Kapitel 11

Windows-Runtime (WinRT)

In diesem Kapitel:

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)

Die Windows-Runtime-API (WinRT) ist eine neue Programmierschnittstelle für das Windows-Betriebssystem, die die bisherige Windows 32-API (Win32-API) zunächst ergänzen und vielleicht eines Tages ablösen soll. WinRT ist zunächst nur für Windows 8 inklusive Windows Phone 8 verfügbar.

Windows 8

Dieses Buch will und kann nicht Windows 8 beschreiben. Sofern Sie sich noch nicht mit Windows 8 beschäftigt haben, soll nur eine grundlegende Änderung in Windows 8 hier vorweg kurz beschrieben werden, die relevant ist für die folgenden Ausführungen.

In Windows 8 gibt es zwei verschiedene Benutzeroberflächen:

- **Bisheriger Desktop (aber ohne Start-Schaltfläche)** Hier laufen bisherige Anwendungen mit C++, .NET, Visual Basic 6, Foxpro, Delphi, usw. auf Basis der bisherigen Windows 32-API. Microsoft betonte von Anfang an, dass auf dem bisherigen Desktop alle bisherigen Anwendungen laufen: »All your investments in Win32, .NET and Silverlight will be preserved. You will be able to run those apps on Windows 8.« (Aleš Holeček im Vortrag Platform for Metro style apps, BPS-1005) [*http://channel9.msdn.com/Events/BUILD/BUILD2011/BPS-1005*]
- **Kachel-basierte App-Oberfläche (im Design der Windows Store-Apps, vgl. Windows Phone)** Hier laufen die neuen Windows 8 Apps auf Basis der Windows-Runtime (WinRT). Apps unterliegen vielen Einschränkungen, z. B. können diese nur begrenzt das Dateisystem nutzen und nicht direkt auf Datenbanken zugreifen.

HINWEIS Microsoft selbst sieht Windows Store-Apps primär konsumentenorientiert. »Windows Store-Apps do not currently include built-in support for data-intensive scenarios such as line of business (LOB) scenarios. For example, there are no APIs for client-side SQL and no local database.« (*http://msdn.microsoft.com/en-us/library/windows/apps/hh465136.aspx*). Denkbar sind aber auch Windows Store-Apps als Einstieg in Geschäftsprozessanwendungen: Eine Übersicht über offene Aufgaben/Vorgänge wird als Windows Store-App angezeigt, auch einfache Bearbeitungsschritte sind dort möglich. Für komplexere Interaktionen erfolgt dann aber ein Wechsel zu einer Desktop-Anwendung (ein solcher Wechsel ist programmgesteuert möglich).

ACHTUNG Verwechseln Sie nicht Windows-Runtime (WinRT) und Windows RT. Windows-Runtime (WinRT) ist eine neue Programmierschnittstelle für Windows 8. Windows RT ist hingegen der Name für eine Variante des Windows 8-Betriebssystems, die auf ARM-Prozessoren läuft. Auf dem Windows RT-Betriebssystem gibt es keine Windows 32-API, sondern nur die Windows-Runtime-API. So ist Microsoft zu dem Namen gekommen. Der Name ist wegen der Verwechslungsgefahr sicherlich keine gute Wahl. Der frühere Name *Windows on ARM* (WOA) war besser.

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)

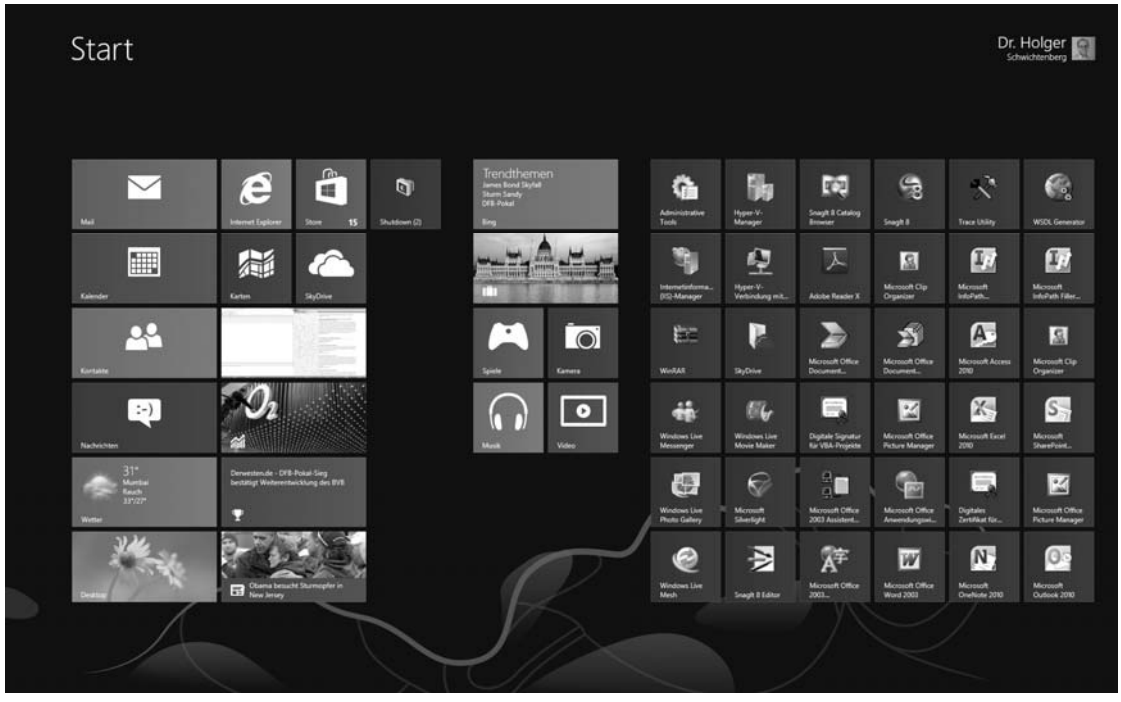

Abbildung 11.1 Startseite der Windows Store-Apps-Oberfläche

Programmieren für Windows 8

Im Vorfeld von Microsofts *BUILD Windows*-Konferenz im September 2011 gab es viele Spekulationen darüber, welche Entwicklungsplattform das Unternehmen für die Windows Store-Apps in Windows 8 verwenden werde. Ursprünglich dachten viele, es würde Microsoft Silverlight sein, wie in Windows Phone 7/7.5. Dann aber wurde klar, dass Microsoft von Silverlight immer mehr Abstand nimmt und HTML5/CSS/JavaScript favorisiert. Und dann gab es das große Gerücht, dass HTML5/CSS/JavaScript die einzige Entwicklungsplattform für Apps sein wird, vergleichbar mit den Gadgets in Windows Vista und Windows 7. Parallel dazu gab es aber auch Gerüchte über neue Laufzeitumgebungen und Bibliotheken wie Jupiter (*http://www.zdnet.com/blog/microsoft/more-on-microsoft-jupiter-and-what-it-means-for-windows-8/8373*) und Redhawk (*http://www.zdnet.com/blog/microsoft/microsoft-codename-redhawk-lives-in-windows-8/9233*).

So ist es nicht gekommen. Man kann Windows Store-Apps mit HTML5/CSS/JavaScript schreiben, aber es gibt insgesamt drei Alternativen (zukünftig wird es auch weitere Sprachen geben):

- Native Code mit C++ optional unter Einsatz der Windows Runtime Library (WRL) und der C++ Component Extensions (C++/CX)
- Managed Code mit C#, Visual Basic .NET oder anderen .NET-Sprachen unter Einsatz der CLR. Wichtig für .NET-Entwickler ist, dass hierbei keine vollständige .NET-Klassenbibliothek zur Verfügung steht, sondern nur eine stark abgespeckte Variante unter dem Namen ».NET for Windows Store-Apps«. Hier entfallen viele Klassen, die in Windows Store-Apps unerwünscht sind (z. B. Datenbankzugriffe) oder die in der Windows-Runtime-API (WinRT) bereits enthalten sind (z. B. Dateisystemzugriffe).
- JavaScript unter Einsatz der JavaScript-Bibliothek Windows Library for JavaScript (*WinJS*)

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)

HINWEIS Verwechseln Sie auch nicht die C++-Bibliothek Windows Runtime Library (WRL) mit der Windows-Runtime-API von Windows 8. Die WRL ist eine C++-basierte Hilfsbibliothek für die Arbeit mit der Windows-Runtime.

Für die Beschreibung der Benutzeroberfläche gibt es ebenfalls drei Alternativen:

- Entwickler, die mit C#, Visual Basic .NET, einer der anderen .NET-Sprachen und nativem C++ arbeiten, können die Oberfläche in der Extensible Application Markup Language (XAML) erstellen. XAML kommt schon in WPF und Silverlight zum Einsatz. Die Windows-Runtime-Variante von XAML unterscheidet sich in einigen Punkten (z. B. den Namensräumen und den verfügbaren Steuerelementen), baut aber auf den gleichen Prinzipien wie WPF-XAML und Silverlight-XAML auf. Bei der Programmierung mit .NET nutzt der Entwickler nicht das volle .NET Framework, sondern eine stark abgespeckte Variante mit Namen ».NET 4.5 for Windows Store-Apps«. Hier fehlen insbesondere Klassen für Funktionen, die entweder von der Windows-Runtime bereitgestellt werden (z. B. IO, Threading) oder aber in Windows Store-Apps gar nicht erwünscht sind (Datenbankzugriffe).
- Entwickler, die natives C++ verwenden, können alternativ auch DirectX nutzen. Wir gehen aber davon aus, dass dies nur für grafisch aufwändige Apps, insbesondere Spiele, eine Rolle spielen wird und nehmen diese Variante aus dem Vergleich heraus.
- JavaScript-Entwickler schließlich können die Oberfläche mit HTML und CSS beschreiben. Als Hilfsinstrument kommt dabei die Windows Library for JavaScript (*WinJS*)-Hilfsbibliothek (*base.js* und *ui.js*) von Microsoft zum Einsatz. WinJS stellt Steuerelemente und Hilfsfunktionen bereit, um HTML »App-fähig« zu machen. Es steht dem Entwickler frei, darüber hinaus weitere JavaScript-Bibliotheken (z. B. jQuery, *kockout.js*) zu nutzen. HTML-basierte Windows App (alias *WinWebApp*) bestehen im Gegensatz zu klassischen Webanwendungen und ähnlich wie einige Web 2.0-Anwendungen nur aus einer einzigen Webseite (Single Page Applications), in die Inhalte dynamisch geladen werden. Dadurch ist der wesentliche Bestandteil einer WinWebApp JavaScript. Die HTML- und JavaScript-Engine für das Rendering ist die gleiche wie in Internet Explorer 10 (»Chakra«) [*http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx*]. Für WinWebApps wird aber nicht der vollständige Browser geladen. Alle APIs des Browsers (z. B. Index DB, Application Cache, WebSockets, File API, CSS3 Transforms, CSS3 Animationen) stehen zur Verfügung. Darüber hinaus gibt es für WinWebApps mit *WinJS* eine zusätzliche JavaScript-Bibliothek, die die Windows Apps-Steuerelemente bereitstellt und den Zugang zu ausgewählten Funktionen des Betriebssystems ermöglicht. *WinJS* wird automatisch in jedes WinWebApp-Projekt eingebunden. Typischerweise sieht man in einem HTML-Tag Custom Attributs wie `data-win-control="WinJS.UI.AppBar"`.

Die folgende Abbildung zeigt die Architektur der Windows 8-Programmierschnittstellen aus der Sicht von Microsoft. Microsoft hat dieses Schaubild auf der BUILD Windows-Konferenz im September 2011 in der Keynote gezeigt. Es gibt an diesem Schaubild aber Einiges zu kritisieren:

- DirectX fehlt als Benutzeroberflächentechnik
- Es ist nicht klar, dass C# und Visual Basic weiterhin auf Basis der CLR laufen
- Es ist nicht klar, dass die Desktop-Anwendungen weiterhin XAML verwenden können
- Microsoft spricht in beiden Welten von »Apps«. Besser wäre, zwischen »Apps« und »Applications« zu differenzieren.
- Die Desktop-Anwendungen sehen unbedeutender aus als die Apps

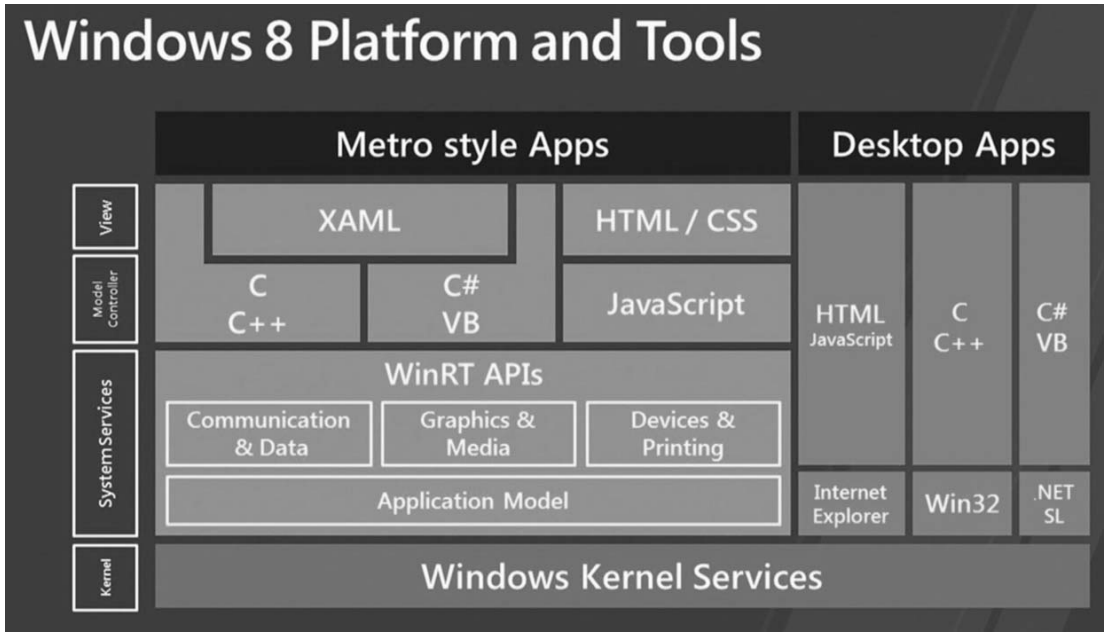

Abbildung 11.2 Die Architektur der Windows 8-Programmierschnittstellen aus der Sicht von Microsoft (Quelle: *Microsoft BUILD Windows-Konferenz 2011*)

Der Autor dieses Kapitels möchte dem daher ein anderes Schaubild entgegensetzen. In diesem Schaubild sind beide Welten gleichberechtigt dargestellt. Es wird klar herausgestellt, was auf Basis der CLR als Managed Code läuft und was nicht. DirectX ist erwähnt.

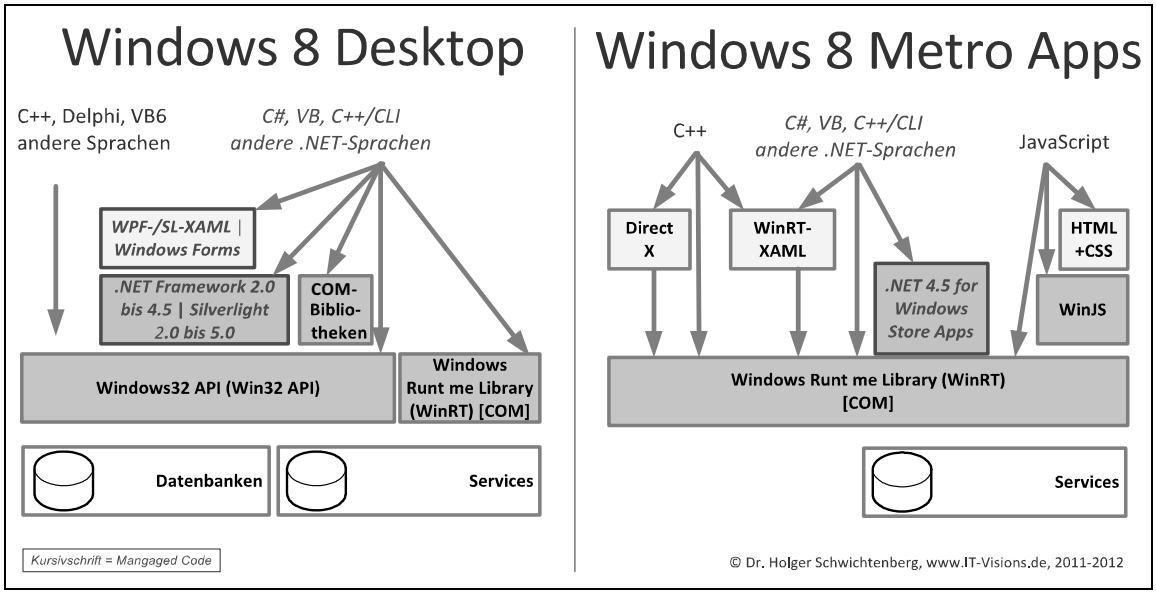

Abbildung 11.3 Die Architektur der Windows 8-Programmierschnittstellen aus der Sicht des Autors

WinRT-Konzepte

WinRT wird von Microsoft als integraler Bestandteil des Betriebssystems betrachtet, nicht als ein zusätzliches Framework. Daher wird WinRT zusammen mit jedem Build von Windows kompiliert. Und es heißt auch, dass WinRT nicht für ältere Betriebssysteme als Add-On nachgeliefert werden soll.

WinRT ist implementiert in C++ als eine COM-Komponente gemäß Component Object Model und liegt komplett in Native Code vor. Die alten COM-Konzepte (`IUknown`, Schnittstellenkonzept statt Klassen, Referenzzählung, Registry) sind dort wieder vorzufinden. Die für C# und Visual Basic sowie JavaScript verfügbaren Klassen sind nur *Wrapper*. Ein neues Typsystem sorgt dafür, dass die Typen in verschiedene Sprachen »projiziert« werden und damit dort verfügbar sind.

Microsoft hat aber einige Konzepte aus .NET zurück nach COM portiert. WinRT-Klassen besitzen nun Metadaten genau wie .NET-Anwendungen. Das Metadatenformat heißt Windows Metadata (WinMD), entspricht aber dem .NET-Metadatenformat. Die Abfrage der Metadaten zur Laufzeit (Reflection) wird unterstützt. Ziel bei WinRT war ein geringer Overhead und maximale Leistung – sicherlich ein Tribut an die größere Menge C++-Entwickler, die Microsoft bisher nicht von Managed Code begeistern konnte.

Beim Design der Klassen hat Microsoft ganz systematisch die asynchronen Prinzipien umgesetzt, die mit der .NET-Klassenbibliothek zum Teil dort nachgerüstet wurden. »Alles, was über 50 Millisekunden dauern könnte, hat nun nur noch asynchrone Methoden«, sagte Aleš Holeček auf der BUILD Konferenz. Microsoft will die Entwickler erziehen, die aus Vereinfachungsgründen bisher den weit leichteren, aber den Thread blockierenden synchronen Weg gewählt haben.

WinRT-Klassen

Das folgende Schaubild gibt einen Überblick über die in WinRT verfügbaren Funktionsbereiche. Neben Standardfunktionen wie Threading, XML und Mehrsprachigkeit erkennt man auch Windows 8-spezifische Funktionen wie *PlayTo*, mit der man die Wiedergabe von Medien auf beliebigen DLNA-Geräten starten kann.

Ebenso wie die .NET-Klassenbibliothek ist WinRT aus Namensräumen aufgebaut z.B. `Windows.ApplicationModel.Background`, `Windows.Data.Xml.Dom`, `Windows.Media.PlayTo`, `Windows.Storage`, `Windows.System.Threading`, `Windows.UI.Xaml.Input`. Als Wurzelnamensraum verwendet Microsoft hier *Windows*, statt wie bisher *System* und *Microsoft*. Einige Betriebssystemfunktionen sind gut versteckt, z.B. die Registry steuert man mit den Klassen im Namensraum `Windows.Storage.ApplicationDataContainer`. In der Dokumentation findet man eine Liste (*http://msdn.microsoft.com/en-us/library/windows/apps/hh464945(v=VS.85).aspx*), wo welche Funktionen der bisherigen Windows-API in der Windows-Runtime zu finden sind. .NET-Entwickler warteten schon lange darauf, dass mehr Funktionen des Betriebssystems (z.B. Zugriff auf Hardware) in Form von .NET-Klassen bereitstehen.

HINWEIS Mit Ausnahme der Oberflächenklassen stehen die WinRT-Klassen auch klassischen .NET-Anwendungen zur Verfügung.

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)

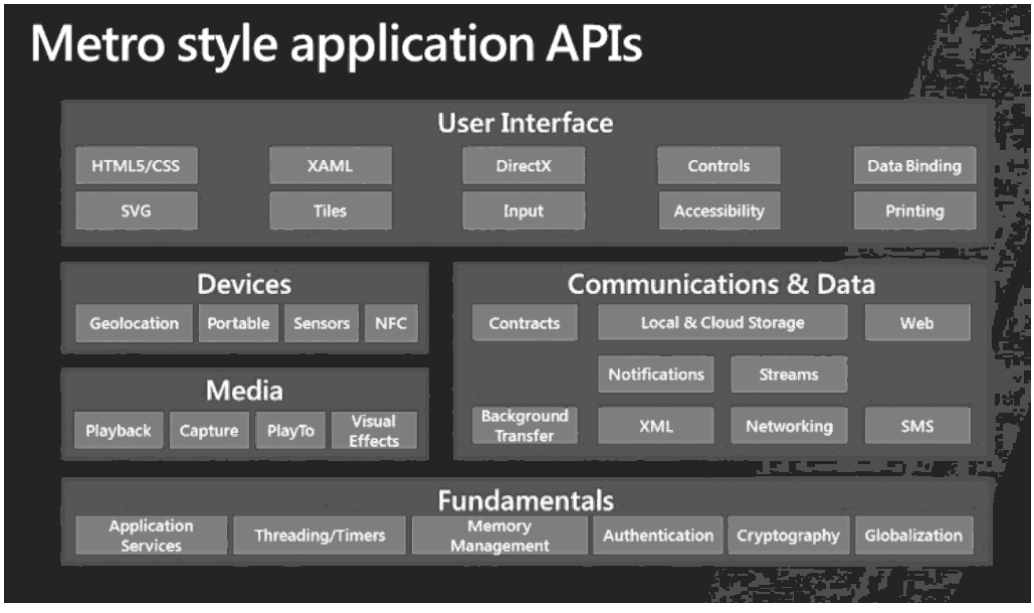

Abbildung 11.4 Funktionsbereiche in WinRT (Quelle: Microsoft BUILD Windows-Konferenz 2011)

Die nächste Abbildung zeigt die in der WinRT verfügbaren Steuerelemente; alle diese Steuerelemente gibt es sowohl für HTML als auch für XAML. Auf den ersten Blick fällt auf, dass nicht alle WPF- und Silverlight-Steuerelemente hier vertreten sind und es zusätzliche Steuerelemente wie `GridView` und `AppBar` gibt. Copy & Paste von XAML aus WPF oder Silverlight zu WinRT (und zurück) wird also in vielen Fällen nicht funktionieren.

Abbildung 11.5 Überblick über Steuerelemente in WinRT (Quelle: Microsoft)

Abbildung von WinRT-Klassen auf .NET-Klassen

Um .NET-Entwickler die Arbeit zu vereinfachen, bildet Microsoft einige Windows-Runtime-Typen auf .NET-Typen ab (Type Mapping). Der Codeeditor zeigt dann den .NET-Typ anstelle des WinRT-Typs, z.B. `IEnumerable<T>` statt `IIterable<T>`. Auch bei primitiven Typen wie `String` findet eine solche Abbildung statt, sodass ein .NET-Entwickler nicht mit der `HSTRING`-Struktur, sondern mit der .NET-Klasse `System.String` arbeiten kann. Die Abbildung erfolgt in einer so genannten Fassaden-Assembly mit der Assembly-Annotation `System.Runtime.CompilerServices.TypeForwardedToAttribute`. Die Fassaden-Assemblys findet man unter *C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\Facades*.

Windows-Runtime-Typ (Namensraum)	.NET Framework-Type (Namensraum)	.NET Framework-Fassaden-Assembly
AttributeUsageAttribute (Windows.Foundation.Metadata)	AttributeUsageAttribute (System)	System.Runtime.dll
AttributeTargets (Windows.Foundation.Metadata)	AttributeTargets (System)	System.Runtime.dll
DateTime (Windows.Foundation)	DateTimeOffset (System)	System.Runtime.dll
EventHandler<T> (Windows.Foundation)	EventHandler<T> (System)	System.Runtime.dll
EventRegistrationToken (Windows.Foundation)	EventRegistrationToken (System.Runtime.InteropServices. WindowsRuntime)	System.Runtime.InteropServices. WindowsRuntime.dll
HResult (Windows.Foundation)	Exception (System)	System.Runtime.dll
IReference<T> (Windows.Foundation)	Nullable<T> (System)	System.Runtime.dll
TimeSpan (Windows.Foundation)	TimeSpan (System)	System.Runtime.dll
Uri (Windows.Foundation)	Uri (System)	System.Runtime.dll
IClosable (Windows.Foundation)	IDisposable (System)	System.Runtime.dll
IIterable<T> (Windows.Foundation.Collections)	IEnumerable<T> (System.Collections.Generic)	System.Runtime.dll
IVector<T> (Windows.Foundation.Collections)	IList<T> (System.Collections.Generic)	System.Runtime.dll
IVectorView<T> (Windows.Foundation.Collections)	IReadOnlyList<T> (System.Collections.Generic)	System.Runtime.dll
IMap<K,V> (Windows.Foundation.Collections)	IDictionary<TKey,TValue> (System.Collections.Generic)	System.Runtime.dll
IMapView<K,V> (Windows.Foundation.Collections)	IReadOnlyDictionary<TKey,TValue> (System.Collections.Generic)	System.Runtime.dll ▶

Windows-Runtime-Typ (Namensraum)	.NET Framework-Type (Namensraum)	.NET Framework-Fassaden-Assembly
IKeyValuePair<K,V> (Windows.Foundation.Collections)	KeyValuePair<TKey,TValue> (System.Collections.Generic)	System.Runtime.dll
IBindableIterable (Windows.UI.Xaml.Interop)	IEnumerable (System.Collections)	System.Runtime.dll
IBindableVector (Windows.UI.Xaml.Interop)	IList (System.Collections)	System.Runtime.dll
INotifyCollectionChanged (Windows.UI.Xaml.Interop)	INotifyCollectionChanged (System.Collections.Specialized)	System.ObjectModel.dll
NotifyCollectionChangedEventHandler (Windows.UI.Xaml.Interop)	NotifyCollectionChangedEventHandler (System.Collections.Specialized)	System.ObjectModel.dll
NotifyCollectionChangedEventArgs (Windows.UI.Xaml.Interop)	NotifyCollectionChangedEventArgs (System.Collections.Specialized)	System.ObjectModel.dll
NotifyCollectionChangedAction (Windows.UI.Xaml.Interop)	NotifyCollectionChangedAction (System.Collections.Specialized)	System.ObjectModel.dll
INotifyPropertyChanged (Windows.UI.Xaml.Data)	INotifyPropertyChanged (System.ComponentModel)	System.ObjectModel.dll
PropertyChangedEventHandler (Windows.UI.Xaml.Data)	PropertyChangedEventHandler (System.ComponentModel)	System.ObjectModel.dll
PropertyChangedEventArgs (Windows.UI.Xaml.Data)	PropertyChangedEventArgs (System.ComponentModel)	System.ObjectModel.dll
TypeName (Windows.UI.Xaml.Interop)	Type (System)	System.Runtime.dll

Tabelle 11.1 Beispiele für WinRT-Klassen, die in .NET-Projekten auf .NET-Klassen abgebildet werden (Quelle: *http://msdn.microsoft.com/en-us/library/windows/apps/hh995050(v=vs.110).aspx*)

Nutzung von WinRT-Klassen in .NET

Mit der neuen Windows-Runtime-API (WinRT) verbinden viele Entwickler immer nur Windows 8-Apps (früher »Metro Apps« genannt), dabei kann man WinRT auch in »klassischen«, mit .NET geschriebenen Desktop-Anwendungen nutzen. WinRT basiert auf einer erweiterten Fassung des Component Object Model (COM) und .NET 4.5 bietet einige Brücken zu diesem neuen COM.

- **Vorteil** In den .NET-Anwendungen stehen dann auch die systemnahen WinRT-Klassen (z. B. für Sensoren, Bluetooth und Windows Search) zur Verfügung. Der Entwickler kann hier viele Hundert Klassen nutzen, für die es bisher nur umständliche Funktionsaufrufe im Windows 32-API gab.
- **Nachteil** Solche Anwendungen laufen dann auch nur unter Windows 8 – zumindest solange nicht Microsoft seine Meinung noch irgendwann ändert und WinRT doch auch für ältere Betriebssysteme anbietet. Zudem kann man nur die nicht-visuellen WinRT-Klassen nutzen. Die Native-Code-basierte XAML-Implementierung ist bisher nicht zugreifbar aus klassischen .NET-Anwendungen.

HINWEIS In Serveranwendungen (ASP.NET, WCF) scheint die WinRT-API nicht lauffähig. Hier erhält man beim jedem Aufrufversuch immer den Fehler: »The parameter is incorrect. (Exception from HRESULT: 0x80070057 (E_INVALIDARG))«

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)

Referenzierung der Windows-Runtime-API

Um WinRT zu nutzen, muss man in einer .NET-Anwendung (z. B. eine Konsolenanwendung, eine Windows Forms-Anwendung oder eine WPF-Anwendung) eine Referenz auf die WinRT-API setzen. In den Vorabversionen von Visual Studio 2012 musste man dazu einfach die WinMD-Dateien im Ordner *c:\Windows\System32\WindowsMetadata* mit *Add Reference* einbinden.

In der endgültigen Version hat Microsoft diesen Weg etwas erschwert. Im Standard erscheint im *Add Reference*-Dialog aber kein Eintrag *Windows Runtime*, selbst auf einem Windows 8-System nicht. Der Versuch, eine solche Referenz auf die WinMD-Dateien zu setzen, scheitert (siehe Abbildung 11.6).

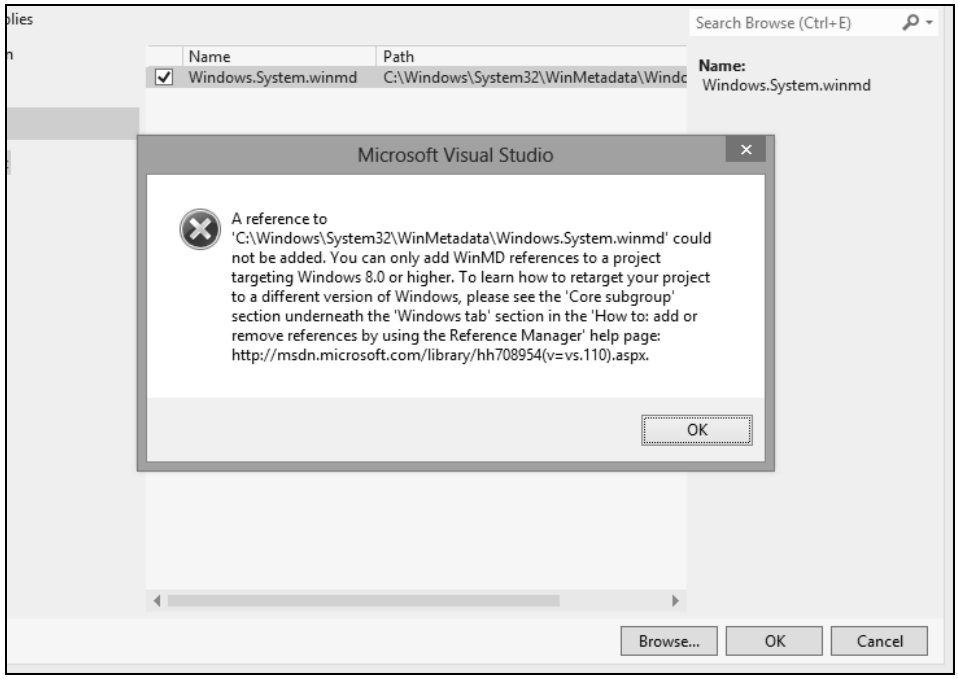

Abbildung 11.6 Fehlermeldung beim Versuch, eine Referenz auf eine WinMD-Datei aus einem .NET-Projekt zu setzen

Die Fehlermeldung weist aber den Weg zur Dokumentation, in der man findet, dass man die Projektdatei (*.csproj* bzw. *.vbproj*) manuell bearbeiten muss. In einem beliebigen Texteditor ergänzt man in dem Tag `<PropertyGroup>` folgenden Eintrag:

```
<TargetPlatformVersion>8.0</TargetPlatformVersion>
```

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProjectGuid>{A22028D2-E741-4334-966F-C0C4590301C2}</ProjectGuid>
    <OutputType>Exe</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>ConsoleApplication1</RootNamespace>
    <AssemblyName>ConsoleApplication1</AssemblyName>
    <TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
    <TargetFrameworkProfile>
    </TargetFrameworkProfile>
    <FileAlignment>512</FileAlignment>
    <SccProjectName>SAK</SccProjectName>
    <SccLocalPath>SAK</SccLocalPath>
    <SccAuxPath>SAK</SccAuxPath>
    <SccProvider>SAK</SccProvider>
    <TargetPlatformVersion>8.0</TargetPlatformVersion>
  </PropertyGroup>
```

Abbildung 11.7 Manuelle Bearbeitung der Projektdatei

Danach lädt man das Projekt neu in Visual Studio und findet dann im Dialog *Add Reference* einen neuen Eintrag *Windows*. Dort gibt es auch nur eine Komponente *Windows* mit der Versionsnummer 255.255.255.255 (☹). Durch Anwählen dieser Komponente stehen nun alle nicht-visuellen WinRT-Klassen für normale .NET-Anwendungen zur Verfügung.

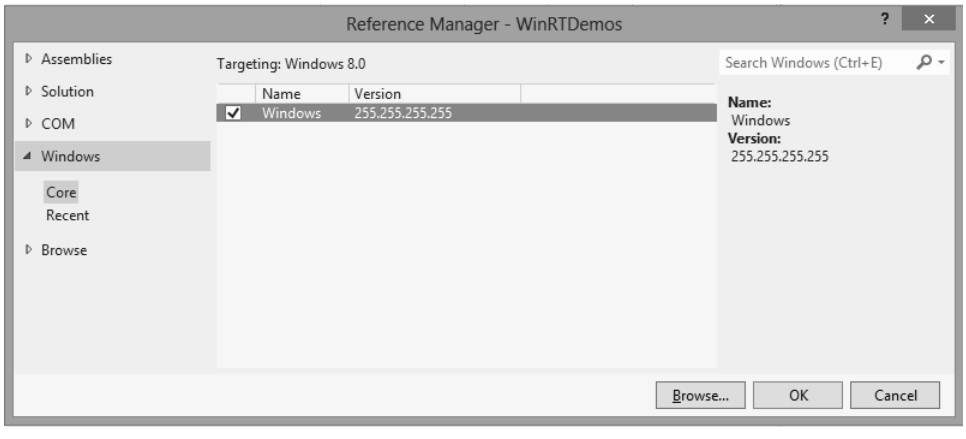

Abbildung 11.8 Korrekte Referenzierung der Windows-Runtime-API in einem .NET-Projekt

Referenzieren der System.Runtime.WindowsRuntime

Für den Aufruf asynchroner Operationen in WinRT benötigt .NET Hilfsklassen, die sich in der `System.Runtime.WindowsRuntime.dll` befinden. Da die meisten Operationen in WinRT asynchron sind, ist also diese Referenz fast immer notwendig.

Ohne diese Referenz erhält man beim Aufruf asynchroner Methoden den Fehler: *»Error 2 'await' requires that the type … have a suitable GetAwaiter method. Are you missing a using directive for 'System'?«*. Der Fehler enthält leider keinen direkten Hinweis auf die `System.Runtime.WindowsRuntime.dll`.

ACHTUNG Die `System.Runtime.WindowsRuntime.dll` findet man nicht in der *.NET*-Sektion im *Add Reference*-Dialog, sondern im Dateisystem unter *C:\Windows\Microsoft.NET\Framework64\v4.0.30319*.

Referenzieren der Fassaden-Assemblys

In den meisten Fällen bei der Arbeit mit der WinRT-API ist auch eine Referenz auf eine oder mehrere Fassaden-Assemblys notwendig. Diese enthalten die Abbildungen von WinRT-Klassen auf .NET-Klassen. Man findet sie unter:

```
C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5\Facades
```

HINWEIS Kurioserweise kann man den Inhalt dieser Fassaden-Assemblys nicht mit dem Object Browser von Visual Studio betrachten. Auch Microsoft IL Disassembly (*ildasm.exe*) zeigt nichts an. Mit einem Drittanbieter-Disassembler wie ILSpy [*http://wiki.sharpdevelop.net/ILSpy.ashx*] sieht man schon etwas mehr (siehe Abbildung 11.9). Zwar verfängt sich ILSpy in einer Rekursion der Typen, aber zeigt zumindest, dass diese Assemblys aus nichts anderem als Typweiterleitungen bestehen.

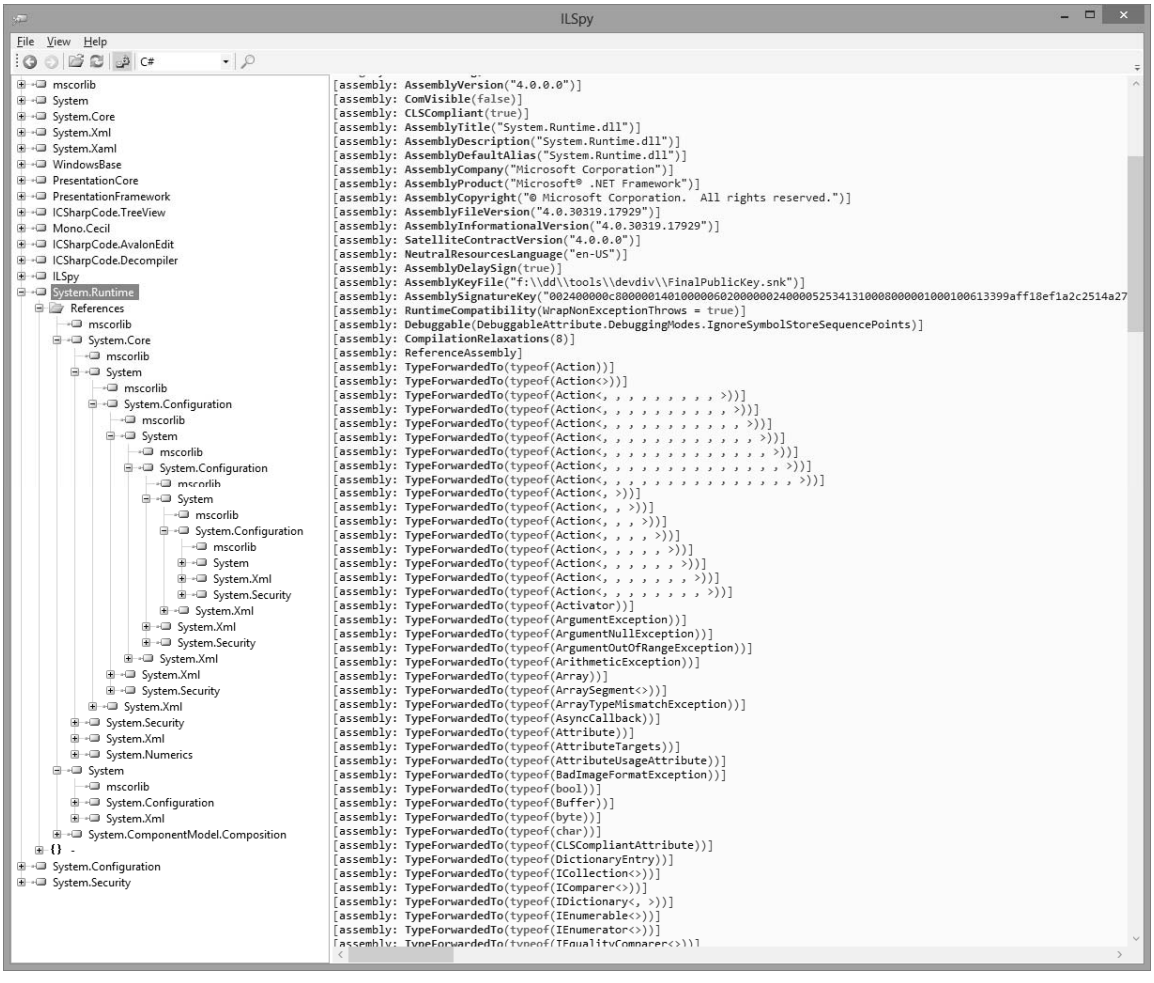

Abbildung 11.9 Typweiterleitungen in der *System.Runtime.dll*

Beispiel 1: Geräteinformationen ausgeben

Das folgende Beispiel zeigt eine Ausgabe der Geräte aus dem Windows Gerätemanager. Zum Einsatz kommen die Klassen `Windows.Devices.Enumeration.DeviceInformationCollection` und `Windows.Devices.Enumeration.DeviceInformation`. Notwendige Fassaden-Assembly ist die `System.Runtime.dll`.

Wie die meisten WinRT-Funktionen ist auch dieser Abruf der Geräteliste mit `FindAllAsync()` eine asynchrone Operation. Hier kann man aber die in C# 5.0 und Visual Basic 11.0 neu eingeführten Sprachkonstrukte `async/await` verwenden. Die asynchronen Operationen mit `async/await` wurden schon in Kapitel 4 genauer erläutert.

```
        static void Main(string[] args)
        {
            Console.WriteLine("Starte");
            //DateiSuchDemo("Mountain Bike", @"w:\dokumente");
            DeviceDemo();
            Console.WriteLine("Warte...");
            while (true) { Console.Write("."); System.Threading.Thread.Sleep(100); }
        }



        private static async void DeviceDemo()
        {
            int count = 0;
            Windows.Devices.Enumeration.DeviceInformationCollection geraeteListe = await↙
 Windows.Devices.Enumeration.DeviceInformation.FindAllAsync();

            foreach ( Windows.Devices.Enumeration.DeviceInformation geraet in geraeteListe)
            {
                count++;
                Console.WriteLine("{0:000} ------------------------------------", count);
                Console.WriteLine("    " + geraet.Name);
                Console.WriteLine("----------------------------------------");
                foreach (var p in geraet.Properties)
                {
                    Console.WriteLine(" " + p.Key + ": " + p.Value);
                }
            }
        }
```

Listing 11.1 Abfrage der vorhandenen Geräte mit WinRT

Beispiel 2: Dateisuche mit Windows (Desktop) Search

Das folgende Beispiel zeigt eine Dateisuche mit Windows (Desktop) Search, implementiert in .NET 4.5 mit C# 5.0 in einer Konsolenanwendung und unter Verwendung der WinRT-Klassen im Namensraum `Windows.Storage.Search` und `Windows.Storage`. Notwendige Fassaden-Assembly ist die `System.Runtime.dll`.

Wie die meisten WinRT-Funktionen ist auch die Windows-Suche nur durch eine asynchrone Operation ausführbar. Die asynchronen Operationen mit `async`/`await` wurden in Kapitel 4 genauer erläutert.

```
        static void Main(string[] args)
        {
            Console.WriteLine("Starte");
            DateiSuchDemo("Mountain Bike", @"w:\dokumente");
            Console.WriteLine("Warte...");
            while (true) { Console.Write("."); System.Threading.Thread.Sleep(100); }
        }


        /// <summary>
        /// Suche ausführen mit Windows Search auf Basis der WinRT-Bibliothek Windows.Storage.Search
        /// </summary>
        private static async void DateiSuchDemo(string Begriff, string Ordner)
        {
            // Suche definieren
            var docs = Windows.Storage.KnownFolders.DocumentsLibrary;
            var queryOptions = new Windows.Storage.Search.QueryOptions();
            queryOptions.FolderDepth = Windows.Storage.Search.FolderDepth.Deep;
            queryOptions.IndexerOption = Windows.Storage.Search.IndexerOption.UseIndexerWhenAvailable;
            queryOptions.UserSearchFilter = "'" + Begriff + @"' folder:" + Ordner; // Advanced Query↙
Syntax (AQS)
            var query = docs.CreateFileQueryWithOptions(queryOptions);

            // Suche starten
            Console.WriteLine("Windows Suche im Ordner " + Ordner + " nach Dateien mit Begriff: " +↙
Begriff);
            Console.WriteLine("Windows Suche beginnt...");

            IReadOnlyList<Windows.Storage.StorageFile> ergebnis = await query.GetFilesAsync();

            // Ergebnisse zeigen
            Console.WriteLine("\nSuche abgeschlossen. Gefundene Dateien: " + ergebnis.Count);
            Console.WriteLine("-----------------------------------------------------");
            foreach (Windows.Storage.StorageFile f in ergebnis)
            {
                Console.WriteLine(f.Name + " ("  + f.FileType + "): " + f.Path);
            }
        }
```

Listing 11.2 Windows-Suche mit WinRT

Pro- und Contra WinRT

Dieses Abschnitt diskutiert Vor- und Nachteile von WinRT

WinRT als neue Betriebssystem-API

Die bisherige Betriebssystem-Programmierschnittstelle von Windows, die Windows 32-API (Win32), ist unzweifelhaft absolut veraltet. Sie ist weder komponenten- noch objektorientiert und passt nicht mehr in die moderne Softwareentwicklungswelt. Eine Ablösung für diese Dinosaurier-API ist längst überfällig. Die Windows-Runtime-API bietet viele Klassen für Funktionen des Betriebssystems (z.B. Geräte, Bluetooth, Windows-Suche, Restart Manager), die bisher nur über die Win32-API oder allenfalls ein paar ebenso schwerfällige COM-Komponenten ansprechbar waren. WinRT kann man nicht nur in Windows Apps, sondern – in vielen Teilen – auch in klassischen Desktop-Anwendungen nutzen.

WinRT bietet die Oberflächenbeschreibungssprache XAML, die Microsoft bereits in WPF und Silverlight verwendet, in einer auf Native Code basierenden und damit mutmaßlich performanteren Variante. Im O-Ton von Microsoft sind die Oberflächen damit »Fast and fluent«, was mit WPF und Silverlight nicht immer gelang. Zudem ist XAML nun auch für Entwickler von Native Code-Anwendungen mit C++ verfügbar, was in WPF und Silverlight kaum möglich war.

Als Alternative zu XAML bietet Microsoft im Rahmen von Windows 8 die Sprache HTML an, die Programmierung erfolgt mit JavaScript und dabei kommen (mit Ausnahme der XAML-UI-Klassen) wieder WinRT-Klassen zum Einsatz. Viele Entwickler, die bisher nicht viel mit der Microsoft-Welt zu tun hatten, können nun mit ihrem bestehenden Knowhow Anwendungen für Windows schreiben. Hier sieht man in Redmond viel Potenzial.

WinRT lässt sich nicht nur in .NET-Sprachen, C++ und JavaScript, sondern auch in anderen Sprachen verwenden (Microsoft spricht von »projizieren« oder »Language Projection«). WinRT kennt viele Konzepte, die es auch in .NET gibt: Vererbung, Konstruktoren, Statische Mitglieder, Metadaten, Exceptions und vieles mehr.

WinRT ist COM

WinRT ist wieder Unmanaged (Native) Code. Noch schlimmer: WinRT basiert auf dem Component Object Model (COM) mit `IUnknown` und – trotz der objektorientierten Überarbeitung – vielem anderem alten Zeugs wie Referenzzählung, `HSTRING`-Strukturen, `HRESULT`s-Fehlercodes und Verzeichnung aller DLLs in der Registry. Auch wenn das WinRT-COM nicht mehr das bedingt objektorientierte COM der 90er Jahre ist, sondern eine neue Version mit vielen .NET-Konzepten, sieht man bei der Programmierung wieder mehr `HRESULT`s-Fehlercodes in COM-Exceptions statt sprechender .NET-Exceptions. WinRT-Komponenten schreiben wieder alles Mögliche in die Registry – unter der Registry-Hölle haben wir doch alle zu lange gelitten und Microsoft hat 2000 gepriesen, dass die Registry nun für die Komponentenregistrierung nicht mehr gebraucht werde! Und die Plattformneutralität ist auch dahin. Compile Once – Run anywhere ist mit dem WinRT-COM gestorben.

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)

Microsoft hatte doch im Jahr 2000 bei der Vorstellung von .NET und die folgenden elf Jahre verkündet, dass .NET eine Ablösung für COM sei. Microsoft hatte gesagt, wir Windows-Anwendungsentwickler sollen .NET und nicht mehr COM programmieren. Microsoft hat gesagt, Managed Code sei robuster, sicherer und dennoch schnell genug. Wir müssen (bis auf Ausnahmefälle wie Treiberprogrammierung) nicht mehr unsere Anwendungen mit Native Code programmieren. .NET sei das EINE Framework für ALLE Plattformen und ALLE Sprachen.

Dass Microsoft da mit gespaltener Zunge redete, wurde über die Jahre durchaus klar, wenn man betrachtete, wie wenig Microsoft selbst .NET für die Produktentwicklung verwendete. Insbesondere die Flaggschiffe Windows und Microsoft Office nutzen bis heute fast gar kein .NET. Begründet wurde dies in Redmond immer damit, dass die Codebasis zu groß sei für eine Migration. Es würde technisch möglich sein, es wären nur wirtschaftliche Gründe, aus denen man nicht migrieren würde. Dass das nicht alles sein kann, wurde uns klar im Zeitalter von Windows Vista. Microsoft hatte 2003 zunächst WinFX (Windows Framework) angekündigt, mit dem die Benutzeroberfläche (mit WinFX-WPF) und andere Teile des Betriebssystems (z. B. das Dateisystem – wir erinnern uns an das Windows File System (WinFS)) in Managed Code mit .NET realisieren sollte. Aber Leistungsprobleme führten zu einer anderen Entwicklung: Die Oberfläche von Vista wurde doch mit Unmanaged C++ erstellt, WinFS wurde ganz eingestellt und die Reste von WinFX gingen 2006 ins .NET Framework 3.0 ein.

Nun, gestehen wir Microsoft zu, dass sie sich geirrt haben. Nehmen wir einmal an, Microsoft glaubte früher wirklich an »das EINE Framework für ALLE Plattformen und ALLE Sprachen«. Und dann stellte man gerade in der Windows Division bei Microsoft fest, dass man zum Beispiel die Windows-Oberfläche nicht performant mit .NET realisieren kann. Irren ist nicht schön, aber muss auch einem Softwareriesen zugestanden werden.

Inkompatibilitäten

Was aber kaum als sachlicher Irrtum zu entschuldigen ist, sondern aus .NET-Entwicklersicht eine schlimme Fehlentscheidung ist, sind die API-Änderungen, die Microsoft im Zuge der Windows Runtime Library vollzogen hat. WinRT besitzt viele Klassen, die es auch schon im .NET Framework gibt, z. B. für Dateisystemoperationen. So gibt es als Pendant zur .NET-Klasse `System.IO.File` die Klasse `Windows.Storage.FileIO`. Statt `WriteAllText()` gibt es dort nun `WriteLinesAsync()`. Ein weiteres Beispiel: `Environment.CurrentManagedThreadId` ersetzt die `ManagedThreadId` in der Klasse `System.Threading.Thread`.

Wenn man Anwendungen für den klassischen Desktop in Windows 8 schreibt, hat man die Wahl zwischen ähnlichen .NET- und WinRT-Klassen. Für Windows Store-Apps sind aber viele .NET-Klassen ausgeblendet im so genannten ».NET 4.5 for Windows Store-Apps«. Hier muss man zwangsweise die WinRT-Klassen verwenden. Und selbst die .NET-Klassen, die es im .NET 4.5 for Windows Store-Apps gibt, sind nicht alle identisch zu den .NET-Klassen des vollständigen .NET Frameworks. So hat der Konstruktor der Klasse `StreamWriter` nur vier statt sieben Überladungen. Eine Methode `Close()` gibt es nicht mehr. Stattdessen gibt es nun ein `Dispose()`. Auch beim UI-Code ist Einiges zu ändern, allein schon, weil sich die Namensräume von `System.Windows.*` auf `Windows.UI.Xaml.*` ändern. Die Liste der Änderungen ist lang (vgl. *http://msdn.microsoft.com/en-us/library/windows/apps/br230302%28v%3DVS.85%29.aspx#UI*). Ein Vergleich zwischen WinRT und Silverlight Version 5 (die vermeintlich am ähnlichsten sein könnten), zeigt ebenfalls große Unterschiede (siehe Abbildung 11.10).

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)

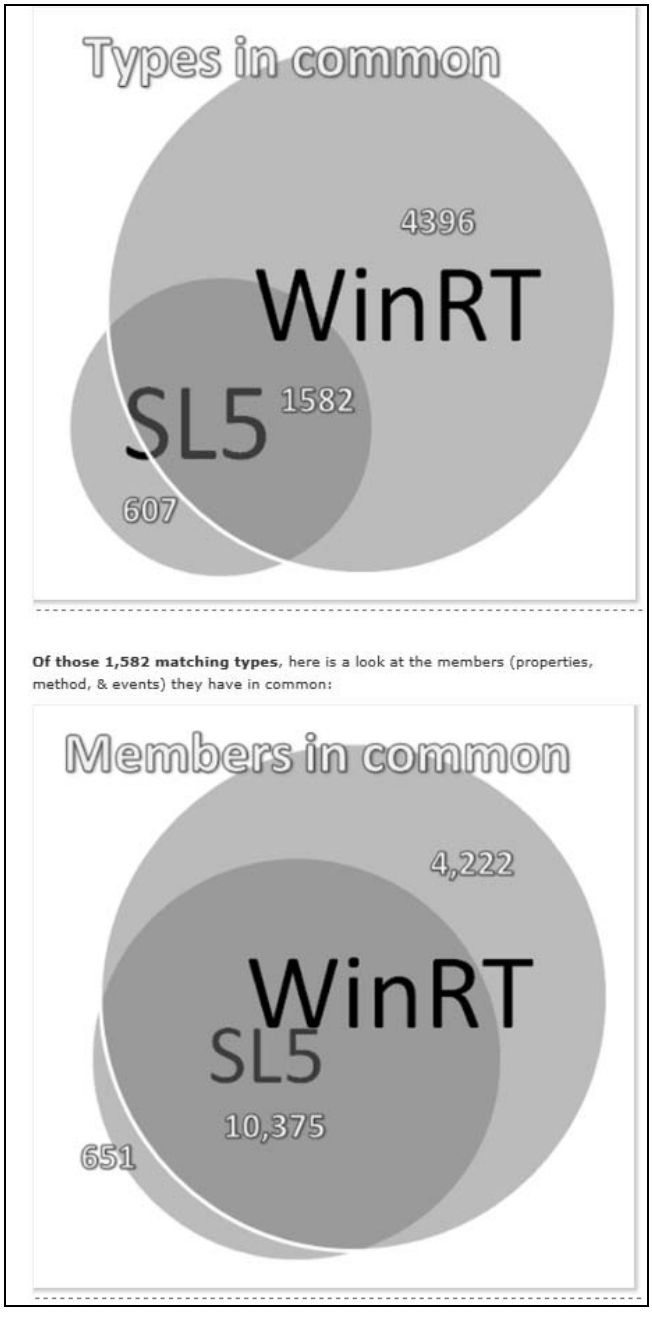

Abbildung 11.10 Unterschiede zwischen der Windows Runtime Library (WinRT) und Microsoft Silverlight Version 5.0
(Quelle: *http://programmerpayback.com/2011/11/17/the-winrt-genome-project/*)

Fragmentierung der .NET-Welt

Die oben erwähnten Inkompatibilitäten bedeuten in der Summe: Bestehenden .NET-Programmcode kann man in den meisten Fällen nicht einfach durch Neukompilieren in die Windows Store-Apps-Welt übernehmen. Und auch zukünftig wird man schwerlich eine gemeinsame Codebasis für klassische Desktop-Anwendungen und Windows Store-Apps besitzen können. Die Fragmentierung des .NET Frameworks (siehe Abbildung 11.11), die schon mit Silverlight begonnen hatte, schreitet voran.

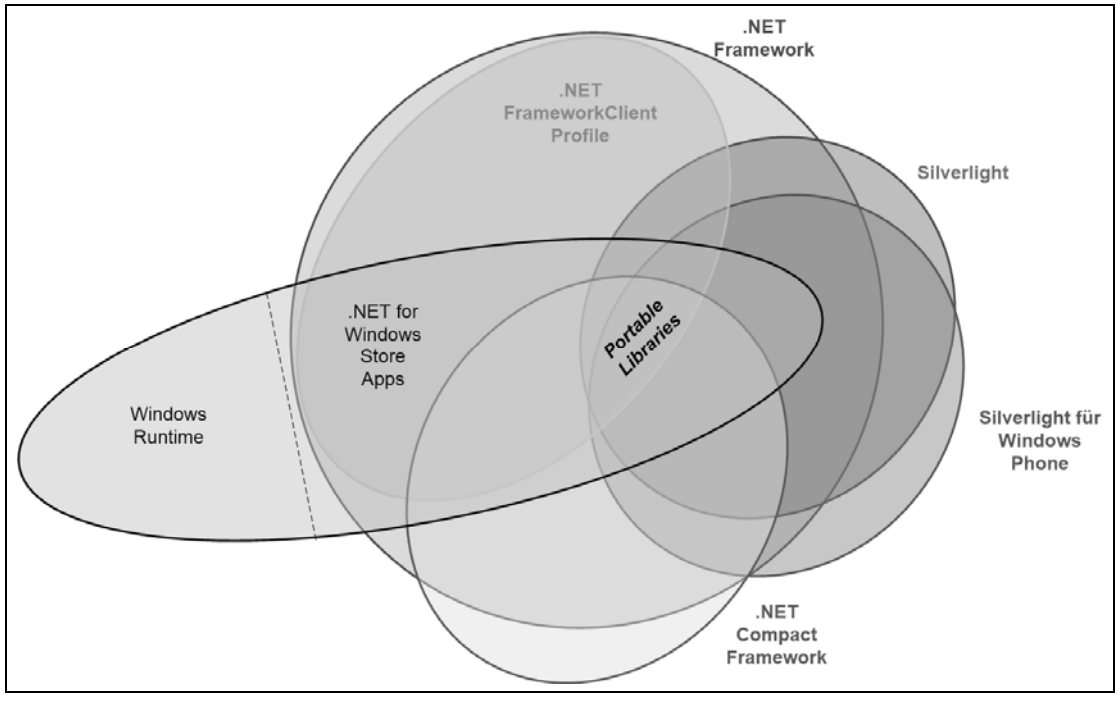

Abbildung 11.11 Fragmentierung der .NET-Welt in zahlreiche Profile mit mehr oder weniger Überschneidung (Hinweis: In diesem Schaubild sind die Überlappungen nicht exakt proportional zur Anzahl der gleichen Klassen und Klassenmitglieder)

Im Wesentlichen wird für reine Datenklassen eine Portable Library, die auf allen Plattformen läuft, realisierbar sein. Wie klein der gemeinsame Nenner aller .NET-fähigen Plattformen geworden ist, zeigt eindrucksvoll ein Visual Studio-Projekt vom Typ *Portable Library*, das man in Visual Studio 2012 anlegen kann (siehe auch Kapitel 2). In so einer Portable Library gibt es eine signifikante Anzahl von Klassen nur in den Namensräumen `System.Collections`, `System.Text`, `System.IO`, `System.Linq`, `System.Diagnostics`, `System.Globalization`, `System.Reflection` und `System.Xml`. Keineswegs sind aber alle Klassen dieser Namensräume verfügbar.

Einige Test zeigen interessante Inkonsistenzen. So sieht man in einem Projekt für eine Windows App nicht mehr die Klasse `System.Threading.Thread`. In einer Portable Library, für die auch *.NET 4.5 for Windows Store-Apps* aktiviert ist, sieht man aber diese Klasse noch mit einigen wenigen Mitgliedern, z.B. `CurrentThread`. Wenn man diese Portable Library in ein Windows App-Projekt einbindet, wird z.B. die Eigenschaftsabfrage `System.Threading.Thread.CurrentThread.ManagedThreadId` tatsächlich ausgeführt. Es ist allerdings zu bezweifeln, ob diese App es durch den Richtlinienfilter in den Microsoft App Store schafft. Meines Ermessens ist dann die Entwicklungsumgebung in Bezug auf die Portable Libraries noch verbesserungsbedürftig, damit die Entwickler gar nicht versehentlich solche Aufrufe machen können.

Microsoft sagte auf der BUILD-Konferenz, dass »bei der Gelegenheit« in einigen APIs »aufgeräumt« worden sei (vgl. *http://msdn.microsoft.com/en-us/library/windows/apps/br230302%28v%3DVS.85%29.aspx#UI*). Ich kann dies nicht nachvollziehen. Ein Entwickler, der vorher mit WPF oder Silverlight entwickelt hat, wird zwar einen Großteil seines Wissens über Visual Studio, C# und XAML in der WinRT wiederverwenden können. Wenige Unterschiede zwischen Desktop-Anwendungen und Windows Apps sind sicherlich gerechtfertigt, aber viele Konzepte aus .NET in WinRT neu zu erfinden, macht keinen Sinn. Selbst ein Sicherheitssystem in Form einer Sandbox besitzt .NET seit Version 1.0. Ein »Aufräumvorgang« und »Redesign« ist extrem kontraproduktiv für alle Softwarehersteller in Hinblick auf die entstehenden Kosten für die Migration und die Pflege von mehreren Codebasen.

WinRT ist nur für Windows 8 verfügbar

Außerdem muss noch ein zentraler Kritikpunkt genannt werden: Die meisten Entwickler von klassischen Desktop-Anwendungen, die zwar prinzipiell die Klassen aus WinRT nutzen können, werden davon vorerst nicht Gebrauch machen können, denn dann läuft ihre Anwendung nur unter Windows 8. Microsoft hat erklärt, dass man WinRT nicht für ältere Betriebssysteme bereitstellen werde. Das halte ich ebenfalls für eine Fehlentscheidung. Eine neue Betriebssystem-API ist gut, aber angesichts der Tatsache, dass viele Unternehmen und Privatleute nur sehr träge auf neue Windows-Versionen umsteigen, hätte Microsoft eine Möglichkeit bieten müssen, die neue API jetzt schon einsetzen zu können, ohne einen großen Nutzerkreis auszuschließen. Mit dieser Weigerung einer Rückportierung auf ältere Windows-Versionen wird es Microsoft vielen Anwendungsentwicklern erst in vielleicht 10 Jahren ermöglichen, WinRT zu nutzen, wenn Windows 9, 10 und 11 die Installationen von Windows 7 weitestgehend verdrängt haben. Zudem: Das schnellere WinRT-XAML kann man aber nicht in klassischen Desktop-Anwendungen nutzen, nur die nicht-visuellen Teile von WinRT sind dort verfügbar.

Kritische Bewertungen WinRT

Ein Kommentar von Dr. Holger Schwichtenberg

Die Windows-Runtime (WinRT) polarisiert die Entwicklergemeinde. Viele sind begeistert von der neuen Betriebsystem-API und den Windows Apps im Stil der neuen Benutzeroberfläche. Aber es gibt auch einigen Grund zur Kritik, denn Microsoft opfert mit WinRT die Kompatibilität und erschwert die Migration.

Den sachlichen Irrtum bezüglich der Leistung von Managed Code kann ich noch akzeptieren, die Änderungen an den APIs nicht. Hätte ich die Entscheidungsgewalt gehabt, dann hätte ich das .NET Framework erweitert und zu DER Betriebssystem-API von allen Windows-Varianten von PC über Tablet bis Phone heraufgestuft, mit den gleichen Klassen für Desktop-Anwendungen und Windows Store-Apps. Ich hätte mit Cross-Compiling dafür gesorgt, dass es das .NET Framework zukünftig in zwei Varianten gibt: Einer Managed Code-Variante und einer Native Code-Variante. Der Entwickler hätte dann die Wahl beim Kompilieren gehabt: Maximale Plattformunabhängigkeit oder maximale Leistung. Das wäre aber nur ein Kompilierungsschalter gewesen. Die Klassen wären in beiden Fällen die gleichen gewesen. Diese .NET Framework-basierte API hätte ich zumindest auch für Windows Vista und Windows 7 und die zugehörigen Serverversionen bereitgestellt.

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)

Es bleibt die Frage, warum sich Microsoft nicht für diese oder eine andere naheliegende Idee, sondern ganz anders entschieden hat. Auf diese Frage gibt es leider keine klare Aussage von Microsoft. Noch während der BUILD-Konferenz in Anaheim im September 2011 haben viele Journalisten versucht, eine Aussage zu diesen und andere Hintergrundfragen zu bekommen. Dabei blockten die Developer-Evangelisten alle Fragen ab mit dem Verweis, dass es ihnen aufgrund einer Unternehmensrichtlinie nicht gestattet ist, Aussagen zu Windows 8 zu machen. Microsoft würde nur über offizielle Kanäle direkt aus Redmond kommunizieren. Auf Anfrage bei der US-Pressestelle kam die Antwort, man habe alles was man sagen wolle, in den Vorträgen der BUILD-Konferenz und der Dokumentation gesagt. Zitat: »Unfortunately, we are not going to be able to dive deeper into this questions beyond the general statement we provided previously on WinRT.«. Eine solche extrem eingeschränkte Kommunikationspolitik kennt man von anderen Herstellern, aber bisher nicht von Microsoft. Selbst MVPs bekamen keine weiteren Informationen.

Ich will mich darüber nicht beklagen. Microsoft versorgt mich als MVP zu anderen Themen extrem gut mit Informationen. Ich kann akzeptieren, dass Microsoft in diesem Fall schweigen möchte. Man tut sich vermutlich nicht leicht mit einer für alle plausiblen Antwort. Microsoft muss sich dann nur nicht wundern, dass es viele wilde Spekulationen über die Beweggründe in der Entwicklergemeinde gibt. Eine dieser Spekulation ist, dass die jetzige Entwicklung die Ausgeburt eines hausinternen Machtkampfes zwischen der Windows Division (WinDiv) und der Developer Division (DevDiv) ist (vgl. z.B. *http://arstechnica.com/microsoft/news/2011/06/windows-8-for-software-developers-the-longhorn-dream-reborn.ars*), den die WinDiv für sich entschieden hat, weil dort mehr Geld verdient wird. Ich denke, das könnte stimmen. Aber Beweise habe ich nicht.

Dass Microsoft sich ganz anders entschieden hat, sehe ich mit einem weinenden und einem lachenden Auge. In meiner Rolle als Softwareentwickler, der seit 2001 ganz auf .NET setzt, finde ich die Lösung sehr enttäuschend. In meiner Rolle als Berater und Trainer bin ich natürlich begeistert über die vielen Consulting- und Trainingsleistungen, die unsere Kunden benötigen werden.

Dr. Holger Schwichtenberg, Manfred Steyer, Dr. Joachim Fuchs: Microsoft .NET 4.5 Update,
Microsoft Press 2013 (ISBN 978-3-86645-468-2)