Marcus Heege

MIX AND MATCH

Windows Forms in MFC-Anwendungen über C++-Interop integrieren

Obwohl die Erstveröffentlichung des Microsoft .NET Frameworks fast fünf Jahre zurückliegt und Version 2.0 gerade erschienen ist, bestehen immer noch viele C++-Anwendungen aus reinem nicht verwaltetem Code. Doch unter C++-Entwicklern wächst das Interesse am .NET Framework rapide. Und das aus guten Gründen: Viele zukünftige Windows-APIs werden auf dem .NET Framework basieren. Das gilt sicherlich für die WinFX-Komponenten Windows Presentation Foundation, Windows Communication Foundation und Windows Workflow Foundation.

Dennoch möchten viele C++-Entwickler native APIs und keine Wrapper um vorhandene APIs verwenden. Wrapper stehen oftmals im Ruf, fehlerhaft, langsam und unflexibel zu sein. Abgesehen davon wäre es wirklich schwierig, riesige APIs wie WinFX so abzubilden, dass sie sich von nativem Code aus ansprechen lassen.

In der Mehrzahl der Fälle lassen sich C++-Anwendungen mit Features aus dem .NET Framework viel einfacher erweitern, als die meisten Entwickler glauben. Visual C++ umfasst ein Feature namens C++-Interop, manchmal auch als „It Just Works" oder IJW bezeichnet. Damit können Sie nahtlos .NET-basierten Code in Ihren vorhandenen C++-Quellcode integrieren.

C++-Interop basiert auf zwei Hauptfeatures. Erstens können Sie Ihren vorhandenen C++-Code nach MSIL (Microsoft Intermediate Language) mit dem Schalter */clr* des C++-Compilers kompilieren. Ihrem Code stehen dann .NET-Features wie Garbage Collection, Sandbox-Sicherheit und Typen aus der riesigen .NET-Framework-Basisklassenbibliothek offen.

Das andere Feature heißt Mixed-Mode (gemischter Modus) und ist gleichermaßen wichtig. Es bezieht sich auf die Kombination von verwaltetem und nicht verwaltetem Code. Wenn Sie C++-Code nach MSIL kompilieren, erzeugt der Compiler verwaltete Objektdateien, die MSIL-Code enthalten, anstelle der normalen, nicht verwalteten Objektdateien, die nativen Assembly-Code enthalten. Der Linker ist in der Lage, sowohl verwaltete als auch nicht verwaltete Objektdateien als Eingabe zu verarbeiten. Wenn der Linker mindestens eine verwaltete Eingabe entdeckt, erzeugt er eine .NET-Assembly, die sowohl verwalteten Code als auch nicht verwalteten Code enthalten kann. Aus diesem Grund spricht man vom Mixed-Mode (siehe Abbildung 1). Dieses Feature ist äußerst wichtig für Leistungsoptimierungen, weil Sie damit die Anzahl der Übergänge zwischen den verwalteten und nicht verwalteten Räumen drastisch verringern können.

C++-Interop als treuer Begleiter

Meine Erfahrung hat gezeigt, dass C++-Interop sehr zuverlässig ist. Und trotz einiger Beschränkungen ist es eine lohnende Ergänzung für Ihre Entwickler-Toolbox. Um die Funktionsweise zu verstehen, sollten Sie sich klar machen, dass C++-Interop ein Entwurfsfeature des .NET Frameworks war. Die CLI (Common Language Infrastructure), die die Basisspezifikation des .NET Frameworks darstellt, ist von vornherein für dieses Feature ausgelegt. In der Tat hat man viele Aspekte der CLI einfach deshalb definiert, weil C++-Interop sie verlangt.

Zum Beispiel unterstützen verwaltete Metadaten globale Funktionen. Jedoch verlangen C# und die meisten anderen .NET-Sprachen, dass alle Methoden innerhalb des Gültigkeitsbereichs einer

kurz & bündig

Folgende Technologien werden verwendet:

C++, MFC, .NET Framework, Windows Presentation Foundation

- **Wie C++-Interop funktioniert**
- **Wann Sie C++-Interop in Ihren Projekten einsetzen**
- **Windows-Forms-Steuerelemente in MFC-Anwendungen einsetzen**
- **Von Windows Forms zu WPR-Ansichten übergehen**

Klasse liegen. Und zwar deshalb, weil C++-Code, der auf .NET-basierten Code abgebildet werden sollte, globale Funktionen umfassen kann – und um diesen C++-Code auf verwalteten Code abzubilden, brauchen Metadaten ein äquivalentes Konzept.

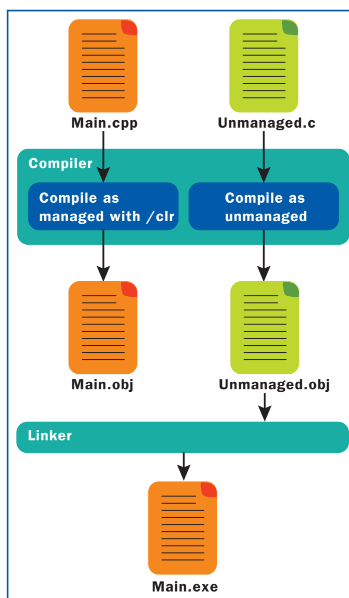

Abb. 1: Gemischten Code kompilieren und linken

Der MSIL-Befehlssatz ist ebenfalls mit C++-Interop im Hinterkopf entstanden. Um C++-Code auf MSIL-Code abzubilden, muss MSIL alle allgemeinen Operationen der Datenmanipulation wie die typischen booleschen, arithmetischen und Gleitkommaoperationen unterstützen. MSIL unterstützt auch Zeiger-basierten Zugriff auf virtuellen Speicher. Das ist der Schlüssel zur Verwendung nativer Typen in verwaltetem Code. Wenn Ihr C++-Code auf ein Feld eines nativen Objekts zugreift, kann der C++/CLI-Compiler MSIL-Code ausgeben, der die Adresse des nativen Objekts auf den Stack legt, dazu den Offset des Felds addiert und die resultierende Adresse für den Zugriff auf das Feld verwendet. Dies ist alles möglich, weil der MSIL-Befehlssatz über Operationen verfügt, die auf virtuellen Speicher über Adressen zugreifen können.

Ein anderes C++-spezifisches Feature der IL-Sprache ist der *CALLI*-Befehl. Mit ihm lassen sich native Funktionen über Funktionszeiger aufrufen. Dieses Feature wird genutzt, um virtuelle Funktionen von nativen Typen aufzurufen. Es ist wichtig, weil COM-Schnittstellen native Typen mit virtuellen Funktionen sind. Diese Methode der COM-Interoperabilität besitzt viele Vorteile gegenüber den von vielen anderen Sprachen verwendeten alternativen Techniken der COM-Interoperabilität.

Alle Quelldateien, die mit der Option /clr kompiliert werden, können Typen aus der .NET-Framework-Basisklassenbibliothek verwenden. Mit dem größten Teil der Basisklassenbibliothek arbeiten Sie so, wie Sie es gewohnt sind.

Schließlich ist der Entwurfseinfluss erwähnenswert, der allen Entwicklern, die .NET einsetzen, am meisten ins Auge springt: das Dateiformat für Komponenten. Während Java sein eigenes Dateiformat (die *.class*-Datei) definiert hat, verwendet das .NET Framework das standardisierte PE-Dateiformat. Demzufolge haben .NET-basierte Komponenten die geläufigen Dateierweiterungen DLL und EXE. Die Tatsache, dass PE-Dateien immer noch verwendet werden, ist wesentlich, um Assemblys im Mixed-Mode zu unterstützen.

Wann man C++-Interop verwendet

Um zu beurteilen, ob C++-Interop für Ihre Projekte geeignet ist, sollten Sie auch wissen, wie dieses Feature die Erstellungsweise Ihrer Projekte, die Compilereinstellungen, sowie die Ausgabe dieses Buildprozesses beeinflusst.

Wenn Sie mit der Option */clr* kompilieren, hängt Ihr Code implizit von der multithreaded DLL-Version der C-Laufzeitbibliotheken (C Runtime Libraries, CRT) ab. Demzufolge müssen alle Dateien in Ihrem Projekt – selbst die, die ohne */clr* kompiliert wurden – die multithreaded DLL-Version der CRT verwenden. Dies ist auch wichtig für MFC-Projekte. Die statische MFC-Bibliothek hängt von der statischen CRT-Bibliothek ab, die ihrerseits nicht kompatibel mit der */clr*-Kompilierung ist. Deshalb müssen Sie sicherstellen, dass Ihre MFC-Projekte die DLL-Version der MFC einbinden.

Es kann auch Probleme im Hinblick auf die Ausführungszeit geben. Das Initialisieren der CLR braucht eine bestimmte Zeit und die JIT-Kompilierung bringt ebenfalls einen gewissen Overhead mit sich. Deshalb müssen Sie mit einer etwas längeren Startphase rechnen. Allerdings sollte das für die meisten Projekte kein großes Problem darstellen.

Methodenaufrufe mit Übergängen von verwaltetem zu nicht verwaltetem Code brauchen länger als normale Methodenaufrufe. Wie bereits erwähnt, können Sie die Anzahl der Übergänge zwischen verwaltetem und nicht verwaltetem Code beträchtlich verringern, wenn Sie explizit festlegen, welche spezifischen Teile Ihrer Anwendung zu verwaltetem Code kompiliert werden sollen. Die Aufrufkonvention *__clrcall* ist ein weiteres wichtiges Optimierungsfeature, das die Anzahl der Übergänge beträchtlich reduzieren kann, doch gehört dies nicht zu den Themen dieses Artikels.

In den meisten Fällen ist der JIT-kompilierte Code an sich nicht die Ursache der spürbaren Leistungseinbußen. In der Tat gibt es einige Optimierungen, die ein JIT-Compiler vornehmen kann, die aber beim C++-Compiler/Linker nicht möglich sind. Zum Beispiel lässt sich mit einem JIT-Compiler der Code für die Prozessorarchitektur auf dem Zielcomputer optimieren. Im Gegensatz zum C++-Compiler beherrscht ein JIT-Compiler auch komponentenübergreifendes Inlining. Diese Optimierung kann recht effizient sein, weil der mit komponentenübergreifendem Inlining kompilierte Code für weitere Optimierungen offen ist.

Abgesehen von den hier erwähnten Problemen gibt es noch andere Einflussbereiche, auf die ich hier aber nicht im Detail eingehe. Die meisten Probleme lassen sich leicht auflösen, wenn man Compiler- und Linker-Einstellungen in der richtigen Weise modifiziert. Die optimalen Compiler-Einstellungen liegen nicht immer auf der Hand. Einige dieser Einstellungen aktivieren Features, die für C++-Interop-Kompilierung obligatorisch sind, während andere Einstellungen Features deaktivieren können, die nicht mit der */clr*-Kompilierung kompatibel sind oder die für MSIL in abweichender Form gelöst werden.

Windows-Forms-Steuerelemente in MFC-Anwendungen

Alle Quelldateien, die mit der Option */clr* kompiliert werden, können Typen aus der .NET-Framework-Basisklassenbibliothek verwenden. Mit dem größten Teil der Basisklassenbibliothek arbeiten Sie so, wie Sie es gewohnt sind. Nur wenige Fälle verlangen besondere Aufmerksamkeit. Ein derartiges Beispiel ist das Integrieren von Windows-Forms-Steuerelementen in MFC-Anwendungen.

Man kann nicht *System::Windows::Forms::Control^* übergeben, wenn ein *CWnd** erwartet wird. Allerdings haben das Windows-

Forms-API und die MFC ein gemeinsames Erbe: die gute alte *User32.dll*. Beide APIs halten einige Hintertüren zu den *HWND*s offen. Während die MFC-Klasse *CWnd* einen Umwandlungsoperator auf *HWND* mitbringt, implementiert die Windows-Forms-Steuerelementklasse die Schnittstelle *IWin32Window* aus dem Namespace *System::Windows::Forms*, um das Fensterhandle offen zu legen:

```
public interface IWin32Window {
  property IntPtr Handle {
    IntPtr get ();
  }
};
```

Allerdings genügt es oftmals nicht, ein Fensterhandle von einem Fensterobjekt zu erhalten. Verschiedene Klassen der MFC unterstützen die Behandlung von Windows-Forms-Steuerelementen in unterschiedlichen Szenarios. Diese Klassen sind in *afxwinforms.h* deklariert.

Die wichtigste dieser Klassen ist *CWinFormsControl*, die im Namespace *Microsoft::VisualC::MFC* definiert ist. Die Klasse erlaubt es Ihnen, ein Windows-Forms-Steuerelement in einer von *Dialog* oder *CDialog* abgeleiteten MFC-Klasse zu hosten. Es ist eine auf einer Vorlage basierende Klasse mit einem Vorlagenargument, über das der Typ des zu hostenden Windows-Forms-Steuerelements übergeben wird. Das kann entweder der konkrete Typ sein, den Sie später instanziieren möchten, oder eine Basisklasse des konkreten Typs. Die Vorlage *CWinFormsControl* ist recht einfach und weist nur zwei interessante Merkmale auf: Sie erbt von *CWnd* und sie stellt mehrere Überladungen einer selbsterklärenden Methode – *CreateManagedControl* – bereit.

Wenn Sie ein Windows-Forms-Steuerelement in einer von *Dialog* oder *CDialog* abgeleiteten MFC-Klasse hosten möchten, definieren Sie eine Membervariable des Typs *CWinFormsControl<TWinFormsCtrl>* in Ihrer Klasse und wählen die passende Überladung von *CreateManagedControl*. Wenn Sie eine von *CDialog* abgeleitete Klasse implementieren, kommt *OnInitDialog* als Kandidat für das Erstellen des verwalteten Steuerelements infrage. Für die meisten anderen Fälle ist *OnCreate* (der Nachrichtenhandler für *WM_CREATE*) die beste Option. Eine der Überladungen ist speziell für Dialogklassen interessant. Diese Überladung erwartet, dass Sie ein statisches Platzhaltersteuerelement zu einer Dialogressource hinzufügen. Platzieren Sie das statische Steuerelement so, dass es die Position und Größe besitzt, die das Windows-Forms-Steuerelement haben sollte, und geben Sie ihm eine eindeutige Kennung (ID), wie es in Abbildung 2 zu sehen ist. Das gehostete Steuerelement instanziieren Sie durch Aufruf von *CreateManagedControl* (wie in Listing 1 gezeigt). Alternativ können Sie eine spezielle Variante des Dialogfeld-Datenaustauschs (Dialog Data Exchange, DDX) wie in Listing 2 verwenden.

Tabelle 1: Schnittstellen

System::Windows::Forms::Control
IOleObject
IOleControl
IOleInPlaceObject
IOleInPlaceActiveObject
IOleViewObject
IOleViewObject2
IPersist
IPersistStreamInit
IPersistPropertyBag
IPersistStorage
IQuickActivate

Beachten Sie bitte, dass diese Art des Hostings von Windows-Forms-Steuerelementen in MFC-Anwendungen noch ein anderes gemeinsames Erbe von Windows Forms und MFC verwendet: ActiveX-Steuerelemente. Wie Tabelle 1 zeigt, implementiert *System::Windows::Forms::Control* viele Schnittstellen, die OLE- und ActiveX-Steuerelement-Veteranen vertraut sein sollten. Innerhalb von *Create-ManagedControl* wird das Windows-Forms-Steuerelement über den *gcnew*-Operator instanziiert und auch genauso wie ein normales ActiveX-Steuerelement in-place aktiviert.

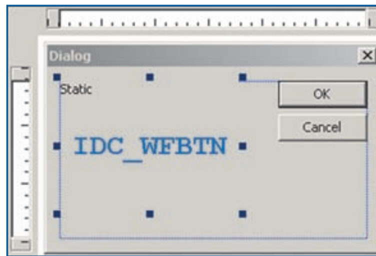

Abb. 2: Ein ActiveX-Steuerelement verwenden

Weiterhin ist zu beachten, dass *CWinFormsControl* den Operator -> überlädt, damit das gehostete Steuerelement zurückgegeben wird. Dadurch können Sie die *Text*-Eigenschaft der Schaltfläche über eine einfache Zuweisung initialisieren:

```
m_wfBtn.Text = "Click me!";
```

Eine sauberere Lösung wäre es natürlich, die Eigenschaften eines Steuerelements im Eigenschaftenfenster zu modifizieren. Wenn Sie aber mit der momentan nur beschränkten Entwicklerunterstützung leben können, steht Ihren altvertrauten MFC-Dialogfeldern die ganze Welt der Windows-Forms-Steuerelemente offen.

Listing 1: Ein gehostetes Steuerelement erstellen

```
using Microsoft::VisualC::MFC;
using System::Windows::Forms;

class CMyDialog : public CDialog
{
public:
    CMyDialog() : CDialog(CMyDialog::IDD, NULL) {}

    enum { IDD = IDD_MYDIALOG };

private:
    CWinFormsControl<Button> m_wfBtn;

public:
    BOOL OnInitDialog() {
        CDialog::OnInitDialog();

        return this->m_wfBtn.CreateManagedControl(
            WS_VISIBLE | WS_CHILD, IDC_WFBTN, this));
    }
}
```

Listing 2: Ein Steuerelement mit DDX hosten

```
using Microsoft::VisualC::MFC;
using System::Windows::Forms;

class CMyDialog : public CDialog
{
public:
    CMyDialog(CWnd* pParent = NULL): CDialog(CMyDialog::IDD, pParent) {}

    enum { IDD = IDD_MYDIALOG };

private:
    CWinFormsControl<Button> m_wfBtn;

protected:
    virtual void DoDataExchange(CDataExchange* pDX) {
        DDX_ManagedControl(pDX, IDC_WFBTN, this->m_wfBtn);
    }
};
```

Steuerelementereignisse behandeln

Es gibt noch einen anderen Aspekt, der hier zur Sprache kommen soll. Bisher wird das *Click*-Ereignis einer Schaltfläche nicht behandelt. Da Sie ein Windows-Forms-Steuerelement haben, das wie ein ActiveX-Steuerelement gehostet wird, gibt es zwei Optionen für die Ereignisbehandlung: vom ActiveX-Steuerelement bereitgestellte Verbindungspunkte und Ereignismember des Windows-Forms-Steuerelements. Die Verbindungspunktvariante ist nur erfolgreich, wenn das Windows-Forms-Steuerelement darauf vorbereitet ist, COM-basierte Ereignisse zu unterstützen. Bei den meisten Windows-Forms-Steuerelementen ist das nicht der Fall, sodass die Ereignisbehandlung in .NET-Manier hier die einzige Option ist. Ereignisse von verwalteten Klassen verkörpern eine spezielle Art von Typmember, der die Registrierung und Deregistrierung von Ereignishandler-Delegaten erlaubt. Die folgende Zeile zeigt, wie Sie das *Click*-Ereignis der Schaltfläche registrieren:

```
m_wfBtn.Click += <an event handler delegate for the click event>
```

Um einen solchen Delegaten bereitzustellen, ist eine Funktion mit der folgenden Syntax erforderlich:

```
void EventHandler(Object^ sender, EventArgs^ e);
```

Leider genügt es nicht, eine derartige Methode in der von *CDialog* abgeleiteten Klasse zu implementieren. Zielfunktionen von Delegaten müssen Memberfunktionen von verwalteten Klassen sein und die von *CDialog* abgeleitete Klasse ist eine native Klasse. Visual C++ bietet Ihnen mit der Header-Datei *\Msclr\Event.h* eine generische Lösung für dieses Problem. Die Definitionen in dieser Header-Datei können Sie nutzen, wenn Sie ein .NET-Ereignis mit einer Methode einer nativen C++-Klasse behandeln möchten. Wie dieser Artikel später noch zeigt, können Sie diesen Header auch verwenden, wenn Sie die Visuals-Merkmale der Windows Presentation Foundation integrieren. Da diese Header-Datei in so vielen unterschiedlichen Szenarios hilfreich ist, bietet es sich an, die von ihr bereitgestellte Lösung eingehend zu diskutieren.

Um die Ereignisse des Windows-Forms-Steuerelements zu behandeln, könnten Sie eine verwaltete Klasse implementieren, die den Nachrichtenhandler enthält. Mit C++-Interop lässt sich dies elegant realisieren, weil diese verwaltete Klasse in die von *CDialog* abgeleitete Klasse eingeschachtelt werden kann:

```
class CMyDialog : public CDialog {
    ref class nested_managed_class {
        void OnWFBtnClick(Object^ sender, EventArgs^ e) {
            // handle the button"s click event here
        }
    };

    // other members
};
```

Das *Click*-Ereignis der Schaltfläche lässt sich am bequemsten verarbeiten, wenn man diesen Methodenaufruf an eine Memberfunktion der von *CDialog* abgeleiteten Klasse weiterleitet. Die Methode, die schließlich das *Click*-Ereignis behandelt, kann leicht auf andere Member der abgeleiteten Klasse zugreifen – genauso wie man es von Ereignishandlern erwartet. Die verwaltete Klasse, die den Delegaten enthält, ist letztlich ein einfacher Proxytyp. Obwohl man einen derartigen Proxytyp und seine Proxy-Ereignishandler mit Makros erzeugen kann, die in *Event.h* definiert sind, werfen wir zuerst einen Blick auf eine manuelle Definition (siehe Listing 3).

Wie der Code zeigt, besitzt der Delegat-Proxy einen Konstruktor, der ein *CMyDialog**-Argument übernimmt. Ein derartiger Konstruktor ist notwendig, um Methodenaufrufe an die von *CDialog* abgeleitete Klasse weiterzuleiten. Dieser Delegat-Proxytyp ließe sich für alle Ereignishandler aller Windows-Forms-Steuerelemente der Dialogfeldklasse einsetzen. Es ist nicht möglich, eine Membervariable des *delegate_proxy_type* zur abgeleiteten Klasse hinzuzufügen, da native Klassen keine Handles auf Ob-

Listing 3: Die Klasse CMyDialog

```
class CMyDialog : public CDialog
{
public:
    ref class delegate_proxy_type;
    gcroot<delegate_proxy_type^> m_gc_managed_native_delegate_proxy;
    delegate_proxy_type^ get_proxy(CMyDialog* pNativeTarget) {
        if((delegate_proxy_type^)m_gc_managed_native_delegate_proxy ==
            nullptr)
        {
            m_gc_managed_native_delegate_proxy =
                gcnew delegate_proxy_type(pNativeTarget);
        }
        return (delegate_proxy_type^)m_gc_managed_native_delegate_proxy;
    }

    ref class delegate_proxy_type {
        CMyDialog* m_p_native_target;
    public:
        delegate_proxy_type(CMyDialog* pNativeTarget) :
            m_p_native_target(pNativeTarget) {}
        void OnWFBtnClick(Object^ sender, EventArgs^ e) {
            this->m_p_native_target->OnWFBtnClick(sender, e);
        }
    };

    void OnWFBtnClick(Object^ sender, EventArgs^ e) {
        this->m_wfBtn->Text = "Thanks for clicking";
    }

    // other members
};
```

Listing 4: Die vereinfachte CMyDialog-Klasse

```
class CMyDialog : public CDialog
{
public:
    CMyDialog(CWnd* pParent = NULL): CDialog(CMyDialog::IDD, pParent) {}

    enum { IDD = IDD_MYDIALOG };

private:
    CWinFormsControl<Button> m_wfBtn;

protected:
    virtual void DoDataExchange(CDataExchange* pDX) {
        DDX_ManagedControl(pDX, IDC_WFBTN, this->m_wfBtn);
    }

public:
    BEGIN_DELEGATE_MAP(CMyDialog)
        EVENT_DELEGATE_ENTRY(OnWFBtnClick, Object^, EventArgs^)
    END_DELEGATE_MAP()

    void OnWFBtnClick(Object^ sender, EventArgs^ e) {
        this->m_wfBtn->Text = «Thanks for clicking»;
    }

public:
    virtual BOOL OnInitDialog(){
        CDialog::OnInitDialog();

        this->m_wfBtn->Click += MAKE_DELEGATE(System::EventHandler,
            OnWFBtnClick);
        return TRUE;
    }
};
```

jekte, die der Garbage Collection unterliegen, als Felder haben können. Mithilfe der *gcroot*-Vorlage können Sie diese Einschränkung hier umgehen. Demzufolge ist eine Membervariable vom Typ *gcroot<delegate_proxy_type>* erforderlich. Allerdings bleibt noch eine Frage: Wie könnte die auf *gcroot* basierende Referenz auf den Delegat-Proxytyp in der richtigen Weise initialisiert werden? Dies lässt sich durch eine separate Methode behandeln, die aufgerufen wird, wenn ein Handle für *delegate_proxy_type* erforderlich ist.

Mit dem Code von Listing 3 lässt sich ein Delegat leicht registrieren, wie es im folgenden Beispiel zu sehen ist:

```
virtual BOOL OnInitDialog() {
    CDialog::OnInitDialog();

    this->m_wfBtn->Click +=
        gcnew System::EventHandler(get_proxy(this),
        &delegate_proxy_type::OnWFBtnClick);

    return TRUE;
}
```

Diesen ganzen Code nur zu schreiben, um Ereignisse eines Windows-Forms-Steuerelements in einer von *CDialog* abgeleiteten Klasse zu behandeln, wäre zu viel Aufwand. *Event.h* enthält Makros und Vorlagen, um das Gleiche (und noch mehr) mit wesentlich weniger Zeilen zu erreichen, wie es Listing 4 zeigt.

Jede neue Version der MFC bringt mindestens eine neue Abbildung. Wie Sie sehen können, führt die aktuelle Version die Delegat-Zuordnung ein. Das Makro *BEGIN_DELEGATE_MAP* definiert den *delegate_proxy_type*, der die Delegat-Ziele enthält. Jede *EVENT_DELEGATE_ENTRY*-Zeile fügt eine derartige Zielmethode in die verwaltete Klasse ein. *END_DELEGATE_MAP* beendet einfach die verwaltete Klasse. Schließlich instanziiert *MAKE_DELEGATE* einen Delegaten, der auf die Zielmethode von *delegate_proxy_type* verweist. Praktisch lässt sich mit den Makros noch mehr bewerkstelligen als das, was eben erwähnt wurde. Die Makros sind sogar auf Szenarios vorbereitet, in denen die verwaltete Klasse Ereignisse auslöst, die an ein natives Objekt, das nicht mehr existiert, zu senden sind.

Steuerelemente als Dialogfelder

Ein anderes einfaches und doch effizientes Hilfsmittel ist die Vorlage *CWinFormsDialog*. Man kann sie einsetzen, um ein Windows-Forms-Steuerelement als modales oder nicht modales Dialogfeld anzuzeigen. Diese Klasse ist recht einfach. Sie erweitert *CDialog*, um das Standardverhalten von MFC-Dialogfeldern zu realisieren. Abgesehen davon verwendet sie eine *CWinFormsControl*-Membervariable, um das Windows-Forms-Steuerelement zu hosten, initialisiert die Titelleiste des Dialogfelds mit dem Wert der *Text*-Eigenschaft des gehosteten Steuerelements, legt die anfängliche Größe des Dialogfelds fest, damit das Steuerelement perfekt passt, und stellt sicher, dass die Größe des gehosteten Steuerelements immer angepasst wird, wenn sich die Größe des übergeordneten Fensters ändert. Der folgende Einzeiler instanziiert ein temporäres Dialogfeldobjekt, das *YourWinFormsDlgControl* hostet und es als modales Dialogfeld anzeigt:

```
CWinFormsDialog<YourWinFormsDlgControl>().DoModal();
```

Allerdings ist diese Konstruktion nicht zu empfehlen, da Sie mit *DoModal* nicht mehr an den Rückgabewert des Dialogfelds herankommen. Für realistischere Szenarios sollten Sie Ihre eigene Klasse von *CWinFormsDialog<YourWinFormsDlgControl>* ableiten. Damit können Sie *OnInitDialog* überschreiben, um Eigenschaften des verschachtelten Windows-Forms-Steuerelements zu setzen oder um Ereignisse des gehosteten Steuerelements zu behandeln. Die weiter vorn beschriebenen Makros aus *Event.h* können hier ebenfalls nützlich sein. Weitere Informationen finden Sie unter msdn2.microsoft.com/ahdd1h97.aspx.

Windows-Forms-Ansichten

Zu guter Letzt können Sie Windows-Forms-Steuerelemente als Ansichten hosten. *CWinFormsView* ist der Gehilfe, den Sie für dieses Szenario brauchen. Im Gegensatz zu *CWinFormsDialog* handelt es sich nicht um eine Vorlagenklasse. Dennoch gibt es auffallende Ähnlichkeiten. Beide überlassen das eigentliche Hosting des Steuerelements der *CWinFormsControl*-Vorlage und beide behandeln die Größenanpassung des gehosteten Steuerelements.

Um ein Windows-Forms-Steuerelement als MFC-Dialogfeld zu hosten, müssen Sie Ihre Ansichtsklasse von *CWinFormsView* ableiten. Der folgende Code zeigt die Deklaration einer einfachen Ansichtsklasse, die auf *CWinFormsView* basiert:

```
class CMyWinFormsBasedView : public CWinFormsView
{
    CMyWinFormsBasedView ();
    DECLARE_DYNCREATE(CMyWinFormsBasedView)
};
```

Wurde Ihre Ansichtsklasse durch einen Assistenten generiert, achten Sie darauf, dass Sie nicht nur in der Basisklassenliste der Klassendeklaration Ihrer Ansicht zu *CWinFormsView* wechseln, sondern auch in ihrer Implementierung. Dadurch müssen Sie gegebenenfalls Parameter von Makros wie *IMPLEMENT_DYNAMIC* und *BEGIN_MESSAGE_MAP* ändern. Um das gehostete Windows-Forms-Steuerelement zu instanziieren, erhält der Konstruktor von *CWinFormsView* (der Basisklasse Ihrer Ansicht) ein Handle auf das *System::Type*-Objekt des Windows-Forms-Steuerelements. Die Implementierung der eben gezeigten Ansichtsklasse kann ebenfalls sehr einfach sein:

```
IMPLEMENT_DYNCREATE(CMyWinFormsBasedView, CWinFormsView)

CMyWinFormsBasedView:: CMyWinFormsBasedView ()
    : CWinFormsView(WinFormsViewControl::typeid) {}
```

Listing 5: View überschreiben

```
#using <mfcmifc80.dll>
using Microsoft::VisualC::MFC::IView;

public ref class MyWinFormsViewControl :
    public System::Windows::Forms::UserControl,
    public IView
{
    ...
protected: // IView implementation
    virtual void OnInitialUpdate() = IView::OnInitialUpdate {
        // implementation
    }

    virtual void OnUpdate() = IView::OnUpdate {
        // implementation
    }

    virtual void OnActivateView(bool activate) = IView::OnActivateView {
        // implementation
    }
};
```

Auf den ersten Blick sieht dieser Code viel zu einfach für ein reales Szenario aus, doch in vielen Fällen brauchen Sie gar nicht mehr. Die Ansichtsklasse agiert als einfacher Proxy für die reale Implementierung, die sich im Steuerelement der Ansicht befindet.

Viele Implementierungen überschreiben virtuelle Funktionen der MFC-Basisklasse *CView*. Um die native Ansichtsklasse weiterhin als einfachen Proxy agieren zu lassen, könnten Sie diese Methoden in der nativen Klasse überschreiben und die Aufrufe an äquivalente Methoden im Windows Forms-Steuerelement der Ansicht weiterleiten. Für die drei wichtigsten Überschreibungen von *CView* ist derartiges Weiterleiten bereits in der Basisklasse *CWinFormsView* implementiert. Diese drei Methoden sind *OnInitialUpdate*, *OnUpdate* und *OnActivateView*. Das Weiterleiten basiert auf einer verwalteten Schnittstelle namens *Microsoft::VisualC::MFC::IView*, die in der Assembly *Mfcmifc80.dll* definiert ist. Um diese Methoden zu überschreiben, implementieren Sie diese Schnittstelle in der Windows-Forms-Steuerelementklasse der Ansicht, wie es in Listing 5 zu sehen ist.

Listing 6: Befehlshandler registrieren

```
public ref class MyWinFormsViewControl :
    public System::Windows::Forms::UserControl,
    public ICommandTarget
{
...
protected: // ICommandTarget implementation
    virtual void RegisterCmdHandlers(ICommandSource^ cmdSrc) =
        ICommandTarget::Initialize
    {
        cmdSrc->AddCommandHandler(ID_EDIT_PASTE,
            gcnew CommandHandler(this, &WinFormsView::OnEditPaste));
        cmdSrc->AddCommandUIHandler(ID_EDIT_PASTE,
            gcnew CommandUIHandler(this,
            &WinFormsView::OnUpdateEditPaste));

        // register further command handlers ...
    }

    void OnEditPaste(unsigned int)
    {
        // implement command handler
    }

    void OnUpdateEditPaste(unsigned int, ICommandUI^ cmdUI)
    {
        cmdUI->Enabled = ...; // your logic here
    }
};
```

Listing 7: HwndSource verwenden

```
BOOL CWPFDemoDialog::OnInitDialog()
{
    __super::OnInitDialog();

    CRect rectClient;
    this->GetClientRect(&rectClient);

    System::Windows::Interop::HwndSourceParameters hwsPars;
    hwsPars.ParentWindow = System::IntPtr(this->m_hWnd);
    hwsPars.WindowStyle = WS_CHILD | WS_VISIBLE;
    hwsPars.PositionX = 0;
    hwsPars.PositionY = 0;
    hwsPars.Width = rectClient.Width();
    hwsPars.Height = rectClient.Height();
    System::Windows::Interop::HwndSource^ hws;
    hws = gcnew System::Windows::Interop::HwndSource(hwsPars);

    using System::Windows::Controls::MonthCalendar;
    MonthCalendar^ mc = gcnew MonthCalendar;
    hws->RootVisual = mc;

    return TRUE;
}
```

Außer dem Überschreiben von virtuellen Funktionen können Ansichten auch Befehlshandler implementieren. Auch hier wäre es möglich, einen derartigen Befehlshandler in der nativen Ansichtsklasse zu implementieren und an das Windows-Forms-Steuerelement weiterzuleiten, doch da dies ein gebräuchliches Szenario ist, stellt MFC eine komfortablere Lösung bereit. Die Assembly *Mfcmifc80.dll* enthält einige weitere verwaltete Hilfsklassentypen für dieses Szenario. Um Befehlsnachrichten von der Befehlsweiterleitungsinfrastruktur der MFC zu empfangen, müssen Windows-Forms-Steuerelemente die Schnittstelle *Microsoft::VisualC::MFC::ICommandTarget* implementieren:

```
interface class ICommandTarget
{
    void Initialize(ICommandSource^ commandSource);
};
```

Beim Erstellen einer Ansicht, die *ICommandTarget* implementiert, wird *ICommandTarget::Initialize* auf dieser Ansicht aufgerufen und ein Handle zu einem neuen Befehlsquellobjekt als Argument übergeben. Dieses Argument ist vom Typ *ICommandSource^*. Das Objekt, auf das das Argument verweist, ist ein Container für Handler. Diese Handler sollten MFC-Entwicklern vertraut sein: Es sind Befehlshandler, mit denen sich steuern lässt, ob die Benutzeroberfläche eines Steuerelements auf Benutzerinteraktionen reagieren kann, ob das Steuerelement aktiviert ist, ob es mit einem Optionsfeld ausgestattet ist und welcher Text in der Benutzeroberfläche des Befehls angezeigt wird.

Um einen Handler zu registrieren, können Sie Methoden wie *AddCommandHandler* und *AddCommandUIHandler* auf der Befehlsquelle aufrufen. Diese Methoden erwarten zwei Argumente: eine vorzeichenlose Ganzzahl für die Befehlskennung und einen Delegaten, der für die Übergabe der Handlerfunktion des Steuerelements verwendet wird. Da Befehlshandler und Handler der Benutzeroberfläche unterschiedliche Signaturen besitzen, sind im Namespace *Microsoft::VisualC::MFC* zwei Delegattypen definiert:

```
delegate void CommandHandler(unsigned int);
delegate void CommandUIHandler(unsigned int,
    Microsoft::VisualC::MFC::ICommandUI^);
```

Listing 6 zeigt, wie ein Befehlshandler für den Befehl *ID_EDIT_PASTE* registriert wird.

Tabelle 2: CreateWindowEx und HwndSourceParameters

CreateWindowEx Arguments	HwndSourceParameters Properties
LPCTSTR lpClassName	*N/A*
N/A	*WindowClassStyle*
LPCTSTR lpWindowName	*WindowName*
DWORD dwStyle	*WindowStyle*
DWORD dwExStyle	*ExtendedWindowStyle*
int x	*PositionX*
int y	*PositionY*
int nWidth	*Width*
int nHeight	*Height*
HWND hWndParent	*ParentWindow*
HMENU hMenu	*N/A*
HINSTANCE hInstance	*N/A*

Soll eine Ansicht einen Befehl behandeln, kommt der Delegat ins Spiel, um eine Memberfunktion der Ansicht aufzurufen. Die ID des Befehls wird als Argument übergeben. Wenn jeder Befehl seine eigene private Handlerfunktion besitzt, ist das Argument nicht wichtig, aber eine einzelne Methode kann auch als Handler für mehrere Befehle fungieren. Mit *AddCommand[UI]RangeHandler* können Sie einen Delegaten für mehrere Befehle registrieren.

Der Delegat für Handler von Benutzeroberflächenbefehlen besitzt ein zusätzliches Argument vom Typ *ICommandUI^*. Das mithilfe dieses Parameters übergebene Handle ist ein Wrapper für die MFC-Klasse *CCmdUI*, mit der Sie die Erscheinung und Nutzbarkeit von Benutzeroberflächenelementen wie Menübefehlen und Symbolleistenschaltflächen steuern können.

Außer Befehlshandler in *ICommandSource::Initialize* hinzuzufügen, kann es auch nützlich sein, das *ICommandTarget*-Handle in einer Membervariablen des Windows-Forms-Steuerelements zu speichern. Dadurch haben Sie die Möglichkeit, später weitere Befehlshandler hinzuzufügen oder zu entfernen, und Sie können sogar Befehlsereignisse entweder synchron oder asynchron über *ICommandTarget::SendMessage* oder *ICommandTarget::PostMessage* auslösen.

Brücken zu Avalon

Da Windows Presentation Foundation noch nicht in der endgültigen Form freigegeben ist, besitzt die aktuelle Version der MFC keine Hilfsklassen und Vorlagen für die Integration ihrer Visuals in *CWnds* und *CDialogs*. Allerdings heißt das nicht, dass die Integration von Visuals in MFC-Anwendungen nicht möglich ist. In der Tat sind selbst ohne diese kleinen Hilfsklassen nur wenige Codezeilen notwendig, um ein Visual in einem *CWnd* oder einem *CDialog* zu hosten.

Das Schlüsselmerkmal für diese Integration stammt von Windows Presentation Foundation selbst. Die Assembly *Windows.PresentationCore.dll* stellt einen Namespace *System::Windows::Interop* mit einer sehr leistungsfähigen Klasse namens *HwndSource* bereit. Diese Klasse schlägt die Brücke zwischen einem *HWND* und einem Visual. Für das hostende *USER32*-Fensterobjekt sieht es wie ein untergeordnetes Fenster mit einem normalen *HWND* aus. Mithilfe der Eigenschaft *RootVisual* können Sie in die neue Welt von Windows Presentation Foundation wechseln. (Beachten Sie, dass die hier angegebenen Informationen auf einer Prerelease-Version von Windows Presentation Foundation beruhen und sich mit der endgültigen Release noch ändern können.)

Der *HwndSource*-Konstruktor erwartet einen *HwndSourceParameters*-Werttyp als Argument. Wie Tabelle 2 zeigt, enthält dieser Werttyp ähnliche Informationen wie die, die Sie als Argumente in einem Aufruf der Win32-Funktion *CreateWindowEx* übergeben würden. Listing 7 demonstriert, wie *HwndSource* in einer *OnInitDialog*-Implementierung verwendet wird.

Anzeige

Einfach Integrieren

Mit C++-Interop lässt sich verwalteter Code nahtlos in vorhandene C++-Quellen integrieren. Die MFC-Unterstützung für Windows Forms kann man als überzeugenden Beweis ansehen, dass diese Integration ein vorgesehenes und unterstütztes Feature von Visual C++ ist. Um diese Integration zu vereinfachen, existieren mehrere Hilfsklassen und Vorlagen. Diese Integrationsschicht ist weder komplex noch schwierig zu verwenden.

Was noch fehlt, ist eine nahtlose Integration dieser Features in die Assistenten von Visual Studio. In den meisten Szenarios ist dies kein großes Problem, da ein großer Teil der Implementierung dem gehosteten Windows-Forms-Steuerelement zufällt, das in Visual Studio hervorragende Designerunterstützung genießt. Bei Auslieferung von Windows-Presentation Foundation ist es sehr wahrscheinlich, dass äquivalente Unterstützung für das Integrieren von Visuals in MFC-Anwendungen bereitgestellt wird. Doch auch ohne derartige Unterstützung ist es einfach, Visuals in MFC-Anwendungen zu hosten.

Marcus Heege arbeitet als Kursautor und Trainer für DevelopMentor und bietet Beraterleistungen für verschiedene IT-Unternehmen an. Er postet regelmäßig seine Gedanken zu C++/CLI und .NET in seinem Blog unter www.heege.net.