



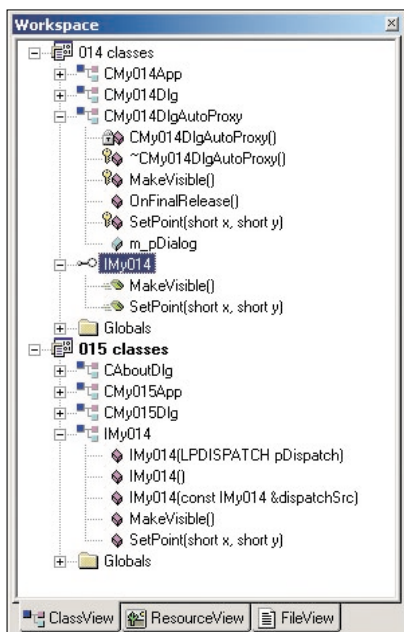
Steuerung von Programmen

Automation ganz einfach

Ein gutes Windows-Programm besteht nicht nur aus einer Menuleiste, einer Werkzeugleiste und einem Arbeitsbereich: Ein solches Programm sollte man auch fernsteuern können.

THOMAS WÖLFER

Die Automatisierung von Programmen verbirgt sich unter Windows hinter dem Begriff "Automation" – und wie man ein Programm Automations-fähig macht, erfahren Sie in diesem Beitrag. Automation hat verschiedene Vorteile – unter anderem den, dass das Programm zum Beispiel direkt von einem Office Makro aus verwendet werden kann. Angenehm



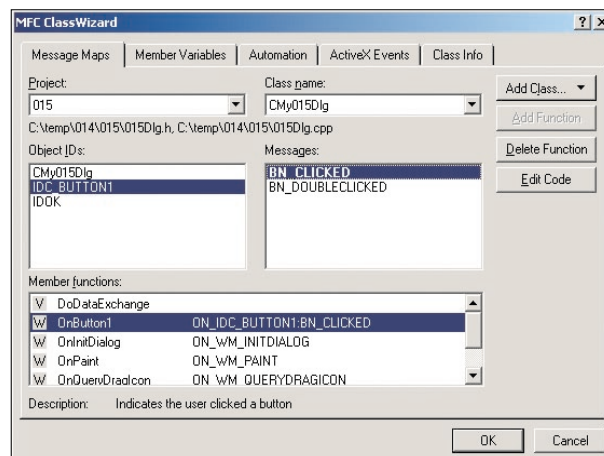
DIE KLASSENANSICHT im Arbeitsbereich wird bei Automation sehr interessant, denn hier sehen Sie nicht nur die C++-Klasse und Methoden, sondern auch die bislang definierten Interfaces.

merweise ist es mit MFC relativ einfach ein automatisierbares Programm zu schreiben, und das liegt daran, dass dies

omatisierbare Objekte in anderen Anwendungen erzeugen können, die die eher komplexen Details verbergen, die für die Nutzung von automatisierbaren Objekten notwendig sind. Obendrein sorgen sie dafür, dass Ihr Code weiterhin typensicher bleibt.

Die Grundlagen

Damit ein Programm automatisierbar wird, muss es in irgendeiner Form Funktionalität exportieren. Auch im Fall von Automation passiert dies im Common Object Model (COM wird im nächsten Beitrag behandelt).



MIT DEM KLASSENASSISTENTEN binden Sie eine Methode an ein Ereignis. Eine solche Methode nenne man Event-Handler. Der Klassenassistent erzeugt den kompletten dafür benötigten Code.

bei MFC von Haus aus vorgesehen ist. In der MFC-Bibliothek findet sich dazu eine eigene Klasse mit Namen `CCmdTarget`, die alle für Automation benötigten Funktionen mitliefert. Dabei muss diese Klasse normalerweise noch nicht einmal verwendet werden, denn die zu automatisierenden Klassen sind ohnehin meist von dieser Klasse abgeleitet. Das gilt zum Beispiel für die Anwendungsklasse und ähnliche in der Hierarchie weiter oben angesiedelte Klassen.

Darüber hinaus enthält der Klassen-Assistent Funktionalität, mit der Sie leicht C++-Wrapper-Klassen für auto-

Im Fall von Automation sieht die Sache so aus, dass hauptsächlich ein ganz spezielles Interface exportiert wird, und das ist das `IDispatch`-Interface. Dieses Interface ist das, welches von den anderen Programmen verwendet werden kann. (Es gibt noch anderer Spielarten, aber hier soll nur einmal die elementare Variante erläutert werden.) Das `IDispatch`-Interface zeichnet sich durch eine ganz spezielle Methode namens `InvokeHelper()` aus, und die ist in der Lage andere Methoden aufzurufen. Der Vorteil daran ist der, dass nach außen hin eigentlich nur die `InvokeHelper`-Methode bekannt sein muss – alle anderen



Methoden können dann über diese eine erreicht werden. Der Nachteil: Das ist reichlich langsam.

Für ein per Skript automatisierbares Programm reicht die Geschwindigkeit aber häufig aus. Im Wesentlichen enthält ein automatisierbares Programm also mindestens ein Interface, das nach außen hin sichtbar ist, und das nach innen mit den tatsächlich implementierten C++-Methoden kommuniziert.

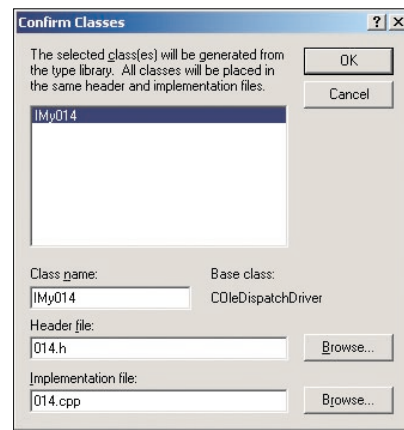
Im Beispielprogramm ist das ein bisschen komplizierter, denn das Beispielprogramm ist rein Dialogbox-basiert – und da sind noch ein paar weitere Vorkehrungen zu treffen. Weil die MFC-Dialogbox-Klasse nicht von `CCmdTarget` abgeleitet ist, braucht man eine weitere Klasse, die für die Automatisierung nach außen hin als Dialogbox auftritt. Mit anderen Worten: Man braucht eine Proxy-Klasse, mit der Programme von außen kommunizieren können, und die die Arbeit dann an die eigentliche Dialogbox-Klasse weiterreicht.

Für diesen Proxy braucht man außerdem die passenden IDL-Scripte, damit der MIDL-Compiler in der Lage ist, die Type Library zusammenzubauen. Das bedeutet auch, dass Funktionen, die nach außen freigegeben werden sollen, eigentlich zweimal zu implementieren sind: Einmal im Proxy und einmal in der echten Dialogbox-Klasse. (Und natürlich ein drittes Mal in der IDL-Datei, das geht allerdings vollautomatisch.) Die Implementierung in der Proxy-Klasse muss dabei aber nicht viel tun, sondern kann direkt die "echte" Implementierung der Methode in der C++-Klasse aufrufen.

Ein Programm, das automatisierbar ist, nennt man "Automation-Server" und wie man sich bei diesem Namen leicht denken kann, gibt es dann auch einen entsprechenden "Automation Client". Genau wie bei COM kann auch der Automation-Client in einer beliebigen Sprache programmiert werden – beim COM-Projekt haben Sie zum Beispiel einen Client in VBScript per WSH und in HTML eingebettet gesehen. In diesem Beispielprojekt wird der Automation-Client aber in C++ programmiert werden, und darum gibt es hierzu auch noch ein paar Hintergründe.

Handelt es sich beim Client um ein VBS-Programm, ist nicht viel zu beachten: VBS ist alles andere als typensicher – also sind die Typen der Parameter und

Rückgabewerte des Automation-Servers in VBS auch nicht weiter von Bedeutung: Das COM-System und der Server müssen sich aus Sicht von VBS



BEIM EINLADEN DER TYPE LIBRARY müssen Sie angeben, wie die Dateien genannt werden sollen, die den Code für die Wrapper-Klasse aufnehmen werden.

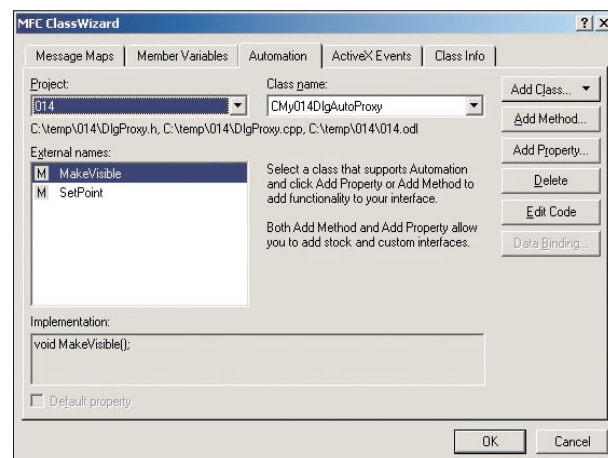
darum kümmern, dass eventuell benötigte Typenumwandlungen richtig durchgeführt werden. Das ist bei einem C++-Programm natürlich anders, denn einer der Gründe warum man in C++

kompatibel zu den normalen C++-Datentypen: Zwar überlappt sich das Typensystem der beiden Sprachen in vielen Teilen, doch unterscheidet es sich zum Teil auch recht stark.

Nun ist es aber so, dass man im Quellcode dauernd mit irgendwelchen Interface-Pointern hantieren muss, wenn man Methoden der dahinter liegenden Interfaces aufrufen will: Die sind aber natürlich auch aus dem IDL-Typensystem. In der Praxis würde das bedeuten, dass Sie im Quellcode dauernd irgendwelche Typumwandlungen – also casts – durchführen müssen. Das ist nicht nur sehr unschön und unpraktisch, sondern auch gefährlich. Aus diesem Grund hat man sich bei Microsoft für VC++ etwas anderes einfallen lassen: die eingangs erwähnten Wrapper-Klassen.

Immer wenn Sie eine Type Library haben, von der Sie die zugeordneten Objekte verwenden wollen, können Sie mit dem Klassen-Assistenten eine solche Wrapper-Klasse für alle in der Type Library enthaltenen Klassen erzeugen lassen. Das Resultat ist dann eine saubere, typensichere Kapselung der Funktionalität aus der Type Library bzw. der dahinter liegenden Objekte. Statt des Interface Pointers selbst verwenden Sie dann nur noch diese Wrapper-Klasse und der darin befindliche Code kümmert sich darum, dass alle benötigten Konvertierungen korrekt durchgeführt werden.

Um Automation mit C++ vollständig ausprobieren zu können braucht man nicht nur ein, sondern zwei Programme. Das eine ist der Automation-Server, das andere der Automation-Client. Für dieses



IM KLASSENASSISTENTEN erzeugen Sie neue Methoden für die Proxy-Klasse. Die benötigten IDL-Statements werden dann gleich automatisch miterzeugt.

programmiert, ist die Typensicherheit. Nun ist es aber so, dass die Typen des Automation-Servers aus der Type Library gelesen werden. Das kann bereits vor dem Compilieren oder erst zur Laufzeit des Programms geschehen (im Beispiel ist ersteres der Fall) – doch in beiden Fällen ergibt sich ein kleines Problem: Die in der Type Library verwendeten Datentypen sind IDL-Datentypen – und die sind alles andere als

Beispiel sind beide Programme als Dialogbox angelegt, aber wie erwähnt, ist das bei MFC keine Pflicht. Außerdem ist der im Beispiel implementierte Automation-Server auch nicht sonderlich beeindruckend: Das Programm malt ein Icon in einen Bereich auf der Dialogbox und exportiert eine Methode, mit der die Position dieses Icons festgelegt werden kann. Der Automation-Client, also das andere Programm, legt dann die Positi-

on des Icons fest. Damit man da ein bisschen mehr sehen kann, wird das Icon dabei mehrfach an per Zufallszahl produzierte Positionen geschoben. Das Beispielprojekt ist dabei im Arbeitsbereich verschachtelt: Der Automation-Client ist als vom Automation-Server abhängiges Projekt definiert.

■ Zur Praxis: der Server

Zunächst zum Automation-Server. Den erzeugen Sie ganz normal mit dem Anwendungs-Assistenten für eine MFC-Anwendung. Dabei verwenden Sie den Typ "Dialogbox": Das Programm wird also weder SDI noch MDI unterstützen, sondern sich einfach nur aus einer Dialogbox zusammensetzen. Das einzige, auf was Sie dabei achten müssen, ist, dass Sie bei der Auswahl der Features des Programms die Option *Automation* ausgewählt haben, sonst fehlt der komplette Automation-Support im erzeugten Quellcode und das ist in diesem Fall nicht gerade wenig.

Zunächst ist es ganz hilfreich, wenn Sie die später zu exportierende Funktionalität lokal testen können. Das bedeutet, dass das Programm nun zunächst einmal mit der gewünschten Funktionalität plus ein wenig Testfunktionen ausgestattet werden muss. Wie erwähnt soll im Programm ein Icon innerhalb einer bestimmten Fläche verschoben werden können. Dazu brauchen Sie zunächst einmal diese Fläche auf der Dialogbox. Dazu ziehen Sie aus der Control-Leiste ein Image-Control auf die Dialogbox, ziehen es in die gewünschte Größe und Position, vergeben die sonstigen Attribute (die in diesem Fall alle keine Rolle spielen) und vergeben einen Namen für das Control. Im Beispiel wurde der symbolische Name `IDC_AREA` gewählt, weil es sich um den Bereich für die Anzeige des Icons handelt.

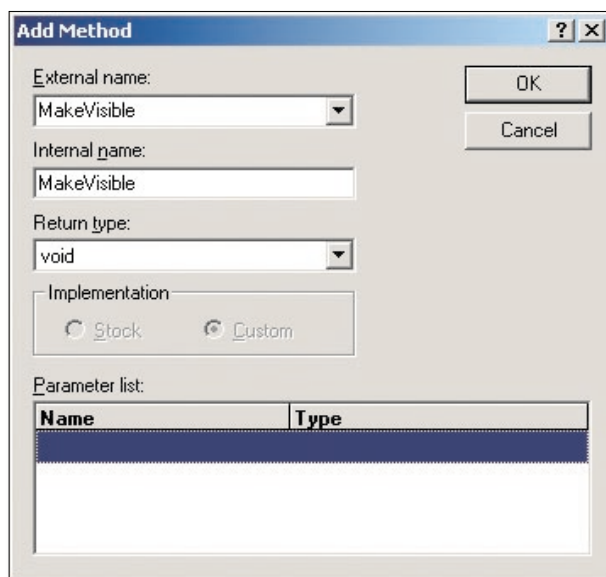
Dann brauchen Sie noch einen Punkt, an dem das Icon angezeigt werden soll. Dazu fügen Sie in der Klassendefinition

der Dialogboxklasse ein neues Member vom Typ "CPoint" ein. Im Beispielprogramm wurde dieses Member `m_pt` genannt. Nachdem es sich um ein Daten-Member handelt, sollten Sie den Punkt im "private"-Bereich der Klasse unterbringen.

Ein Icon haben Sie praktischerweise schon, denn der vom Anwendungsassistenten erzeugte Quellcode benutzt ein solches und kümmert sich auch bereits darum, dass es richtig geladen wird. Verwenden können Sie dieses Icon über den bereits vorliegenden Member `m_hIcon` der Dialogboxklasse. Sie müssen sich allerdings nun noch darum kümmern, dass das Icon auch gezeichnet wird. Dazu sind zwei Dinge zu tun:

- 1.) Der Punkt, also die Position des Icons, muss initialisiert werden und
- 2.) das Icon muss an dieser Position ausgegeben werden.

Die Initialisierung des Punkts nehmen Sie im Konstruktor der Dialogboxklasse vor. Mit welchen Werten Sie



DIE EINZELNEN ZU EXPORTIERENDEN METHODEN müssen Sie mit einem internen und einem externen Namen versehen. Am besten verwenden Sie für beides den gleichen Namen.

den Punkt initialisieren ist dabei egal – diese Werte sollen ja später ohnehin von einem anderen Programm geliefert werden. Sie brauchen also eine Zeile, in der in etwa folgender Ausdruck steht:

```
m_pt = CPoint( 15, 15);
```

Nun müssen Sie sich noch darum kümmern, dass das Icon auch gemalt wird, und dazu brauchen Sie die *OnPaint()*-Methode des Dialogs. Diese ist bereits vorhanden und vom Anwendungsassistenten mit Quellcode aufgefüllt worden, der sich um die Anzeige des Icons im minimierten Zustand der Dialogbox kümmert. Diesen Code erweitern Sie, sodass danach zusätzlich zu den vorhandenen die folgenden Statements in *OnPaint()* zu finden sind (siehe Listing 1).

LISTING 1

```
else
{
    CDialog::OnPaint();
    CPaintDC dc( GetDlgItem( IDC_AREA));
    dc.DrawIcon( m_pt, m_hIcon);
}
```

Dazu ist nicht mehr zu sagen: Sie sollten Ihre Anwendung nun übersetzen und ausprobieren: Im Bereich des Controls mit dem Namen `IDC_AREA` sollte dann das MFC-Icon zu sehen sein.

Um nun auszuprobieren, ob das Icon überhaupt verschiebbar gemacht werden kann, platzieren Sie einen neuen Button auf der Dialogbox und implementieren dafür einen Click-Handler. Im Beispiel hat der Button die ID `IDC_BUTTON1` und den Click-Handler *OnButton1()*. (Das ist genau das, was herauskommt, wenn Sie einfach die Vorgabewerte des Dialogbox-Editors und des Klassen-Assistenten verwenden. Im Eventhandler verschieben Sie dann die Werte für den Punkt und invalidieren das Fenster, was seinerseits dazu führt, dass das Icon an einer neuen Stelle zu sehen ist:

```
void CMy014Dlg::OnButton1()
{
    m_pt = CPoint( 50, 50);
    Invalidate();
}
```

Einmal neu übersetzt und getestet – und Sie wissen nun, wie das Icon prinzipiell zu verschieben ist. Nun kommt der Automations-Teil hinzu, und der stellt sich interessanterweise auch nicht als wesentlich komplizierter heraus. Zumindest wenn man das "Geheimnis" der unsichtbaren Automations-Server kennt, und das werden Sie gleich kennen lernen.

■ Automation im Server

Wenn Sie ihr aktuelles Projekt in der Klassenansicht betrachten, werden Sie



feststellen, dass dort deutlich mehr Klassen angezeigt werden, als man vermuten würde: Im Besonderen findet sich dort eine Klasse in deren Namen das Wort "Proxy" auftaucht sowie ein Interface das offensichtlich zu diesem Proxy gehört. Beim Proxy handelt es sich natürlich um den zuvor erwähnten Proxy für die Dialogbox.

Wenn Sie die Klassendefinition der Dialogbox-Klasse untersuchen, werden Sie ein C++-Statement finden, das an keiner anderen Stelle in diesem Sonderheft erwähnt wurde: das *friend*-Statement. Es befindet sich recht weit oben in der Klassendefinition der Dialogbox-Klasse und deklariert die "Proxy"-Klasse als "Friend" (also Freund) dieser Klasse. Das bedeutet im Wesentlichen eines: Die Proxy-Klasse darf auf die privaten Daten der Dialogbox-Klasse zugreifen, so auch auf den von Ihnen mühevoll im "private" Bereich eingefügten Punkt vom Typ CPoint. Das ist zwar nicht sehr sauber – aber auf der anderen Seite durchaus logisch nachvollziehbar. Schließlich "ist" der Proxy nach außen hin die Dialogbox, sodass er aus dieser Sichtweise betrachtet auch durchaus ein Recht auf deren private Daten hat.

Was man nun braucht, ist eine neue Methode im zum Proxy gehörenden Interface. Die können Sie genau wie beim COM-Beispiel mit einem Klick der rechten Maustaste auf das Interface in der Klassenansicht generieren. Die daraufhin erscheinende Dialogbox unterscheidet sich aber von der im COM-Projekt: Das liegt daran, dass Sie jetzt eine Methode für Automation anlegen.

Dazu müssen Sie einen "externen" und einen "internen" Namen angeben. Der "externe" Name ist der Name, unter dem die Methode exportiert und von anderen

Programmen verwendet werden soll. Der "interne" Name ist der Name, den die Methode in der Implementierung des Interfaces, also in der Proxy-Klasse erhalten soll. Um das Chaos nicht unnötig zu vergrößern, sollen Sie hier also besser keine unterschiedlichen Namen verwenden, im Beispiel ist das *SetPoint()*.

Ferner müssen Sie einen Returnwert angeben und die Parameter für die Methode spezifizieren. Als Returnwert geben Sie "void" an – die Methode soll keinen Returnwert haben. Als Parameter geben Sie zwei Stück an. Die nennen Sie "x" und "y" (für die neue Position) und

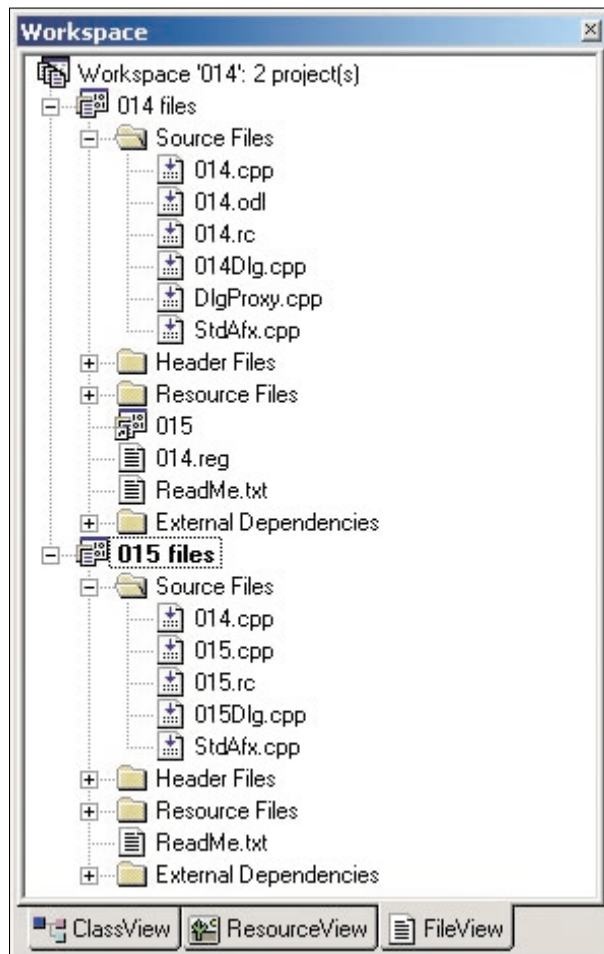
wählen als Typ einen Integer-Typ wie zum Beispiel "short".

Wenn Sie die Dialogbox nun schließen, können Sie die Methode in der Klassenansicht sowohl beim Interface als auch bei der Proxy-Klasse wieder finden. In der Proxy-Klasse müssen Sie aber noch die Implementierung zur Verfügung stellen. Die ist aber verhältnismäßig unspektakulär, schließlich müssen Sie nur genau das tun, was Sie auch schon in der Testfunktion zuvor implementiert haben: Einen Punkt verschieben und dann den Dialog invalidieren (siehe Listing 2).

Jetzt wird noch das Geheimnis verraten: Wenn Sie einen Automation-Server

LISTING 2

```
void CMY014DlgAutoProxy::SetPoint(short x, short y)
{
    m_pDialog->m_pt = CPoint(x, y);
    m_pDialog->Invalidate();
}
```



IN EINEM ARBEITSBEREICH können auch mehrere Projekte enthalten sein. Bei diesem Beispiel ist das der Fall. Das macht die Arbeit an zusammenhängenden Projekten deutlich einfacher.

per Automation von einem anderen Programm aus starten, wird dieser – zumindest wenn er mit MFC implementiert wurde – von Haus aus nicht sichtbar gemacht. Das kann durchaus eine längere Problemsuche provozieren, denn schließlich scheint alles zu funktionieren: Nur sehen tut man eben nichts. Aus diesem Grund fügen Sie nun eine weitere Implementierung zum Interface und dem Proxy hinzu. Die Methode hat keine Parameter und trägt den Namen *MakeVisible()*. Die Implementierung von *MakeVisible* ist nicht sonderlich aufwändig (siehe Listing 3 auf der nächsten Seite).

Damit ist Ihr Automations-Server fertig gestellt. Fehlt nun noch ein Programm, mit dem Sie das auch testen können, und das folgt nun.

Der Automation-Client

Auch den Automation-Client legen Sie als Dialogbox-Projekt an. Damit die beiden Projekte gleichzeitig besser bearbeitet werden können, sollten Sie das neue Projekt aber zum Arbeitsbereich des alten Projekts hinzufügen – dann haben Sie immer beide Projekte im Zugriff. Welches der beiden Zeilen erzeugt wird, können Sie über die entsprechende Werkzeugleiste oder das "Erstellen"-Menü einstellen.

Auf die Option "Automation" können Sie bei den Einstellungen zum zwei-

LISTING 3

```
void CMY014DlgAutoProxy::MakeVisible()
{
    m_pDialog->ShowWindow( SW_SHOW);
}
```

ten Projekt übrigens verzichten, denn das zweite Projekt soll ja nicht automatisierbar sein.

Denkbar wäre das natürlich trotzdem: Sie können auch eine ganze Kette von automatisierbaren Programmen schreiben, bei denen das erste in der Kette das zweite betreibt, das zweite das dritte und so weiter. Fürs Beispielprogramm wäre das aber ein wenig übertrieben, und daher gibt es eben nur zwei Programme, von denen nur eines automatisierbar ist.

Wenn der Anwendungsassistent den initialen Quellcode für das Projekt erzeugt hat, müssen Sie zunächst einmal sicherstellen, dass der Library Support für OLE geladen und initialisiert wird. Der wird nämlich sehr wohl für die Nutzung eines Automation-Servers aus dem Client herausgebraucht. Der notwendige Code dafür ist aber simpel.

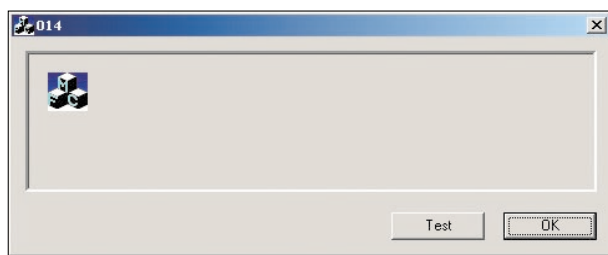
Fügen Sie dazu in der *InitInstance()*-Methode Ihrer Anwendungsklasse die folgenden Zeilen ein (siehe Listing 4).

Damit sind Sie schon mehr oder minder am Ziel: Ihr Programm ist

wenden, ist also ohne Bedeutung.

Die Verwendung ist dabei ganz einfach: Wie bereits erwähnt, kann der Klassenassistent eine oder mehrere Wrapper-Klassen

für die in der Type Library definierten Objekte erzeugen. Öffnen Sie dazu den Dialog des Klassenassistenten. Rechts oben auf dieser Dialogbox finden Sie einen Button



NICHT SCHÖN, ABER FUNKTIONAL: Das MFC Icon wird innerhalb des festgelegten Bereiches bewegt – und zwar von einem anderen Programm.

mit dem Titel *Klasse hinzufügen* und wenn Sie diesen Button betätigen dann öffnet sich ein Menü, in dem unter anderem der Befehl *Aus einer Typbibliothek* enthalten ist. Den wählen Sie nun aus.

Der Klassenassistent öffnet daraufhin einen ganz normalen File-Open-Dia-

se wählen Sie nun aus. Der Klassenassistent erfragt nun noch ein paar Dinge von Ihnen. Im Wesentlichen hätte er gerne gewusst, wie der Name der zu erzeugenden Wrapper-Klasse lauten soll, welchen Namen die zugehörige zu erzeugende Header-Datei haben soll und wie der Name der zu erzeugenden Implementierungs-Datei lauten soll. Dabei können Sie aber einfach alle Vorgaben akzeptieren. Das hat einen Vorteil: Wenn sich die Type Library in der Zukunft einmal ändern sollte – zum Beispiel, weil Sie zusätzliche Funktionen exportieren möchten – dann wissen Sie beim nächsten Einlesen der Type Library, dass der vorgeschlagene Name der gleiche wie beim letzten Mal ist, sodass die erzeugten Klassen nicht plötzlich in zwei unterschiedlichen Dateien in Ihrem Projekt

auftauchen.

Dazu aber noch eine Warnung: Der Klassenassistent erzeugt die angegebene Datei nicht immer neu, sondern hängt neu erzeugten Quellcode an die vorhandene Datei hinten an. Wenn Sie die Type Library also später erneut laden, enthält sie alle bisher bereits existierenden Methoden und Objekte doppelt. Das wird der Compiler natürlich bemängeln. Sie müssen die "alte" Klasse dann von Hand im Quellcode Editor löschen.

Ist der Quellcode generiert können Sie die erzeugte Klasse direkt verwenden. Für das Beispiel fügen Sie dazu einen neuen Button auf Ihrem Dialog ein und stattdessen mit einem Eventhandler für das Click Event aus. In diesem Handler werden Sie nun den Automation-Server verwenden. Damit das aber geht, müssen Sie natürlich das vom Klassenassistenten soeben erzeugte Header File noch inkludieren. Im Beispiel trägt die vom Klassenassistenten erzeugte Wrapper-Klasse den Namen *IMy014* – bei Ihnen wird das vermutlich ein anderer Name sein, denn der Name wird durch den Namen des exportierten Objekts bestimmt.

Das "I" im Namen wird aber auch bei Ihnen vorhanden sein: Damit will der Klassenassistent klarstellen, dass es sich hier um eine "Interface"-Klasse handelt. Davon erzeugen Sie in Ihrem Click-Handler nun eine neue Instanz:

LISTING 4

```
if (!AfxOleInit())
{
    AfxMessageBox("OLE konnte nicht initialisiert werden");
    return FALSE;
}
```

nun in der Lage einen Automation-Server zu verwenden. Doch welchen soll man nehmen? – Und wie tritt man mit dem Server in Kontakt? – Bei der Erstellung des ersten Projekts wurde nicht nur das Programm selbst, sondern auch eine Type Library für vom Programm exportierte Methoden und Objekte angelegt.

Diese Library finden Sie im gleichen Verzeichnis, wie das fertige Programm – bei einem Debug Build also im Verzeichnis "Debug" und bei einem "Release Build" im Verzeichnis "Release". Dabei unterscheiden sich diese beiden Type Libraries aber nicht – welche der beiden Sie daher ver-

log, mit dem Sie den Pfad zur verwendeten Type Library angeben können. Dabei können Sie übrigens auch feststellen, dass diese Libraries in verschiedenen Formen vorliegen können: So wie hier im Beispiel in Form einer separaten .TLB- Datei ist das genauso möglich, wie in einer DLL. Sie können also die Typinformation auch in der gleichen DLL ablegen, in der auch die Objekte abgelegt sind, die exportiert werden sollen: Das macht den Transport solcher Objekte natürlich einfacher, weil dann nur noch eine Datei zu behandeln ist. Wie auch immer – in diesem Fall haben Sie eine Type Library in Form einer separaten Datei vorliegen, und die-



```
IMY014* p = new IMY014();
```

Danach müssen Sie diese Instanz mit einer Instanz des zugehörigen Automation-Servers verbinden. Das tun Sie, indem Sie die "Prog-ID" des Servers angeben. Diese Prog-ID können Sie zum Beispiel dem IDL File, der .REG-Datei oder auch dem C++-Quellcode des Automation-Servers entnehmen. Im Beispiel lautet dieser Name "My014.Application". Der benötigte Aufruf lautet dann:

```
p->CreateDispatch  
("My014.Application");
```

Damit wird das Dispatch-Interface für den Automation-Server erzeugt. Mit anderen Worten: Dieser Aufruf startet den Automation-Server und verbindet Ihren Interfacepointer mit dem DispatchInterface im Automation-Server. Sie können nun die Methoden des Automation-Servers aufrufen. Wenn an dieser Stelle etwas schief läuft, liefert *CreateDispatch()* false zurück. Das hat meist immer den gleichen Grund: Irgendetwas stimmt mit der Registrierung des Automation-Servers nicht. Wechseln Sie in diesem Fall in das Projektverzeichnis des Automation-Servers, und verwenden Sie die dort vorliegende .REG-Datei, um den Server in der Registry Ihres Rechners anzumelden (siehe Listing 5).

Ansonsten steht der Arbeit mit dem anderen Programm nun nichts mehr im Wege. Damit aber auch etwas davon sichtbar wird, muss das Programmfenster selbst zunächst sichtbar gemacht werden:

```
p->MakeVisible();
```

Danach können Sie die Position des anzuzeigenden Icons im anderen Programm einfach nach Lust und Laune setzen. Zum Beispiel, indem Sie Zufallszahlen verwenden:

```
long t;  
time(&t);  
srand( t);  
for( int i = 0; i<1000; i++)  
{  
    int x = 160;  
    int y = 160;  
  
    // wert zwischen 0 und 150  
    while( x > 150)  
    {  
        x = rand();  
    }  
  
    // wert zwischen 0 und 70  
    while( y > 70)  
    {  
        y = rand();  
    }  
  
    // Positions veraendern  
    p->SetPoint( (short)x,  
                (short)y);  
}
```



EIN DRUCK AUF "DOIT!", und der Automation-Client beginnt mit seiner Arbeit. Dazu lädt er zunächst den Automation-Server.

```
}  
    (short)y);  
}  
delete p;
```

Schließlich müssen Sie den Zeiger auf das Dispatch-Interface noch löschen:

Das löscht aber nicht nur den Zeiger, sondern führt auch dazu, dass der Automation-Server wieder geschlossen wird, denn schließlich gibt es dann ja keine Referenz mehr auf den Server.

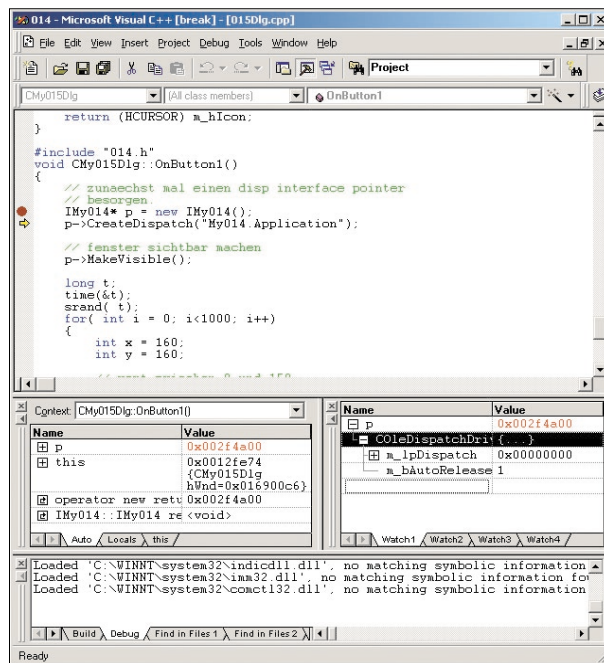
Wenn Sie beabsichtigen in Ihrem Client länger mit dem Server zu kommunizieren, können Sie natürlich auch einfach eine Member-Variable in einer Ihrer Klassen einbetten, um den Zeiger auf das IDispatch-Interface des Auto-

mation-Servers zu speichern: Solange diese Variable Scope hat, wird auch der Automation-Server zur Verfügung stehen. Zumindest in der Theorie, denn in der Praxis kann der Automation-Server ja von Ihnen einfach geschlossen werden,

obwohl der Client noch einen Zeiger auf den Server hält: Diesem Umstand müssen Sie in aufwändigeren Programmen natürlich Sorge tragen.

In diesem Beitrag haben Sie erfahren, wie Sie ein Programm aus einem anderen Programm heraus mit Automation fernsteuern: Mit MFC ist das tatsächlich ein Kinderspiel – allerdings auch ein sehr langsames, denn die Performance von Automation ist zwar für viele Anwendungsfälle ausreichend, aber sicher nicht für alle.

✓ UR



MIT DEM DEBUGGER ist es leicht herauszufinden, ob das Erzeugen des IDispatch-Interfaces gelungen ist.

LISTING 5

```
if( ! p->CreateDispatch("My014.Application"))  
{  
    AfxMessageBox("Vermutlich ist der Automation Server nicht registriert!");  
}
```