zu, da sich Apps für Googles mobiles Betriebssystem vollständig unter Windows entwickeln lassen.

Doch nur, weil zur Entwicklung von Android-Apps eine bekannte Entwicklungsumgebung (Visual Studio), eine bekannte Programmiersprache (C#), eine bekannte Bibliothek (die .NET-Klassenbibliothek) und ein bekanntes Betriebssystem (Windows) zur Verfügung stehen, bedeutet dies nicht, dass es sich um einen hürdenlosen Einstieg handelt. Im Gegenteil. So vertraut die Umgebung auch wirken mag, Android und die Entwicklung für Android haben einige Eigenheiten, die dieser Artikel enträtseln soll.

Android-Versionen

Ein Blick auf die Anteile am Smartphone-Markt verrät, dass Android das mit Abstand verbreitetste mobile Betriebssystem

Smartphone-Marktanteile



Zweites Quartal 2015 (Bild 1)

dotnetpro 11/16

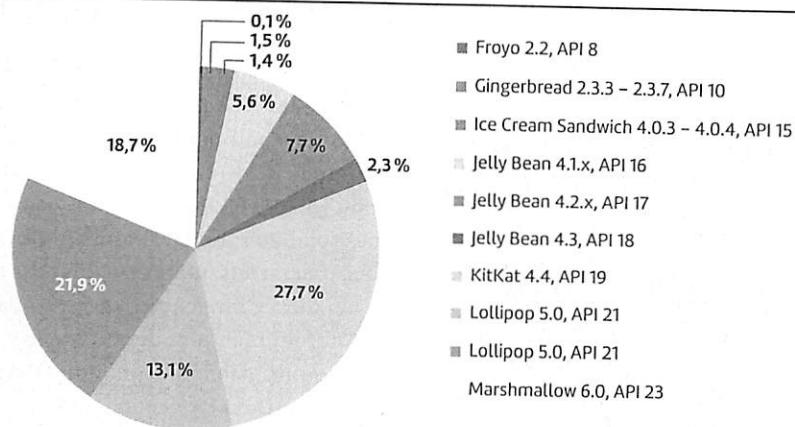
Quelle: IDC

ist. Seit der Einführung der ersten Geräte Ende 2008 konnte das auf Linux basierende, quelloffene Betriebssystem seinen Anteil stetig ausbauen. Laut einer Studie von IDC betrug der Marktanteil von Android im zweiten Quartal 2015 82,8 Prozent, wie Bild 1 belegt [1].

Zu den Erfolgsfaktoren gehört sicherlich, dass Endgeräte mit dem Android-System nicht nur durch Google selbst oder durch Mitglieder der von Google gegründeten Open Handset Alliance, die offiziell den Android Quellcode entwickelt, hergestellt werden, sondern auch von anderen Unternehmen. Jeder dieser Hersteller bestimmt selbst, welche der mittlerweile elf Hauptversionen von Android er mit welchen Anpassungen ausliefert und vor allem auch, ob, wann und wie lange er Betriebssystem-Aktualisierungen für sein Gerät bereitstellt.

Diese Art der Selbstbestimmung führt dazu, dass sich der Android-Markt mittlerweile recht stark fragmentiert hat. So besaßen laut Android Developer Dashboard im August 2016 nur 15,2 Prozent aller aktiven Geräte die derzeit aktuelle Version 6.0 von Android [2], die bereits ein Jahr zuvor, nämlich im August 2015, erschien. Knapp die Hälfte aller Geräte läuft, wie Bild 2 zeigt, noch unter einer Version älter als 4.4, die noch vom Oktober 2013 stammt.

Aktive Android-Geräte



Betriebssystemverteilung nach Version (Stand August 2016) (Bild 2)

dotnetpro 11/16

Quelle: developer.android.com

Dass immer noch so viele alte Versionen im Einsatz sind, liegt selbstverständlich nicht an der Unwilligkeit der Anwender, ihr System zu aktualisieren, sondern daran, dass Android-Updates nicht von Google, sondern nur vom jeweiligen Gerätetersteller bezogen werden können. Diese haben in der Regel mehr Interesse daran, neue Geräte zu verkaufen, als Aktualisierungen für alte bereitzustellen. Somit sind Android-Anwender meist in der unbefriedigenden Situation, ihr Gerät nicht auf dem aktuellen Stand halten zu können und so keinen Zugriff auf neue Funktionen des Betriebssystems zu haben.

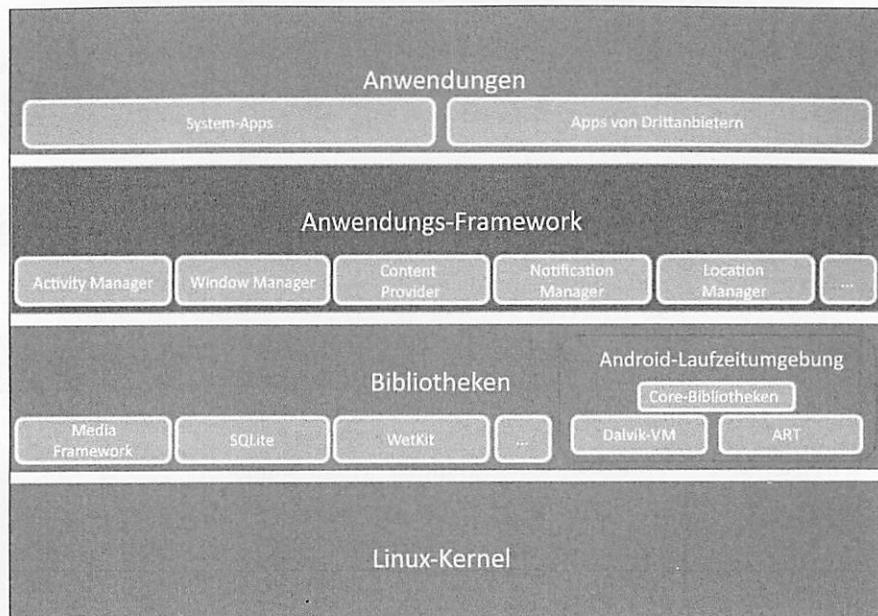
Natürlich ist dieser Umstand nicht nur für Endanwender wenig erfreulich, sondern auch für Entwickler. Wer mit seiner App beispielsweise 80 Prozent aller Android-Nutzer erreichen will, muss diese kompatibel zu Android 4.3

Jelly Bean entwickeln, einer Android-Version, die noch vom Juli 2013 stammt. Dies bedeutet, dass neuere Funktionen des Android-APIs entweder nicht genutzt werden können oder im Quellcode mithilfe von Compiler-Konstanten versionsspezifische Implementierungen gewählt werden müssen wie in den folgenden Zeilen:

```
#if __ANDROID_19__
// nur für Android >= 4.4
#endif
```

Als Alternative zu Compiler-Direktiven sind darüber hinaus auch Prüfungen zur Laufzeit möglich, wie folgender Code zeigt:

```
if (Android.OS.Build.Version.SdkInt >=
    Android.OS.BuildVersionCodes.Lollipop)
{
    builder.SetCategory(Notification.CategoryEmail);
}
```



Die Architektur von Android (angelehnt an Wikipedia) (Bild 3)

Architektur

Aus der Vogelperspektive betrachtet, verfolgt Android eine klassische Schichtenstrategie, wie es in Bild 3 zu sehen ist.

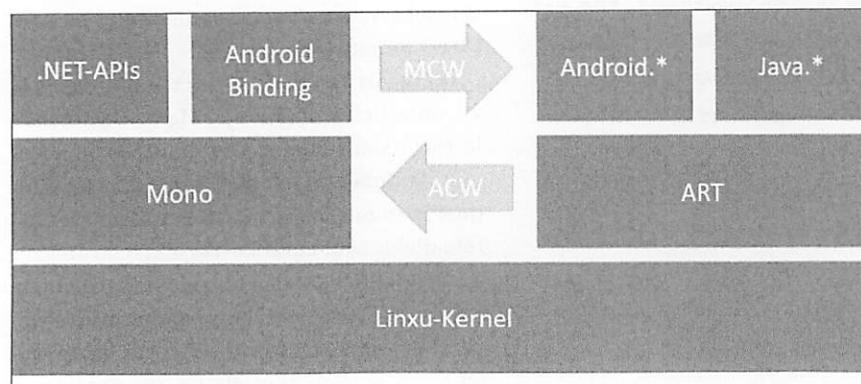
Die unterste Schicht bildet ein Linux-Kernel. Er hat die Aufgabe, mithilfe von Treibern die Interaktion der Hardware mit der darüberliegenden Schichten zu ermöglichen.

In der folgenden Schicht – Bibliotheken – befinden sich einige Kernkomponenten des Betriebssystems, wie zum Beispiel die Datenbank-Engine SQLite. Diese Komponenten sind wegen des Laufzeitverhaltens in C und C++ implementiert. Da die eigentlichen Android-Bibliotheken in Java implementiert wurden, gibt es eine dünne, Android-spezifische Wrapper-Schicht, die sich über diese C+-Komponenten legt. Somit stehen sie den darüberliegenden Schichten trotz des Technologiebruchs auf komfortable Weise zur Verfügung.

Neben den Kernkomponenten beherbergt die Bibliotheken-Schicht auch die Android-Laufzeitumgebung oder -umgebungen. Je nach Android-Version kann nämlich zwischen der Dalvik-VM (Dalvik Virtual Machine) und ART (Android Runtime) unterschieden werden.

Die Dalvik-VM ist eine an Android angepasste Java-Laufzeitumgebung und vergleichbar mit der Common Language Runtime (CLR) des .NET Framework. Genau wie die CLR setzt sie auf Just-in-Time-Kompilierung (JIT), um eine Zwischensprache auszuführen – nur dass es sich bei dieser Zwischensprache im Fall von Android nicht um die Intermediate Language (IL), sondern um den Dalvik Executable (Dex) Bytecode handelt.

Da die JIT-Kompilierung bei jedem Start einer App ausgeführt wird und sich dies negativ auf Laufzeitverhalten und Akkulaufzeit auswirkt, wurde ab An-



Die Architektur von Xamarin Android (Bild 4)

droid-Version 4.4 die neue Laufzeitumgebung ART eingefürt. Diese übersetzt den Java-Bytecode nicht mehr bei jedem Start der App, sondern nur noch einmal während der Installation. Neben einer verbesserten Performance führt diese Strategie auch zu einer längeren Akkulaufzeit und einer geringeren Wärmeentwicklung des jeweiligen Geräts.

Auf eine Xamarin-App treffen diese Änderungen jedoch nur bedingt zu, wie Bild 4 zeigt. Wie unschwer zu erkennen ist, läuft in einer Xamarin-basierten App neben der Android-Laufzeitumgebung immer noch die Mono-Laufzeitumgebung, Xamarins Android-Pendant zur .NET-CLR. Diese übersetzt zur Laufzeit über einen JIT-Compiler die in Intermediate Language vorliegenden .NET-Assemblies. Da die ART-Umgebung den IL-Code nicht zur Installationszeit übersetzen kann, kommt es also in einer Xamarin-App immer zu einer JIT-Kompilierung.

Ein Blick auf Bild 3 zeigt, dass zwei Schichten noch nicht beschrieben sind: Anwendungs-Framework und Anwendungen. Die Schicht Anwendungs-Framework enthält ein Set von APIs, die der Applikationsentwickler typischerweise direkt verwendet. Dazu gehören zum Beispiel die Android-Oberflächenelemente.

In der obersten Schicht, Anwendungen, befinden sich sowohl System-Apps als auch Apps von Drittherstellern, also solche, die zum Beispiel von Lesern der dotnetpro geschrieben werden. Diese Schicht taucht im Architekturschaubild auf,

weil Apps unter Android recht eng miteinander interagieren können. So ist es durchaus möglich, von einer Bildschirmmaske aus App A direkt in eine Maske aus App B zu springen, ohne dass dem Anwender dies bewusst ist.

Entwickeln von Apps

Zum Entwickeln von Android-Apps werden normalerweise die Programmiersprache Java und die kostenfreie Entwicklungsumgebung Android Studio genutzt. Xamarin-Entwickler kommen um beides selbstverständlich herum, da ihnen C# und Visual Studio zur Verfügung stehen. Obwohl sie durchgängig mit der gewohnten IDE und Programmiersprache arbeiten können, sind zur erfolgreichen Übersetzung von Android-Apps jedoch ein installiertes Java-SDK sowie das Android-SDK notwendig. Um die Installation der beiden SDKs muss sich der Visual-Studio-Anwender jedoch nicht selbst kümmern, da die IDE dies bereits erledigt, sobald im erweiterten Setup *Xamarin* ausgewählt wird (siehe Bild 5).

Das Android-SDK besteht aus mehreren einzelnen Werkzeugen, von denen einige optional sind. Auf jeden Fall erforderlich sind beispielweise die Android SDK Tools, die Android SDK Platform Tools und die Android SDK Build Tools. Außerdem wird mindestens für die niedrigste Android-Version, für die entwickelt werden soll, die Android SDK Platform benötigt.

Zu den optionalen Bestandteilen gehören unter anderem Emulator-Systemabbilder für die jeweiligen Android-Versionen. Diese ermöglichen den Test der Anwendung auf dem gewünschten Android im Emulator. Alternativ zum Emulator, der mit dem Android-SDK installiert wird, kann jedoch auch Microsofts kostenloser Android-Emulator verwendet werden. Dieser basiert auf Hyper-V und ist somit erst ab Windows 8 verfügbar. Im Vergleich zu Googles Emulator ist Microsofts Variante um einiges schneller.

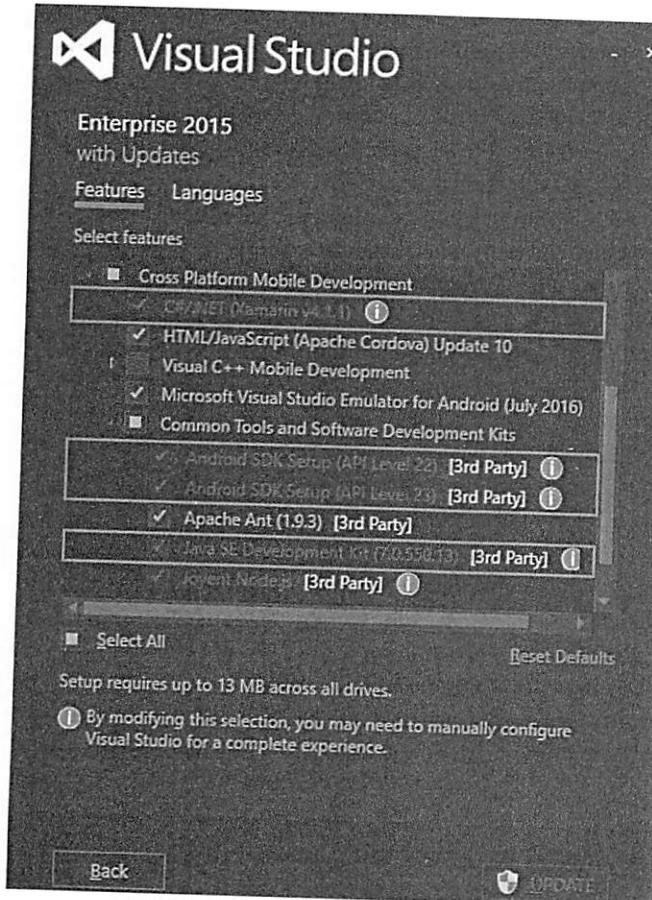
Da die Verwaltung des SDKs mit all seinen Bestandteilen recht aufwendig werden kann, steht zu diesem Zweck eine grafische Oberfläche bereit, der Android SDK Manager. Xamarin-Entwickler müssen diesen jedoch nur recht selten direkt bedienen, da Visual Studio bereits während der Installation eine sehr gute Vorauswahl der von diesem Tool verwalteten Komponenten trifft. So werden beispielsweise die Android-SDK-Plattformen für die APIs der Level 19, 21 22 und 23 bereits mit eingerichtet. Dies ermöglicht das Entwickeln von Apps ab Android 4.4 KitKat bis zu Android 6.0 Marshmallow.

Die API-Level

Bei den zuvor genannten API-Levels handelt es sich um eindeutige ganzzahlige Bezeichner, die eine Android-Version genau identifizieren. Anhand des API-Levels lässt sich also nicht nur ermitteln, um welche Android-Version es sich handelt (19 steht zum Beispiel für Android 4.4 KitKat), sondern auch, welche Bibliotheken mit welchem Funktionsumfang während der Entwicklung zur Verfügung stehen.

Wenn Sie eine App mit Xamarin erstellen, dann können Sie drei verschiedene API-Level für Ihre App konfigurieren:

- Minimum Android Version: Dies ist die niedrigste Version, die von der App unterstützt wird.



Visual Studio installiert sämtliche Komponenten für die Android-Entwicklung (Bild 5)

- Target Android Version: Dies ist die Zielversion der App, also die Version, die alle Merkmale der Anwendung unterstützt und mit der Sie in der Regel am intensivsten getestet haben.
- Target Framework: Mit dieser Version wird die App von Xamarin kompiliert.

Um Überraschungen zu vermeiden, ist es in der Regel sinnvoll, Target Android Version und Target Framework gleich zu halten.

Activities und Intents

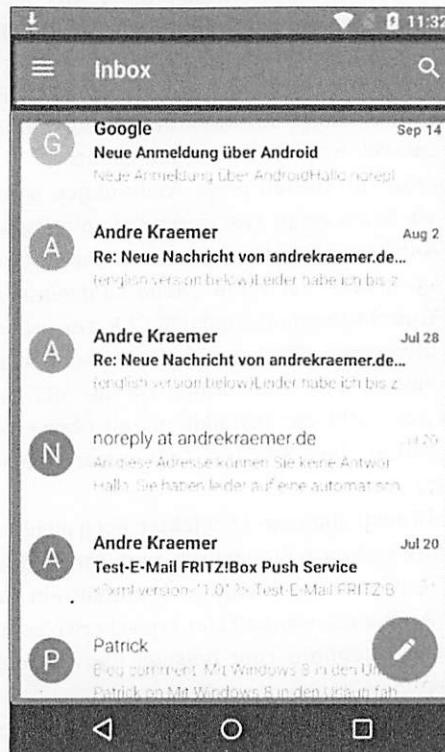
Android-Apps bestehen aus bis zu vier verschiedenen Komponententypen:

- Activities
- Services
- Content Providers
- Broadcast Receivers

Bei den Activities handelt es sich um Klassen, welche die Bildschirmmasken einer App repräsentieren. Typischerweise gibt es je Bildschirmmaske eine Activity. Eine solche Activity erbt von der Basisklasse *Activity* des Android-Anwendungs-Frameworks oder einer davon abgeleiteten Klasse. In *Activity* gibt es vordefinierte Methoden, die im Rahmen der Lebenszyklusereignisse einer Activity aufgerufen werden. Diese lassen sich in den eigenen Activities überschreiben.

● Listing 1: Beispiel einer Android-Activity

```
[Activity(Label = „SavedIt.Droid“, MainLauncher =  
true, Icon = „@drawable/icon“)]  
public class MainActivity : Activity  
{  
  
    protected override void OnCreate(Bundle bundle)  
    {  
        base.OnCreate(bundle);  
  
        // Layout laden  
        SetContentView(Resource.Layout.Main);  
  
        // Event Handler registrieren  
        var saveButton = FindViewById<Button>(Resource.  
Id.saveButton);  
        saveButton.Click += SaveButton_Click;  
  
        var detailButton =  
FindViewById<Button>(Resource.Id.detailsButton);  
        detailButton.Click += DetailButton_Click;  
  
    }  
    // ... weiterer Code wurde zur besseren Übersicht  
    entfernt
```



Eine App im
Android Material
Design (Bild 6)

Eine dieser Methoden, nämlich die Methode *OnCreate()*, muss auf jeden Fall überschrieben werden. Sie wird aufgerufen, wenn die Activity erzeugt wird, ist für die Initialisierung der Activity verantwortlich und lädt das Layout der Bildschirmmaske. Bei den Layouts handelt es sich um XML-Dateien, welche die Struktur der Bildschirmmaske beschreiben. Aus Sicht eines .NET-Entwicklers sind Layout-XML-Dateien mit XAML vergleichbar.

Listing 1 zeigt beispielhaft einen Ausschnitt aus dem Quellcode einer Activity. Hier wird die Methode *OnCreate()* unter anderem dazu verwendet, Ereignishandler zu registrieren; und natürlich zum Laden des Layouts der Oberfläche über den Befehl *SetContentView()*. Den Inhalt eines Layouts zeigt Listing 2. Wie bereits angedeutet, dürfen sich versierte XAML-Entwickler aufgrund der hohen Ähnlichkeit schnell zurechtfinden.

Da Apps meist über mehr als eine Bildschirmmaske verfügen, gibt es in einer typischen Anwendung auch mehr als eine Activity. Die Navigation von einer Activity zur nächsten wird über einen sogenannten Intent ausgelöst. Dabei handelt es sich um ein Nachrichtenobjekt, das zum Auslösen einer Aktion genutzt werden kann.

Das folgende Codebeispiel zeigt, wie ein solcher Intent in der Anwendung verwendet wird. Hier wird die Zielaktivität als Typparameter an den Konstruktor der Klasse *Intent* übergeben:

```
var intent = new Intent(this, typeof(DetailActivity));  
var json = Newtonsoft.Json.JsonConvert.  
    SerializeObject(_savedItems);  
intent.PutExtra("SavedItems", json);  
StartActivity(intent);
```

Über die Methode `PutExtra()` können an `intent` Daten übergeben werden, die sich in der Ziel-Activity auslesen lassen. Im Beispiel ist zu sehen, dass eine Liste von Objekten zunächst in einen String im JSON-Format serialisiert und anschließend an das `Intent`-Objekt gehängt wird. Die Methode `StartActivity()`, die ein `Intent`-Objekt entgegennimmt, verschickt die Nachricht schließlich.

Intents können nicht nur innerhalb einer App, sondern auch App-übergreifend genutzt werden. So lässt sich über einen Intent zum Beispiel eine Activity der registrierten Kamera-App starten, um ein Bild aufzuzeichnen, das in der eigenen App weiterverarbeitet werden soll.

Services, Content-Provider, Broadcast-Receiver

Bei den drei weiteren Android-Kernkomponenten neben den Activities handelt es sich um Komponenten ohne eigenes UI. Services führen Hintergrundaktionen aus, zum Beispiel Datei-Downloads oder das Abspielen von Musik; Content-Provider bieten eine standardisierte Schnittstelle zum Lesen und Schreiben von Applikationsdaten; Broadcast-Receiver verarbeiten über App-Grenzen hinweg systemweite Benachrichtigungen.

Oberflächliches

Die Version 5.0 von Android (Lollipop) führte für Apps den Leitfaden „Material Design for Android“ ein. Darin beschreibt Google, wie visuelle Elemente, Animationen und Interaktionen umgesetzt werden sollten. Der optische Stil des Material Design kann als flaches Design mit leichten Schattierungen beschrieben werden. Auf den ersten Blick erinnert er an Microsofts flaches Windows-Design oder an das Aussehen des aktuellen iOS. Im Gegensatz zum puren Flat Design, das vollkommen auf Dreidimensionalität verzichtet, führt Material Design die dritte Dimension in Form von leichten Schlagschatten ein. Ein Beispiel für eine App im Material Design zeigt Bild 6 [3]. Typische Bestandteile sind:

- die App Bar (schwarz umrahmt): Enthält den Titel und Schaltflächen für Kernfunktionen im rechten Bereich. Links kann ein Navigationsmenü sein.
- der Content-Bereich (rot umrahmt): der Inhalt der App.

Navigation

Die Navigation wird in Android-Apps häufig über Tabs oder eine seitliche Navigation umgesetzt (auch als „Navigation Drawer“ oder „Burger Menu“ bekannt).

● Listing 2: Ein Android-Layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
        android:text="Das habe ich mir gespart"
        android:textAppearance=
            "?android:attr/textAppearanceMedium"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/textView1" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/ItemText"
        android:hint=
            "z. B. einen Burger, einen Kaffee, ..." />

    <EditText
        android:inputType="numberDecimal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/priceText"
        android:hint="Preis. z. B. 0.99" />

    <Button
        android:text="Sparen!"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/saveButton" />

    <TextView
        android:text="Bisher gespart"
        android:textAppearance=
            "?android:attr/textAppearanceMedium"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/textView2" />

    <TextView
        android:text="0,00"
        android:textAppearance=
            "?android:attr/textAppearanceMedium"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/SavedText" />

    <Button
        android:text="Details"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/detailsButton" />

</LinearLayout>
```

Tabs werden meist für Anwendungen mit wenigen primären Bildschirmmasken eingesetzt. Dies ist auch durchaus sinnvoll, da jede Hauptmaske typischerweise einen eigenen Tab hat. Würden Tabs bei sehr vielen primären Masken eingesetzt werden, so müsste der Anwender horizontal in der Tab-Liste scrollen, was zugunsten einer besseren Auffindbarkeit der Tabs vermieden werden sollte.

Eine Besonderheit der Android-Tabs gegenüber Tabs zum Beispiel bei iOS ist, dass sie sich am oberen Bildschirmrand – und nicht wie bei iOS am unteren Bildschirmrand – befinden. Am oberen Rand sind die Tabs zwar bei einhändiger Bedienung des Mobilgeräts etwas schwieriger vom Anwender zu erreichen, dafür hilft diese Position jedoch, Fehleingaben vorzubeugen.

Da am unteren Rand die Softkeys des Betriebssystems zur Navigation sitzen (Zurück, Startbildschirm und App-Auswahl), wäre das Risiko, versehentlich einen solchen statt eines Tabs zu drücken, recht hoch, wenn die Tabs sich ebenfalls unten befänden.

Enthält eine Anwendung sehr viele Bildschirmmasken, wird die Navigation nicht mehr mit Tabs, sondern meist mit einer seitlichen Navigation umgesetzt. Diese Form der Navigation öffnet nach dem Klick auf ein Symbol einen Bereich an der Seite des Bildschirms, in dem sich eine Vielzahl an Menüoptionen unterbringen lässt.

Unter iOS wird im Gegensatz zu Android explizit vom Einsatz der seitlichen Navigation abgeraten, da die Gefahr besteht, dass das Menü durch den Anwender nicht als solches wahrgenommen werden könnte. Da das seitliche Menü, symbolisiert durch drei horizontale Striche, unter Android jedoch auch in den Google-eigenen Apps verwendet wird, ist die Gefahr, dass das Menü unentdeckt bleiben könnte, jedoch gering.

In der Regel sollte der Entwickler davon ausgehen können, dass der Anwender bereits durch die Google-Apps mit dieser Art der Navigation vertraut ist.

Fazit

Die freie Verfügbarkeit der Xamarin-Werkzeuge lädt förmlich zur App-Entwicklung unter Android ein. Trotz der bekannten Instrumente und Programmiersprachen muss dem Entwickler jedoch klar sein, dass Android an einigen Stellen anders funktioniert, als er es von anderen Plattformen gewohnt ist. Kennt man jedoch diese Eigenheiten, wie zum Beispiel Intents und Activities, so steht der Entwicklung einer ersten App nichts mehr im Weg. ■



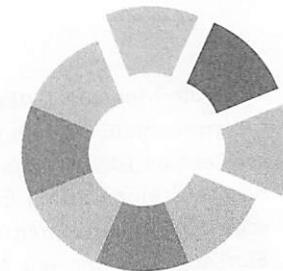
André Krämer

ist selbstständiger Entwickler, Trainer und Berater mit den Schwerpunkten Webanwendungen und mobile Apps, Analyse von Speicher-/Laufzeitproblemen und automatisiertes Erzeugen von Dokumenten.

<http://andrekraemer.de>

dnpCode

A1611Android



SMART DATA

Developer Conference

Big Data & Smart Analytics

Für Softwareentwickler
und IT-Professionals

06. Dezember 2016, Köln

- Effizienter durch standardisierte Reports
- Datenqualität erhöhen
- Recommender Algorithmen: R vs. Spark
- Streaming = Zukunft von Big Data

dotnetpro-
Leser erhalten
15% Rabatt
mit Code SMART16dnp

Veranstalter:

developer media

Neue
Mediengesellschaft
Ulm mbH