

Dynamic Link Libraries

Eine Beispiel-DLL mit MFC

Es gibt eine ganze Menge Möglichkeiten, um Code wieder zu verwenden, und die effektivste ist dabei eine DLL.

THOMAS WÖLFER

DLLs enthalten kompilierten Code, der von anderen Anwendungen benutzt werden kann. Der Quellcode für eine DLL kann dabei auch Klassen enthalten, und auch diese Klassen können in anderen Programmen benutzt werden. Dazu müssen im Prinzip nur die zugehörigen Header-Dateien verfügbar sein. Wie Sie eine wieder verwendbare DLL programmieren, die die MFC verwendet und eine Klasse exportiert, erfahren Sie in diesem Beitrag.

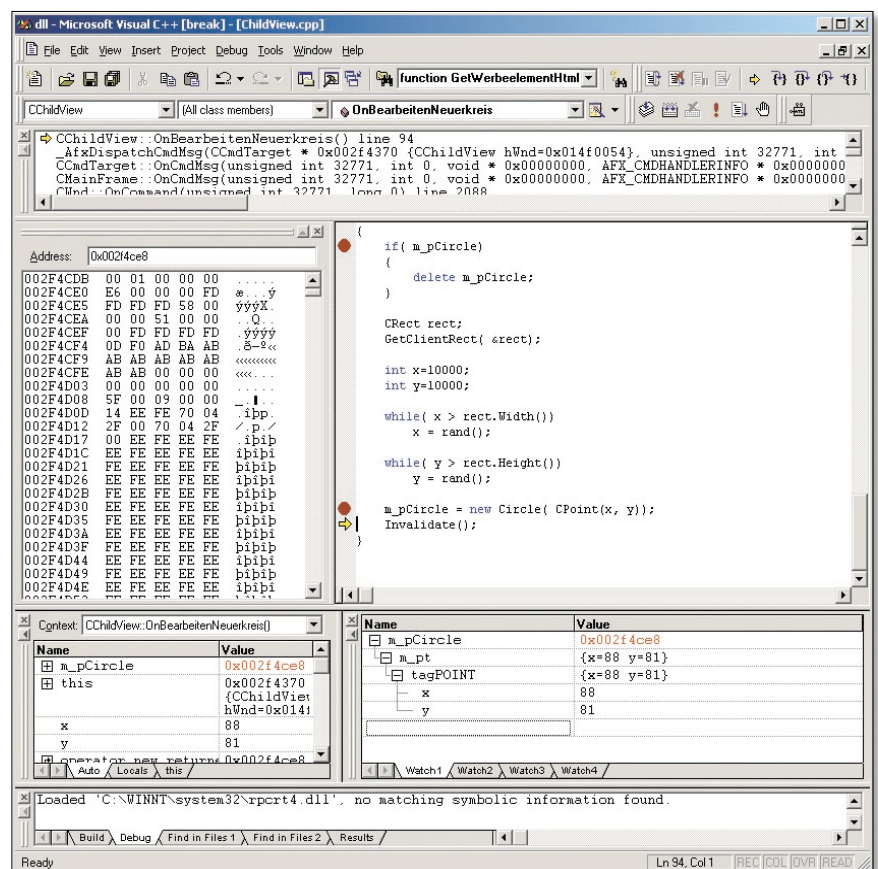
Die Wiederverwendung von Code ist wichtig – es ist nicht nur sinnlos Quellcode in verschiedenen Projekten mehrfach vorliegen zu haben, sondern auch gefährlich und arbeitsintensiv. Der Grund dafür ist einfach der, dass redundanter Code dazu führt, dass mehrere identische Quellcode-Dateien parallel gepflegt werden müssen. Vergisst man einmal ein Update in einer der Dateien, liegen plötzlich unterschiedliche Versionen des eigentlich gleichen Codes vor. Das kann man mit statischen Libraries umgehen. In diesem Fall ist der Quellcode zwar nicht mehr doppelt vorhanden – dafür aber der ausführbare Code, denn die Libraries werden in alle ausführbaren Dateien mit eingebunden. Besser sind DLLs, denn hier ist weder der Quellcode, noch der ausführbare Code doppelt vorhanden. Mehrere Programme können die gleiche DLL parallel benutzen.

Prinzipiell sind DLLs einfach zu haben, allerdings wird es etwas schwierig, wenn die DLL auch Klassen exportieren soll – oder Klassen aus anderen DLLs

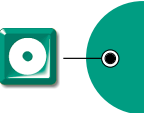
verwenden oder erweitern soll. Ein solcher Fall ist bei Windows-Programmen aber an der Tagesordnung, denn Klassen will man bei einem C++-Programm ganz sicher exportieren, und die Omnipräsenz der MFC legt natürlich nahe, dass diese Klassenbibliothek auch in einer DLL benutzt werden können muss. Darum gibt es für diesen Fall einen eige-

nen Anwendungs-Assistenten. Trotz seines Namens erzeugt dieser allerdings keine Anwendungen, sondern den benötigten Quellcode für eine DLL.

Dabei bietet der Assistent drei Spielarten an. Diese unterscheiden sich durch mehrere Aspekte. Der eine Unterschied liegt im Maße der Wiederverwendbar-



MIT DEM DEBUGGER ist schnell überprüft, ob der Zeiger auch wirklich gesetzt wird. Natürlich kann man auch im Single-Step-Modus in den Quellcode des DLL-Projekts laufen.



keit der DLL: Entweder die DLL kann von allen Win32-Programmen verwendet werden, oder Sie kann nur von solchen Programmen verwendet werden, die MFC verwenden. Ein weiterer Unterschied liegt darin, ob die DLL die MFC-Bibliothek verwenden will oder nicht. Schließlich ist es noch wichtig, ob in der DLL Klassen definiert werden sollen, die von MFC abgeleitet sind oder nicht.

■ Das Beispiel

Jetzt werden Sie eine DLL anlegen, die Sie wieder verwenden können, die die MFC nutzt und die eine eigene Klasse exportiert. Außerdem schreiben Sie noch ein Programm, das die DLL verwendet – und erfahren auch, welche Compiler und Linker-Optionen für einen solchen Vorgang wichtig sind. Fürs Beispiel-Programm wählen Sie die zweite Option des Assistenten aus: Es soll eine MFC-Extension-DLL angelegt werden. Der Assistent tut seine Arbeit und erzeugt Ihnen den minimal benötigten Quellcode, einschließlich aller Einstellungen für Compiler und Linker. Bevor Sie diese DLL nun mit eigenem Quellcode erweitern, ist es natürlich von Interesse zu erfahren, was die DLL eigentlich tun soll. An einer anderen Stelle in diesem Sonderheft haben Sie ein Windows-Programm entwickelt, das verschiedene geometrische Objekte erzeugen und anzeigen konnte. Dabei ging es aber im Wesentlichen nur um Linien und Kreise: Man kann sich nun leicht vorstellen, dass ein "echtes" Malprogramm deutlich mehr geometrische Formen unterstützten muss, es sind also weitere Klassen notwendig.

Um nun in einem solchen Programm die Arbeit in mehrere Blöcke aufzuteilen, liegt es nahe, alle geometrischen Objekte in einer DLL zu implementieren: Zum einen löst das die Implementierung aus dem Hauptprojekt aus – so kann man diese zum Beispiel auch an eine andere Person übertragen – und zum anderen ist es auf diese Weise möglich, die einmal implementierten Objekte in beliebigen zukünftigen Programmen leicht wieder zu verwenden. Das macht die Sache effektiv. Genau dies soll die DLL aus diesem Beispiel tun: Sie wird dafür zuständig sein, dass andere Programme die einmal definierten geometrischen Formen wieder verwenden können.

■ Die DLL und der Kreis

Im Rahmen dieses Beispiels wird die DLL allerdings absichtlich sehr klein

und übersichtlich gehalten: Es wird nur eine einzelne Kreis-Klasse mit einem ganz einfachen Interface definiert. Zusätzlich zu den vom Assistenten erzeugten Dateien brauchen Sie dazu zwei weitere: Eine Header-Datei, die die Informationen über die Klasse enthält, und eine CPP-Datei, in der die Implementierung der Klasse abgelegt wird. Dabei ist eines zu berücksichtigen: Anders als



DAS BEISPIELPROGRAMM erzeugt einen Kreis im sichtbaren Bereich des Fensters.

bei den bisherigen Projekten wird die Header-Datei nicht ausschließlich vom DLL-Projekt verwendet, sondern muss auch von anderen Projekten benutzbar sein. Das bedeutet inhaltlich, dass diese Datei ausschließlich Symbole verwenden

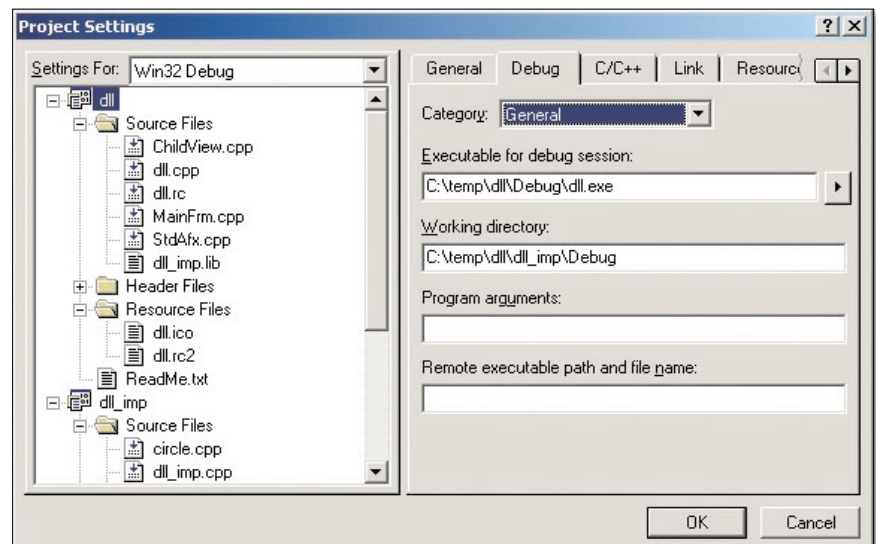
```
#ifndef _INC_CIRCLE_H
#define _INC_CIRCLE_H

class AFX_EXT_CLASS Circle
{
public:
    ..Circle( CPoint pt);
    ..Circle();
    ..void Draw( CDC* pDC);

private:
    ..Circle();
    ..CPoint m_pt;
};

#endif // _INC_CIRCLE_H
```

Dazu sind einige Dinge zu sagen. Zunächst einmal sehen Sie einige Präprozessor-Statements, die Sie vermutlich noch nicht kennen: Das sind `#ifndef`, `#define` und `#endif`. Diese Statements kümmern sich darum, dass der Inhalt der Header-Datei innerhalb einer Übersetzungseinheit nicht mehrfach inkludiert wird. Dazu prüft der Präprozessor zunächst einmal, ob das Symbol `_INC_CIRCLE_H` definiert ist. Ist das nicht der Fall, wird das Symbol zunächst definiert und dann folgt die Klassendefinition. Ist das Symbol hingegen schon definiert, wurde die Datei im Rahmen der aktuellen Übersetzung bereits einmal inkludiert – und der Präprozessor fügt dann den Codeblock bis zum abschließenden `#endif` nicht ein. Sie können sich ein `#ifdef` / `#endif`-Paar also so



DAMIT DIE DLL GEFUNDEN WIRD, müssen Sie sich in den Projekteinstellungen um den korrekten Pfad kümmern.

den darf, die auch in anderen Projekten definiert sein werden. Mit anderen Worten: Es darf keine Referenzen auf Symbole geben, die nur lokal zum DLL-Projekt bekannt sind. Im Beispielprojekt trägt die Header-Datei den Namen "circle.h" und hat folgenden Inhalt:

ähnlich vorstellen, wie einen `if()`-Block im C(++)-Quellcode – nur werden diese Bedingungen zur Compilerzeit und nicht zur Laufzeit ausgewertet. Zur Circle-Klasse selbst ist nicht viel zu sagen. Sie hat einen Konstruktor, der einen Parameter enthält – das wird der Mittelpunkt des Kreises werden – sowie



einen Destruktor. Ferner gibt es eine Methode *Draw()*, die offensichtlich für das Zeichnen des Kreises zuständig sein wird. Der private Konstruktor ohne Parameter stellt sicher, dass keine Kreise ohne Mittelpunktangabe erzeugt werden können.

Einzig das Makro `AFX_EXT_CLASS` ist noch etwas Besonderes. Dabei ist es sogar besonders wichtig, denn dieses Makro ist dafür zuständig, dass Ihre Kreis-Klasse später wieder verwendet werden kann. Und das hat einen einfachen Grund. Wie bereits erwähnt wird diese Header-Datei sowohl von dem Projekt inkludiert, das die Circle-Klasse implementiert, als auch von dem Projekt, das die Implementierung der Circle-Klasse verwenden will. Beim Implementieren der Circle-Klasse muss dem Compiler (und dem Linker) also mitgeteilt werden, dass die Klasse "Circle" exportiert werden soll. Soll die Klasse aber in einem anderen Projekt wieder verwendet werden, muss der Compiler bzw. der Linker dort wissen, dass es sich um eine zu importierende Klasse handelt. Mit einem Wort: Die Header-Datei muss in einen Fall anders übersetzt werden, als im anderen. Um dem Compiler mitzuteilen, dass es sich bei einer Klasse um eine zu im- oder zu exportierende Klasse handelt, gibt es das Schlüsselwort `__declspec`, das einen Parameter erwartet. Um nun eine Klasse zu exportieren, verwendet man `__declspec` mit dem Parameter "dllexport":

```
class __declspec(dllexport) NameDerKlasse {
```

Soll die Klasse hingegen importiert werden, so verwendet man "dllimport":

```
class __declspec(dllimport) NameDerKlasse {
```

Man hat aber natürlich nur eine Quellcode-Datei: Es bleibt also nichts anderes übrig, als die unterschiedlichen Statements vom Präprozessor erzeugen zu lassen. Das funktioniert so, dass das Makro `AFX_EXT_CLASS` in einem MFC-Extension-DLL-Projekt als "`__declspec(dllexport)`" definiert ist, während es bei einem MFC-Anwendungsprojekt als "`__declspec(dllimport)`" definiert ist. Mit anderen Worten: Je nachdem, was man gerade übersetzt, ist das `AFX_EXT_CLASS`-Makro anders definiert und das führt dazu, dass das `__declspec`-Statement richtig gesetzt ist.

Sie können so etwas auch selbst zusammenbauen. Dazu würden Sie in der Circle-Header-Datei in etwa die

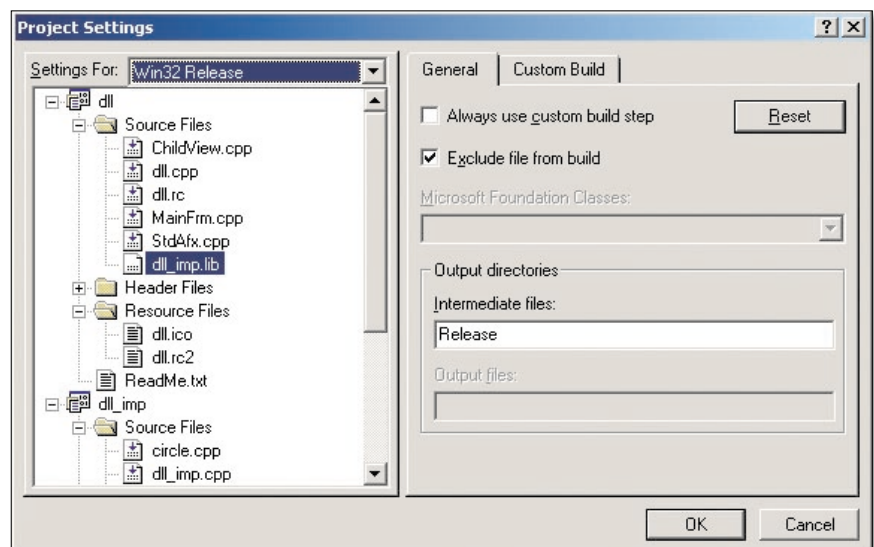
Statements wie in Listing 1 zusammenbauen.

Danach könnten Sie statt `AFX_EXT_CLASS` auch `__IMPORTEXPORT` verwenden. Sie müssten dann allerdings beim Übersetzen der DLL dafür sorgen, dass das Macro `_BUILD_DLL_CIRCLE` gesetzt ist: Das geht über die Präprozessor-Optionen im Dialog zum Einstellen der Projekt-Eigenschaften.

Im Header-File gibt es noch eine weitere Sache, die Sie klären können. Aber

der später erläutert werden wird. Allerdings sind Sie vielleicht beim "#pragma" stutzig geworden, denn hier wird plötzlich auf eine ".lib"-Datei verwiesen – dabei soll doch eigentlich eine .DLL-Datei erzeugt werden. Das ist auch richtig – aber die DLL-Datei alleine reicht nicht aus.

Immer wenn Sie eine DLL erzeugen, erzeugt der Linker auch noch eine .LIB-Datei. Dabei handelt es sich mehr oder minder um eine normale statische Bibliothek, allerdings mit einem speziellen



IN DEN PROJEKT-EINSTELLUNGEN können Sie einzelne Dateien aus dem Build ausschließen.

dazu müssen Sie sich zunächst einmal eine Frage stellen: Wenn die Circle-Klasse wieder verwendet werden soll – woher weiß der Linker in welcher Datei er nach der Implementierung suchen soll? Dazu gibt es ein weiteres Präprozessor-Statement:

```
#pragma comment(  
    ↪ lib "dll_imp.lib")
```

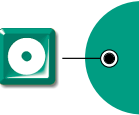
Mit diesem Statement weisen Sie den Präprozessor an, dafür Sorge zu tragen, dass der Linker im Objekt-Code einen Eintrag zu sehen bekommt, der auf die Verwendung der Library "dll_imp.lib" hinweist: Dadurch wird bekannt, dass diese Library verwendet wird, und dass Sie zu durchsuchen ist. Im Beispiel wurde allerdings ein anderer Weg gewählt,

Namen: LIB-Dateien, die zu DLL-Dateien gehören, nennt man "import library". Diese Libraries haben einen ganz speziellen Zweck. Die DLL selbst wird vom fertigen Programm erst zur Laufzeit geladen. Das bedeutet, zum Zeitpunkt des Linkens muss diese Datei noch gar nicht vorliegen. Wie aber soll dann der Linker feststellen, dass auch tatsächlich alle verwendeten Symbole aufgelöst werden können – schließlich liegen diese zum Teil in einer DLL vor?

Genau diese Frage beantwortet eine Import-Library. Diese enthält keinen ausführbaren Code, sondern nur eine Information für den Linker, welche

LISTING 1

```
#ifndef _BUILD_DLL_CIRCLE  
    #define __declspec(dllimport) __IMPORTEXPORT  
#else  
    #define __declspec(dllexport) __IMPORTEXPORT  
#endif
```



Symbole in der DLL definiert sind. Auf diese Weise kann überprüft werden, ob alle Symbole definiert sind, ohne dass die Datei, in der die Symbole vorliegen, bereits vorhanden sein müssen. Der Vorteil: Man braucht keinen speziellen Linker, der auch DLLs lesen kann.

■ Circle-Klasse implementieren

Die Implementierung der Circle-Klasse (in circle.cpp) enthält nur ein einziges Statement, das besonderer Erwähnung bedarf: In der *Draw()*-Funktion finden Sie als erstes Statement die folgende Zeile:

```
AFX_MANAGE_STATE(
    &AfxGetStaticModuleState());
```

Hier handelt es sich um eine Besonderheit von DLLs, die die MFC verwenden. Jede Funktion, in der eine MFC-Klasse verwendet wird, muss als erstes grundsätzlich das `AFX_MANAGE_STATE`-Makro verwenden – sonst kommt MFC durcheinander, und Ihr Programm stürzt ab. Wie gesagt: Das ist eine Spezialität von MFC. Bei normalen DLLs ist so etwas nicht notwendig. Sie können nun Ihr DLL-Projekt übersetzen und haben dann schon eine voll wieder verwendbare Klasse vorliegen. Dabei sollten Sie dieses Mal sowohl die Debug- als auch die Release-Version des Projektes bauen, denn Sie brauchen zur Verdeutlichung eines anderen Aspekts später im Beitrag beide Versionen.

■ Klassen verwenden

Um die Klasse nun in einem anderen Projekt zu verwenden, brauchen Sie zunächst einmal genau das: ein anderes Projekt. Das legen Sie mit dem MFC-Anwendungsassistenten an, diesmal allerdings mit der (exe)-Version. Welche der verschiedenen Optionen Sie dann auswählen, spielt weiter keine Rolle – Sie sollten aber auf jeden Fall eine SDI- oder MDI-Anwendung erstellen lassen, denn für die Anzeige des Kreises brauchen Sie im Beispiel ein echtes Fenster. Das Beispielprogramm auf der Heft-CD ist als SDI-Anwendung angelegt.

Nachdem der Assistent das Projekt angelegt hat, öffnen Sie den Quellcode der View-Klasse, und zwar zunächst die zugehörige Header-Datei. In dieser müssen Sie nun den Header der Circle-Klasse inkludieren. Dabei müssen Sie den Pfad zu dieser Datei mit angeben: Ihr Include-Statement sieht also in etwa folgendermaßen aus:

```
#include "projekte\
    circle\circle.h"
```

Dabei ist Ihnen vielleicht aufgefallen, dass es zwei unterschiedliche Arten gibt, Header-Dateien per `include` zu inkludieren. Die eine Methode verwendet spitze Klammer (`<` und `>`), die andere doppelte Hochhaken. Im Falle der spitzen Klammern werden die angegebenen Header-Dateien in einem speziellen Pfad gesucht, nämlich in dem, der anhand der Environment-Variable "include" definiert ist. Wird ein Programm nicht auf der Kommandozeile, sondern innerhalb der IDE übersetzt, gilt allerdings nicht die Environment-Variable, sondern die entsprechenden Konfigurationseinträge aus den Optionen des Developer Studio.

Header-Dateien, die mit doppelten Hochhaken inkludiert werden, werden im angegebenen Pfad gesucht – ist keiner angegeben, muss die Header-Datei

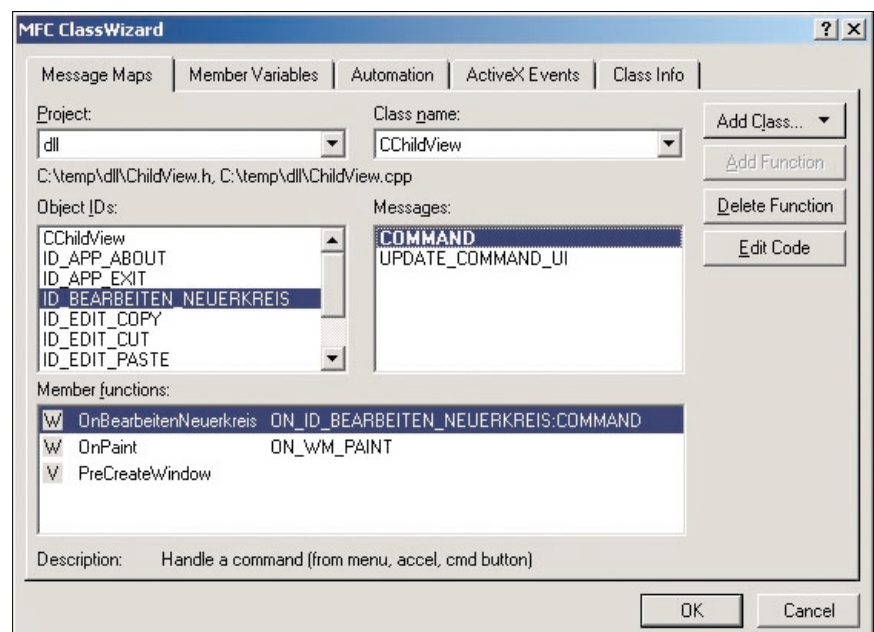
weiterhin mit einem Zeiger auf ein Circle-Objekt:

```
Circle* m_pCircle;

Diesen Zeiger initialisieren Sie im Konstruktor der View-Klasse mit 0. Außerdem erweitern Sie die OnPaint()-Methode der Viewklasse mit ein wenig Code, der sich darum kümmert, einen Kreis anzuzeigen, wenn der Pointer gesetzt ist:
```

```
if ( m_pCircle)
{
    m_pCircle->Draw( &dc);
}
```

Schließlich müssen Sie noch einen Befehl implementieren, mit dem ein neuer Kreis angelegt wird. Das tun Sie wie bereits gewohnt mit dem Menü-Editor und dem Klassenassistenten. Im Beispielprogramm wird der Mittelpunkt für den Kreis dabei mit einer auf Zu-



MIT DEM KLASSENASSISTENTEN erzeugen Sie eine Funktion, die den Befehl zum Anlegen eines Kreises implementiert.

im gleichen Verzeichnis sein, wie der inkludierende C(++)-Quellcode.

Wenn Sie vorhaben im Laufe der Zeit eine Sammlung von wieder verwendbaren Klassen anzulegen, sollten Sie deren Header-Dateien in einem festen Verzeichnis aufbewahren: Wenn Sie dann dieses Verzeichnis in der Environment-Variablen "include" bzw. den entsprechenden Developer-Studio-Optionen mit angeben, brauchen Sie keinen Pfad mehr dafür anzugeben und die Datei wird vom Präprozessor trotzdem gefunden. Jetzt können Sie die View-Klasse um ein neues Member er-

fallszahlen basierenden Koordinate belegt, die sich im sichtbaren Bereich des Fensters befindet (siehe Listing 2).

Jetzt können Sie das Programm zusammenbauen lassen. Sind Compiler und Linker mit ihrer Arbeit fertig, bekommen Sie allerdings eine ärgerliche Meldung: Der Linker sagt, dass er ein nicht aufzulösendes externes Symbol gefunden hat: den Kreis.

Der Grund dafür ist aber einfach: Die Kreisklasse liegt ja in einer DLL vor, und für den Linker haben Sie nur eine Import-Library herstellen lassen. Diese befindet sich aber im "Debug" bzw. "Re-



LISTING 2

```
void CChildView::OnBearbeitenNeuerkreis()
{
    if( m_pCircle)
    {
        delete m_pCircle;
    }

    CRect rect;
    GetClientRect( &rect);

    int x=10000;
    int y=10000;

    while( x > rect.Width())
        x = rand();

    while( y > rect.Height())
        y = rand();

    m_pCircle = new Circle( CPoint(x, y));
    Invalidate();
}
```

lease"-Verzeichnis des DLL-Projekts – und da kann Sie der Linker natürlich nicht finden. Ähnlich wie für Header-Dateien gibt es aber auch einen Suchpfad für Libraries.

Sie könnten also Ihre Library auch in einem festgelegten Verzeichnis ablegen und dem Linker den Suchpfad per Environment-Variable oder Developer-Studio-Einstellung mitteilen. Wenn Sie dann außerdem noch das "lib" #pragma

in der Circle-Header-Datei verwendet hätten, würde der Linker die Datei automatisch finden.

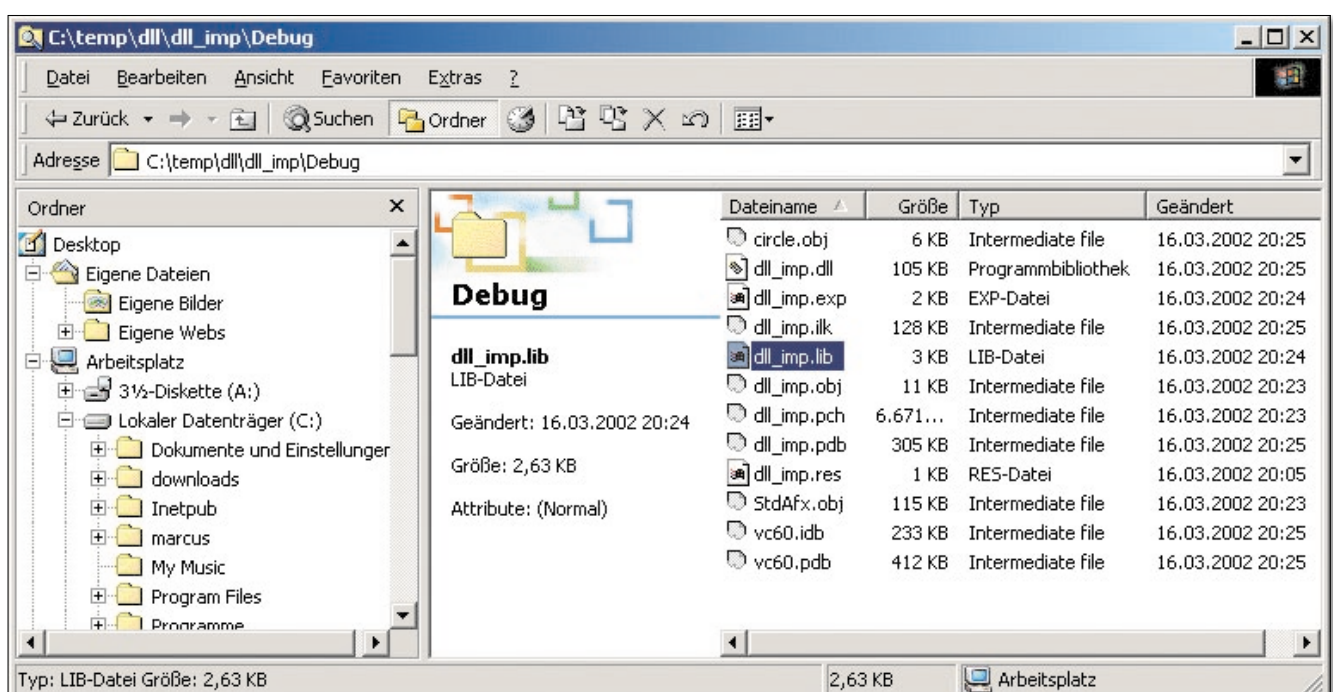
Es geht aber auch anders: Sie können die Library nämlich einfach in Ihr Projekt mit aufnehmen. Dazu fügen Sie die .LIB-Datei einfach dem Projekt hinzu – ganz so, als wäre es eine Datei mit Quellcode.

Dabei stellt sich aber ein Problem. Sie haben zwei Versionen der Import-Li-

brary: Eine "Release"-Variante und eine "Debug"-Variante – und eigentlich müssen beide ins Projekt aufgenommen werden. Allerdings darf die eine nur beim "Debug" Build und die andere nur beim "Release" Build verwendet werden. Das lässt sich aber mit den "Eigenschaften" dieser Datei in den Projekteinstellungen verwirklichen. Dort gibt es nämlich die Möglichkeit, eine Datei beim Build-Prozess aus einer bestimmten Konfiguration auszuschließen.

Ist die Library ins Projekt aufgenommen, läuft der Linker ohne Fehlermeldung durch. Wenn Sie aber nun versuchen, Ihr Programm zu starten, erhalten Sie wieder eine Fehlermeldung. Diesmal kann die DLL-Datei nicht gefunden bzw. geladen werden. Aber auch das ist eigentlich klar, denn diese DLL befindet sich ja auch im "Debug"-Verzeichnis des anderen Projektes – und das ist ganz sicher nicht im Suchpfad enthalten.

Auch dieses Problem ist lösbar: Unter den "Debug"-Einstellungen für Ihr Projekt können Sie ein Verzeichnis angeben, in dem das Programm gestartet werden soll. (Die Position der .EXE-Datei auf der Festplatte spielt dabei keine Rolle). Wenn Sie dort einstellen, dass das Programm in dem Verzeichnis ausgeführt werden soll, in dem sich die DLL befindet, können Sie es auch starten – und verwenden dann eine eigene Klasse aus einem anderen Projekt: herzlichen Glückwunsch. UR



DIE IMPORT-LIBRARY (LIB) ist nur für den Linker da – der ausführbare Code befindet sich in der DLL-Datei.