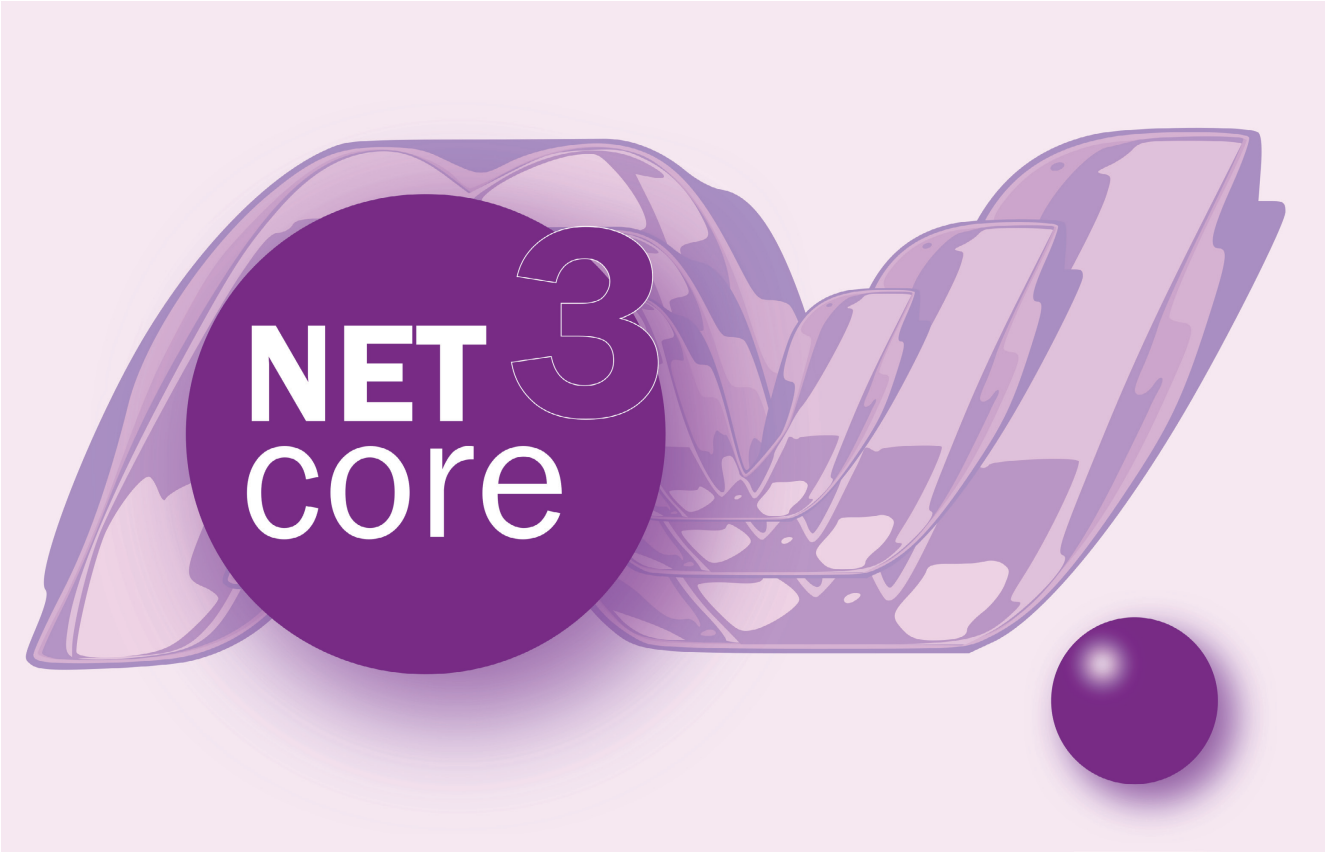


MIGRIEREN VON KOMPLEXEN DESKTOP-ANWENDUNGEN

.NET Core nun endlich für den Desktop

WPF oder WinForms für .NET Core 3.0: Was in der Preview-Version schon geht und wie eine Portierung möglich ist.

Es ist so weit – das von einigen in der Entwicklergemeinde erahnte und vielleicht auch unausweichliche Ende des .NET Frameworks wurde auf der diesjährigen Entwicklerkonferenz „Build" ebenso bekannt gegeben wie die damit verbundene Fokussierung auf .NET Core. .NET Framework 4.8 wird die letzte Hauptversion sein und zukünftig nur noch um Bugfixes und Sicherheitsupdates aktualisiert.

.NET Core schließt in der Version 3.0 mit dem angebotenen API ein ganzes Stück auf das klassische .NET Framework auf. Der vorliegende Artikel wirft einen ersten Blick auf die neuen Möglichkeiten, Desktop-Anwendungen für .NET Core zu entwickeln; er prüft, was mit der Preview 5 bereits alles geht und was noch fehlt, und er beschreibt Möglichkeiten der Portierung bereits existierender Anwendungen.

Microsoft verpackt die plattformspezifischen Windows-Technologien WPF, WinForms und UWP unter dem Begriff „Windows Desktop Packs", was aus technischer Sicht eine auf .NET Core aufbauende Bibliothek darstellt, die ausschließlich unter Windows-Betriebssystemen lauffähig ist. Damit ändert sich nichts an Microsofts Strategie, mit .NET Core eine plattformneutrale Laufzeitumgebung bereitzustellen.

Voraussetzungen

Die beim Schreiben dieses Artikels vorliegende Version ist .NET Core 3.0 Preview 5, die Anfang Mai 2019 erschienen ist. Das entsprechende Software Development Kit (SDK) kann unter [1] bezogen werden. Es empfiehlt sich, immer die letzte verfügbare Version zu verwenden. ▶

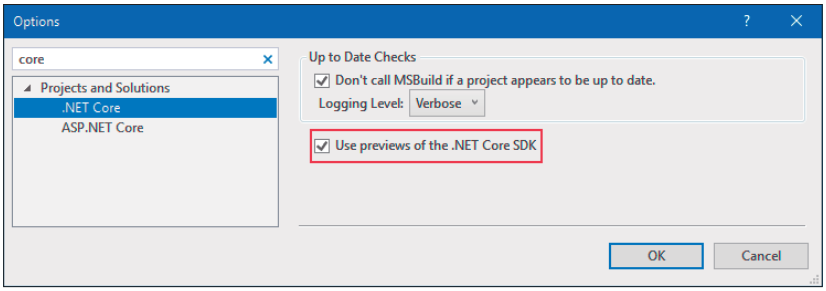
Als Entwicklungsumgebung wird Visual Studio 2019 benötigt. Die Vorgängerversionen inklusive 2017 werden nach Microsoft-Angaben keinen Support für .NET Core 3.0 mehr erhalten. Visual Studio Code kann zwar prinzipiell mit .NET-Core-3.0-Projekten umgehen, diese bauen und sie auch debuggen, jedoch fehlt unter anderem die Unterstützung von IntelliSense für XAML-Dateien, weshalb Visual Studio Code bei der Entwicklung von Desktop-Anwendungen eher ungeeignet ist. Selbst für Puristen ist das kein Arbeiten.

Da es sich derzeit bei .NET Core 3.0 noch um eine Preview-Version handelt, müssen Sie Visual Studio explizit die Erlaubnis erteilen, eine Preview-Version als Zielplattform nutzen zu dürfen. Eine entsprechende Option befindet sich in den Einstellungen unter *Projects and Solutions* | *.NET Core*. Die Visual-Studio-Einstellungen erreichen Sie über das Menü *Tools* | *Options* (Bild 1).

Einschränkungen der Preview

Eine erste Einschränkung besteht bei der Wahl der Programmiersprache: VB.NET wird derzeit noch nicht vollständig unterstützt. Erstellt man in Visual Studio eine neue WPF- oder WinForms-Anwendung, steht nur C# zur Auswahl. Erstellt man mit dem .NET Core CLI ein neues Projekt, so lässt sich darüber zumindest schon eine WinForms-Anwendung erstellen, aber keine WPF-Anwendung. Nach Microsoft-Angaben soll eine entsprechende Unterstützung in der finalen Version enthalten sein.

Ist die WinForms- oder WPF-Anwendung erstellt, bemerkt man recht schnell, dass Visual Studio keinen Dialog-Designer für .NET-Core-Projekte anzeigt. Das mag in WPF-Projekten noch akzeptabel (bezogen auf die Performance des XAML-Designers für manch einen Entwickler sogar ein Segen) sein, aber für WinForms-Anwendungen ist dieser Umstand derzeit ein Showstopper.



Verwenden von .NET-Core-Preview-Versionen in Visual Studio (Bild 1)

Die Code-behind-Dateien in WinForms-Anwendungen werden ausschließlich vom Designer verwaltet, und das manuelle Editieren in der Datei bringt den Visual Studio Designer recht schnell aus dem Tritt. Ein möglicher Workaround besteht darin, ein .NET-Framework-Projekt neben dem .NET-Core-Projekt zu erstellen und die Quellcode- und Ressourcen-Dateien in beiden Projekten mittels Links zu referenzieren. Das .NET-Core-Projekt dient dann zum Bauen und das .NET-Framework-Projekt zum Editieren der Dialoge. Aber auch dieser Workaround hat seine Grenzen. Es ist ziemlich sicher, dass Microsoft hier bis zum Release-Termin im September Abhilfe schaffen wird.

Vorbereitung

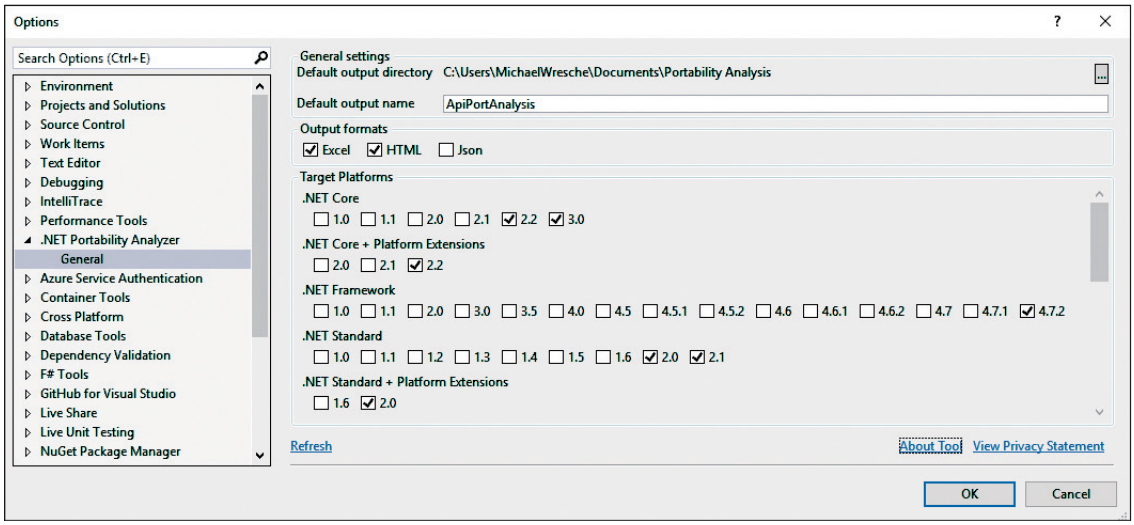
Unabhängig davon, ob eine existierende WPF- oder WinForms-Anwendung nach .NET 3.0 Core portiert werden oder eine komplett neue Anwendung entwickelt werden soll, müssen die Randbedingungen, die mit .NET Core einhergehen, bedacht werden. Microsoft beschreibt unter [2] ältere Technologien, die unter .NET Core heute und in Zukunft nicht verfügbar sein werden. Hierzu gehören unter anderem .NET Remoting und Application Domains. Sind diese Technologien im Einsatz, muss ein Plan erdacht werden, die betreffenden Komponenten gegen neue, moderne Implementierungen auszutauschen, und das notwendigerweise noch vor der Entwicklung/Portierung der UI-Anwendung.

Unabhängig davon, ob die Codebasis einer WPF-Applikation, einer Klassenbibliothek oder eines Webprojekts nach .NET Core portiert werden soll, ist es immer ratsam, die Kompatibilität zu prüfen. Diese gibt Aufschluss darüber, wie viel .NET-Framework-API genutzt wurde, das es in der anvisierten Version von .NET Core oder .NET Standard nicht gibt. Auch wenn die portierte WPF- oder WinForms-Anwendung weiterhin ausschließlich unter Microsoft Windows ausgeführt wird, unterscheiden sich die zur Verfügung stehenden APIs von .NET Framework (Classic), .NET Core und .NET Standard untereinander und natürlich auch in ihren Versionen.

Beinahe so alt wie .NET Core ist der von Microsoft entwickelte .NET Portability Analyzer. Dabei handelt es sich um eine Visual-Studio-Erweiterung, die einzelne oder alle Projekte einer Solution hinsichtlich des genutzten .NET API und deren Kompatibilität zu verschiedenen .NET-Plattformen prüft. Im Ergebnis erhält man eine Übersicht, die Aufschluss gibt, wie viel des genutzten API in der Zielplattform verfügbar ist. Bevor die Analyse ausgeführt wird, empfiehlt es sich, nur die notwendigen Plattformen auszuwählen. In den entsprechenden Konfigurationsdialog gelangt man über das Menü *Analyze* | *Portability Analyzer Settings* (Bild 2).

Hier wählen wir zunächst *HTML* als Ausgabeformat. Da eine WPF- oder WinForms-Anwendung migriert werden soll, wählt man *.NET Core 3.0* aus. Für reine Klassenbibliotheken bietet es sich gegebenenfalls an, sie auf .NET Standard zu migrieren statt auf .NET Core. Daher setzt man zusätzlich einen Haken bei *.NET Standard + Platform Extensions 2.0*.

Mit Platform Extensions ist ein von Microsoft entwickeltes NuGet-Paket namens Microsoft.Windows.Compatibility gemeint, das beim Umstieg auf .NET Core die vielen plattformspezifischen APIs kapselt, die nicht Kernbestandteil von .NET Core sind [3], wie beispielsweise der Zugriff auf die Windows Registry. Dieses NuGet-Paket muss bei der Portierung entsprechend eingebunden werden, wenn es nötig ist. Interessanterweise steht im Einstellungsdialog des Analyzers die Option *.NET Core + Platform Extensions* für Version 3.0 nicht



Konfigurationsdialog für den Portability Analyzer (Bild 2)

zur Verfügung. Ungeachtet dessen berücksichtigt der Analyzer aber bei der Auswahl von .NET Core 3.0 die Platform Extensions implizit mit.

Nachdem die Analyse ausgeführt wurde (rechte Maustaste und *Analyze Assembly Portability*), öffnet sich der HTML-Report. Er enthält im oberen Teil eine prozentuale Zusammenfassung des kompatiblen API. Ein Wert von 100 Prozent bedeutet, dass das gesamte API des Projekts auf der Zielplattform verfügbar ist und keinerlei Anpassungen notwendig sind. Darunter befindet sich eine Aufschlüsselung, welche Projekte ein API einsetzen, das in einer der ausgewählten Zielplattformen nicht verfügbar ist (siehe Bild 3).

Bedauerlicherweise bezieht sich das Ergebnis des zuvor erwähnten Tools ausschließlich auf die Projekte der Solution, nicht aber auf externe Abhängigkeiten wie per NuGet oder direkt eingebundene Bibliotheken. Es ist prinzipiell möglich, dass ein .NET-Core-Projekt eine .NET Framework Assembly referenziert, jedoch unterliegt diese den gleichen Einschränkungen hinsichtlich des nutzbaren API wie das .NET-Core-Projekt. Hat man Zugriff auf die Quellen dieser Projekte, sollte man sie ebenfalls mit dem Portability Analyzer überprüfen und eine Portierung nach .NET Core oder .NET Standard in Erwägung ziehen. Schlecht dagegen sieht es aus, wenn es sich um proprietäre Lösungen handelt und diese nicht für .NET Core oder .NET Standard verfügbar sind. Hier hilft nur das schlichte Ausprobieren weiter.

Grundlegendes

Nachdem die Voraussetzungen für die Entwicklung von Desktop-Anwendungen auf Basis von .NET Core geschaffen sind, wollen wir uns zunächst einmal mit den Änderungen befassen, die in der .NET-Core-Welt schon länger Einzug gehalten haben.

Visual-Studio-Projekte, die für das .NET Core oder .NET Standard Framework geschrieben werden, verwenden das sogenannte SDK-Style-Projektformat. Der Inhalt dieser Projektdatei hat sich deutlich verschlankt, wenngleich sie weiterhin eine XML-Datei für die Build-Plattform msbuild darstellt. Der grundlegende Aufbau der Datei ist dem des früheren Formats sehr ähnlich, jedoch entfallen die vielen Zeilen, welche die Build-Konfigurationen beschreiben. Auch die explizite Definition der Dateien, die vom Compiler übersetzt werden sollen, kann entfallen (File Globbing). Konsequenterweise kommt dieses Projektformat nun auch für WPF- oder WinForms-Projekte auf Basis von .NET Core zum Einsatz. Listing 1 zeigt den Minimalinhalt eines derartigen WPF-Projekts.

Ein nettes Feature in Visual Studio 2019 ist, dass man den Inhalt einer Projektdatei durch einen Doppelklick auf das Projekt im Solution Explorer öffnen und anschließend direkt editieren kann. Unter Zuhilfenahme von entsprechenden Visual Studio Extensions konnte man das zwar auch früher schon, jedoch muss Visual Studio 2019 jetzt das Projekt nach dem Speichern der Projektdatei nicht mehr komplett neu laden. Visual Studio kommt nun besser damit klar, wenn diese Datei manuell editiert wird.

Änderungen gab es seit Beginn von .NET Core auch bei der Art und Weise, wie NuGet-Pakete eingebunden werden. Vielen bekannt dürfte eine Datei namens *packages.config* sein, in der die eingebundenen NuGet-Pakete aufgeführt waren. In der eigentlichen Projektdatei waren parallel die Verweise auf die Assemblies des NuGet-Pakets enthalten.

Diese doppelte Erwähnung von externen Abhängigkeiten ist in der neuen Projektdatei anders gelöst. Die *packages.config* gibt es nun nicht mehr. Stattdessen enthält die Projekt- ▶

Listing 1: SDK-Style-Projektdatei

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0
      </TargetFramework>
    <UseWPF>true</UseWPF>
  </PropertyGroup>
</Project>
```

Portability Summary

Assembly	.NET Core + Platform Extensions,Version=v2.2	.NET Core,Version=v3.0	.NET Standard + Platform Extensions,Version=v2.0	.NET Standard,Version=v2.0
Application, Version=7.2.0.0, Culture=neutral, PublicKeyToken=76a4436585310c1a (.NETFramework,Version=v4.6.1)	100,00 %	100,00 %	100,00 %	100,00 %
Application, Version=7.2.0.0, Culture=neutral, PublicKeyToken=76a4436585310c1a (.NETStandard,Version=v2.0)	100,00 %	100,00 %	100,00 %	100,00 %
Application.Tests, Version=7.2.0.0, Culture=neutral, PublicKeyToken=76a4436585310c1a (.NETFramework,Version=v4.7.2)	100,00 %	100,00 %	100,00 %	100,00 %
Domain, Version=7.2.0.0, Culture=neutral, PublicKeyToken=76a4436585310c1a (.NETStandard,Version=v2.0)	100,00 %	100,00 %	100,00 %	100,00 %
Domain.Tests, Version=0.0.0.0, Culture=neutral, PublicKeyToken=76a4436585310c1a (.NETFramework,Version=v4.7.2)	100,00 %	100,00 %	100,00 %	100,00 %
Infrastructure, Version=7.2.0.0, Culture=neutral, PublicKeyToken=76a4436585310c1a (.NETFramework,Version=v4.7.2)	100,00 %	100,00 %	100,00 %	95,25 %
Infrastructure.Tests, Version=7.2.0.0, Culture=neutral, PublicKeyToken=76a4436585310c1a (.NETFramework,Version=v4.7.2)	100,00 %	100,00 %	100,00 %	98,45 %
Viewer, Version=7.2.0.0, Culture=neutral, PublicKeyToken=76a4436585310c1a (.NETFramework,Version=v4.7.2)	79,16 %	100,00 %	79,16 %	79,16 %

Vollständige .NET-Core-3.0-Kompatibilität des Projekts (Bild 3)

datei sogenannte PackageReferences-Elemente, die den Namen des zu referenzierenden NuGet-Pakets enthalten. Die zusätzliche Einbindung der Assemblies des Pakets ist nicht mehr notwendig. Diese neue Technik wird gemeinhin als PackageReference beschrieben und bietet eine ganze Reine von Vorteilen, die unter [4] nachzulesen sind.

Zu erwähnen wäre noch, dass die heruntergeladenen Pakete nicht mehr standardmäßig in einem *packages*-Ordner in der Solution liegen, sondern unter *%userprofile%\.nuget\packages* zu finden sind.

Portieren von Anwendungen

Im Rahmen dieses Artikels wurden vom Autor drei reale Projekte aus dessen Projektalltag herangezogen, um eine Portierung auf .NET Core durchzuführen. Sie hatten eine Größe von 7k, 18k und 36k Lines of Code (LoC) – geschrieben in C#. Alle verwenden WPF und benutzen die UI-Bibliotheken von DevExpress. Teilweise kam der O/R-Mapper EntityFramework zum Einsatz. Die Projekte machten regen Gebrauch von gängigen Fremdbibliotheken wie Prism, IoC-Containern, Loggern et cetera, die über NuGet eingebunden waren, und wurden in den letzten fünf Jahren entwickelt.

Um es vorwegzunehmen: Es konnte nur eines der genannten Projekte erfolgreich portiert werden. Ist .NET Core für die Entwicklung von Desktop-Applikationen noch nicht so weit?

Die Antwort lautet: Es kommt darauf an! Die Portierung an und für sich ist einfach zu bewerkstelligen (siehe den nachfolgenden Abschnitt). Die aufgetretenen Probleme haben weniger mit dem Preview-Status von .NET Core 3.0 oder der aktuellen Implementierung von WPF zu tun. Vielmehr sind zwei der drei Projekte bei der Portierung an ihren Abhängigkeiten zu Fremdbibliotheken gescheitert, die es eben (noch) nicht für .NET Core oder .NET Standard gibt. Teilweise ist es fraglich, ob diese Bibliotheken überhaupt einmal für .NET Core zur Verfügung stehen werden, und so muss man sich Alternativen suchen.

Die eine erfolgreich verlaufene Portierung war jedoch mit einigem Aufwand verbunden. Der eingesetzte Data Layer auf Basis von EntityFramework 6 war nicht gut auf .NET Core und Visual Studio 2019 zu sprechen, obwohl EntityFramework 6 ein integraler Bestandteil des Windows Desktop Pack ist. Allen Versuchen zum Trotz half hier nur eine Portierung auf EF Core, die für sich auch schon einen gewissen Aufwand darstellt. Dieser hatte sich aber gelohnt, nachdem die Applikation dann endlich als .NET-Core-WPF-Desktop-Anwendung startete.

Ablauf einer Portierung

Die nachfolgenden Ausführungen beschreiben eine mögliche Strategie, wie die Codebasis existierender Desktop-Anwendungen auf .NET Core 3.0 portiert werden kann. Als Ausgangszustand dient eine Solution mit mehreren Projekten, darunter Klassenbibliotheken für Domänenlogik und Datenschicht, ein oder mehrere UI-Projekte auf Basis von WPF oder Windows Forms und Unit-Test-Projekte.

Basieren die zu portierende Applikation und deren referenzierte Projekte noch nicht auf dem letzten verfügbaren .NET Framework, empfiehlt sich zunächst ein Update aller Projekte auf die letzte Version, um gegebenenfalls die Verwendung abgekündigter APIs ausfindig zu machen. Der Vorteil, diese Arbeiten vor der Portierung durchzuführen, ist der, dass die Applikation bei etwaigen Anpassungen noch ausführbar ist, um die Anpassungen losgelöst testen zu können.

In einem ersten Schritt sollten alle Klassenbibliotheken, die keinen Verweis auf eine UI-Technologie haben, in .NET-Standard-Projekte umgewandelt werden. Sie könnten auch nach .NET Core portiert werden, jedoch können diese dann ausschließlich unter .NET Core genutzt werden. Mit .NET Standard hält man sich die Möglichkeit offen, die Assembly unter .NET Framework und unter .NET Core laufen zu lassen. Da es während der Portierung zwangsläufig zu Problemen kommt, in denen man den Zustand der Projekte vor der Portierung betrachten möchte, empfiehlt es sich, nicht die vorhandenen Projekte zu bearbeiten, sondern neue Projekte daneben zu erstellen, die im Projektnamen das Suffix *.Core* aufweisen.

Um die Quelltextdateien in das neue .NET-Standard-Projekt einzubinden, kopiert man entweder alle Dateien auf Da-

teiebene in das neue Projekt oder behilft sich mit dem folgenden Trick: Man öffnet die .NET-Standard-Projektdatei und trägt die folgenden Abschnitte ein:

```
<ItemGroup>
  <Compile Include="..\[OriginalProject]\**\*.cs" />
</ItemGroup>
```

Den Term *OriginalProject* ersetzt man durch den Namen des zu portierenden Projekts. Damit sind die Quelltextdateien im neuen Projekt als Links referenziert. MSBuild erstellt für Projektdateien im SDK-Style-Format standardmäßig eine *AssemblyInfo.cs*.

Da mit der obigen Anweisung jedoch die *AssemblyInfo.cs* des ursprünglichen Projekts enthalten ist, wird es während des Kompilierens zu einem Fehler kommen. In diesem Fall setzt man die automatische Erstellung der *AssemblyInfo.cs* mit den folgenden Zeilen, die in der Projektdatei hinzugefügt werden müssen, außer Kraft:

```
<PropertyGroup>
  <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
</PropertyGroup>
```

Im letzten Schritt fügt man die externen Abhängigkeiten, seien es NuGet-Pakete oder referenzierte Projekte der gleichen Solution, hinzu.

Nun zeigt sich, ob die Analyse der Kompatibilität der Abhängigkeiten und des Projekts gewissenhaft durchgeführt wurde, denn jetzt sollte sich das Projekt übersetzen lassen.

Als Nächstes folgt die Portierung der WPF- beziehungsweise WinForms-Anwendung. Der Ablauf ähnelt prinzipiell dem gerade beschriebenen Weg, nur dass anstelle einer .NET-Standard-Klassenbibliothek eine .NET-Core-3.0-WPF-Anwendung erstellt wird.

Das Hinzufügen der Quelldateien sollte nun durch Kopieren der Dateien auf Dateiebene vorgenommen werden. Die Dateien müssen, wie oben bereits erwähnt, nicht explizit zum Projekt hinzugefügt werden. Unterscheidet sich der Standard-Namespace des Projekts vom Projektnamen, so muss dieser in den Projekteigenschaften noch angepasst werden, da es sonst zu Fehlern bei der Verwendung von Ressourcendateien kommen kann.

Diese Schrittfolge zur Portierung stellt nur eine grobe Richtung für den Ablauf dar. Abhängig vom zu portierenden Projekt kann es zu weiteren und spezifischeren Problemen kommen, die dann individuell gelöst werden müssen.

Fallstricke

Projekte mit vielen externen Abhängigkeiten stellen eine der größten Herausforderungen bei einer Portierung und auch bei neuen Projekten dar. .NET-Core-Anwendungen erlauben prinzipiell das Einbinden von .NET-Framework-Assemblies. Diese Mischung aus beiden Welten kann jedoch zu Problemen führen, wie das folgende Beispiel zeigt:

In einem neuen Projekt soll eine .NET-Framework-Assembly verwendet werden, die zunächst nicht portiert werden soll. Diese hat log4net als Abhängigkeit, und auch die neue Anwendung soll log4net verwenden. Nach dem Einbinden und Starten der Applikation kommt es jedoch zu MissingMethodExceptions. Die Ursache dafür ist, dass die Bibliothek log4net zwar für beide Plattformen verfügbar ist, jedoch jeweils ein leicht unterschiedliches API aufweist. Da die Hauptanwendung die .NET-Core-kompatible Version in den Ausgabeordner legt, fehlen einige API-Methoden, die es ausschließlich für .NET Framework gibt.

Fazit

Microsoft hat einen weiteren Schritt in Richtung .NET Core gemacht und ermöglicht nun einem breiteren Spektrum von Applikationen den Umstieg. Sieht man von den fehlenden, aber notwendigen Features wie eine vollständige Unterstützung der bekannten Oberflächen-Designer ab, geht in der aktuellen Version fast alles, was in WPF-Anwendungen unter .NET Framework auch ging.

Doch das allein reicht noch nicht, um einen Umstieg existierender Anwendungen zu vollführen. Die meisten Anwendungen besitzen eine Fülle an externen Abhängigkeiten, die vor einer Portierung erst einmal .NET-Core-kompatibel sein müssen. Darin liegt die Crux des Ganzen. Es wird eine Zeit dauern, bis die Entwickler dieser Bibliotheken nachgezogen haben und das Problem der Abhängigkeiten aus dem Weg geräumt ist.

Projektteams, die (Stand heute) darüber entscheiden müssen, worauf ihre neue Windows-Desktop-Applikation aufbauen soll, haben es auch ohne die Betrachtung externer Abhängigkeiten schwer. .NET Framework zu verwenden bedeutet (übertrieben gesprochen), auf ein „totes Pferd" zu setzen. .NET Core dagegen ist die einzige Strategie Microsofts für die Zukunft, birgt jedoch derzeit noch ein hohes Risiko, da es noch nicht in der finalen Version freigegeben ist und es außerdem noch wenig Erfahrungswerte gibt. ■

[1] .NET Core3.0 Preview 5, Download SDK, https://dotnet.microsoft.com/download/dotnet-core/3.0
[2] .NET Framework technologies unavailable on .NET Core, www.dotnetpro.de/SL1909NETCorePraxis1
[3] Announcing the Windows Compatibility Pack for .NET Core, www.dotnetpro.de/SL1909NETCorePraxis2
[4] NuGet – PackageReference vs packages.config, www.dotnetpro.de/SL1909NETCorePraxis3
[5] How to port desktop applications to .NET Core 3.0, www.dotnetpro.de/SL1909NETCorePraxis4

Michael Wresche
ist Senior Consultant bei der Saxonia Systems AG und entwickelt seit mehr als zehn Jahren Lösungen im Microsoft .NET-Umfeld. Sie erreichen ihn unter **michael.wresche@saxsys.de**.

dnpCode A1909NETCorePraxis