Datenverwaltung unter MFC

Ziel, Inhalt

- Wir lernen mit der MFC Daten zu verwalten und die Document-View Architektur für unsere Zwecke auszunutzen.

Datenverwaltung unter der MFC

Einführung

Die MFC implementiert das Observer Pattern. Es heisst hier Document-View, wobei das Document die Rolle des Subject's übernimmt und die View als Observer, beziehungsweise als Controller fungiert. Mehr zum Observer Pattern findest du im Unterrichtsstoff zum 5. Semester.
www.devmentor.ch/teaching/additional/00I/Semester5/Abend10/Abend10.pdf

Die CDocument Klasse

In dieser Klasse werden die Daten, die für unsere Anwendung wichtig sind gehalten, gespeichert und wieder geladen. Sie wendet dazu die so genannte Serialisierung an. Sie biete auch die Möglichkeit alle Observer, die im Allgemeinen von der CView Klasse abgeleitete Objekte sind über Änderungen zu informieren.

Serialisierung

Dieser Ausdruck bedeutet im Grunde genommen die Fähigkeit von Objekten, ihren Inhalt, ihren Zustand in einen Strom zu senden um sich am anderen Ende des Stromes wieder selbst aufzubauen. Üblicherweise ist der verwendete Strom eine Datei-Strom, ein „file-stream". Das Objekt landet dadurch als Datenstrom auf der Harddisk und kann aus diesem Datenstrom wieder erzeugt werden.

Die Klasse CObject

Die MFC schreibt dazu nicht nur die Daten von einem Objekt in den Datenstrom sondern auch Information über den Datentypen. Befindet sich ein Objekt der MFC auf einer Harddisk, kann in den ersten Bytes ein wenig Information über die Klasse des Objektes drin stehen. Da es aber meistens nicht möglich ist die Objekte in einer beliebigen Reihenfolge wieder zu erstellen, wird diese Fähigkeit selten ausgenutzt. Die Objekte der MFC, die serialisiert werden können sind alle von der Klasse CObject abgeleitet, die eine Methode *Serialize* anbietet. Hinzu kommen Methoden, mit denen man zur Laufzeit, den Datentypen von einem Objekt feststellen kann.

Serialisierungsmakros

Diese Fähigkeit von CObject abgeleiteten Klassen, den Datentypen zur Laufzeit feststellen zu können und die Serialisierung zu unterstützen wird durch gewisse Makros gewährleistet. Hier zuerst, die Makros, die in die Deklaration der Klasse gehören:

```
DECLARE_DYNAMIC(Klassenname)
DECLARE_DYNCREATE(Klassenname)
DECLARE_SERIAL(Klassenname)
```

Diese Makros deklarieren ein/zwei Methoden, mit denen man den Datentypen feststellen kann. Je nach Makro kommen noch weitere Methoden hinzu, so definiert das DECLARE_SERIAL Makro einen stream-operator << mit dem man das Objekt serialisieren kann. Damit sie funktionieren gehören in die Quellcode-Dateien folgende Makros:

```
IMPLEMENT_DYNAMIC(Klassenname, Basisklassenname)
IMPLEMENT_DYNCREATE(Klassenname, Basisklassenname)
IMPLEMENT_SERIAL(Klassenname, Basisklassenname, Version)
```

Je nach Anwendung kann eine Form der Makros verwendet werden. Mit jeder Form wird eine Methode *IsKindOf* erzeugt, die man so verwenden kann:

```
CObject* pObject = … // irgendwie einen Zeiger auf ein
                     // Objekt holen (z.B. aus Kollektion)
if(pObject->IsKindOf(CRuntimeClass(CMyClass))
{
    // das Objekt is vom Typ CMyClass
    // also ist der folgende type cast sicher
    CMyClass* pMyClass = (CMyClass*)pObject;
}
```

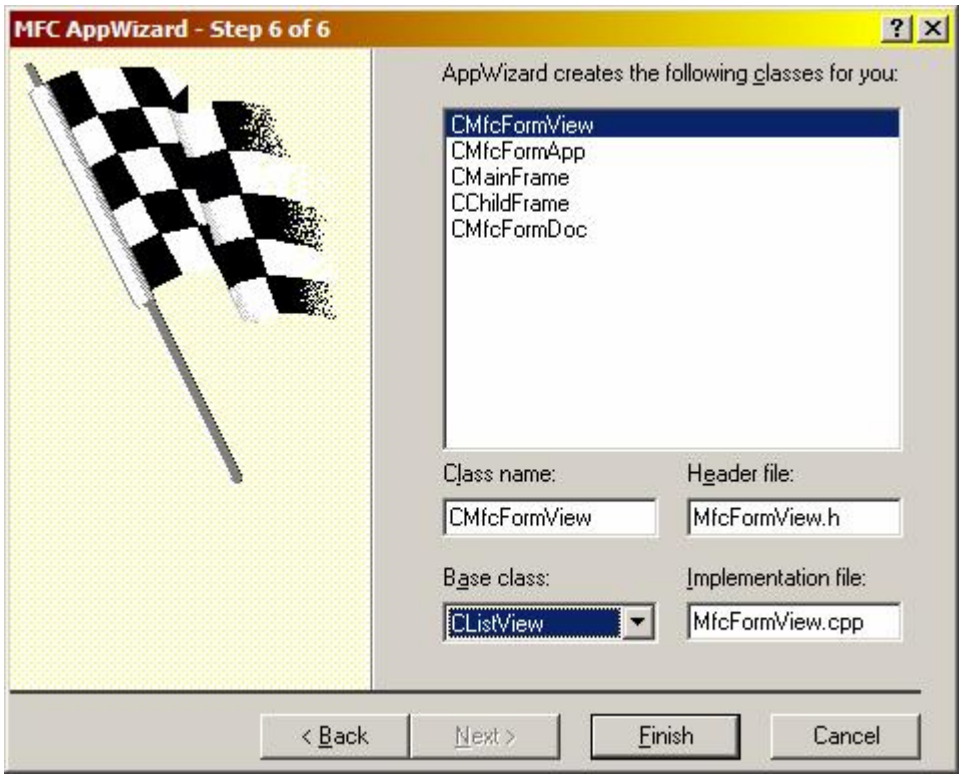
Ich persönlich finde solchen Code nicht gut, man sollte eher darauf achten, dass man die Strukturen, das Design so auslegt, dass man immer den richtigen Datentypen hat. Die Tendenz ist es nämlich, dass man alle Klasse von einer Basisklasse wie CObject ableitet und beliebige Objekte in eine Kollektion von CObject Objekten packt. Ob sich hinter einem Objekt dann ein string oder ein Kriegsschiff versteckt lässt sich nur durch dynamische Typabfrage feststellen. Wie gesagt, ich vermute dahinter meistens ein ungenügendes Design, aber hier können die Meinungen auseinander gehen.
Das dritte Makro (DECALRE_SERIAL) verlangt die Implementation einer *Serialize* Methode.

Beispiel Serialize

Versuchen wir nun ein Projekt zu erzeugen, mit dem wir das Serialisieren von Objekten ausprobieren können.

Projekt erzeugen

Wir erzeugen ein MFC Programm (exe). Am besten wir wählen eine SDI-Applikation. Als Basisklasse für die CView Klasse können wir die CListView Klasse bestimmen.



Wähle als Basisklasse für die View CListView

Serialisierbare Notenklasse

Erzeuge mit dem Assistenten eine neue Klasse CNote. Das C vor dem Namen verwenden wir nur weil wir uns dem Diktat von Microsoft beugen ;-) und dadurch auch anzeigen, dass die Notenklasse sich verhält wie viele andere MFC-Klassen. Wähle als Basisklasse also CObject und überschreibe die Methode *Serialize*.

```
IMPLEMENT_SERIAL(CNote, CObject, 1);

void CNote::Serialize(CArchive& ar)
{
    if(ar.IsStoring())
    {
        ar << m_value;
    }
    else
    {
        ar >> m_value;
    }
}
```

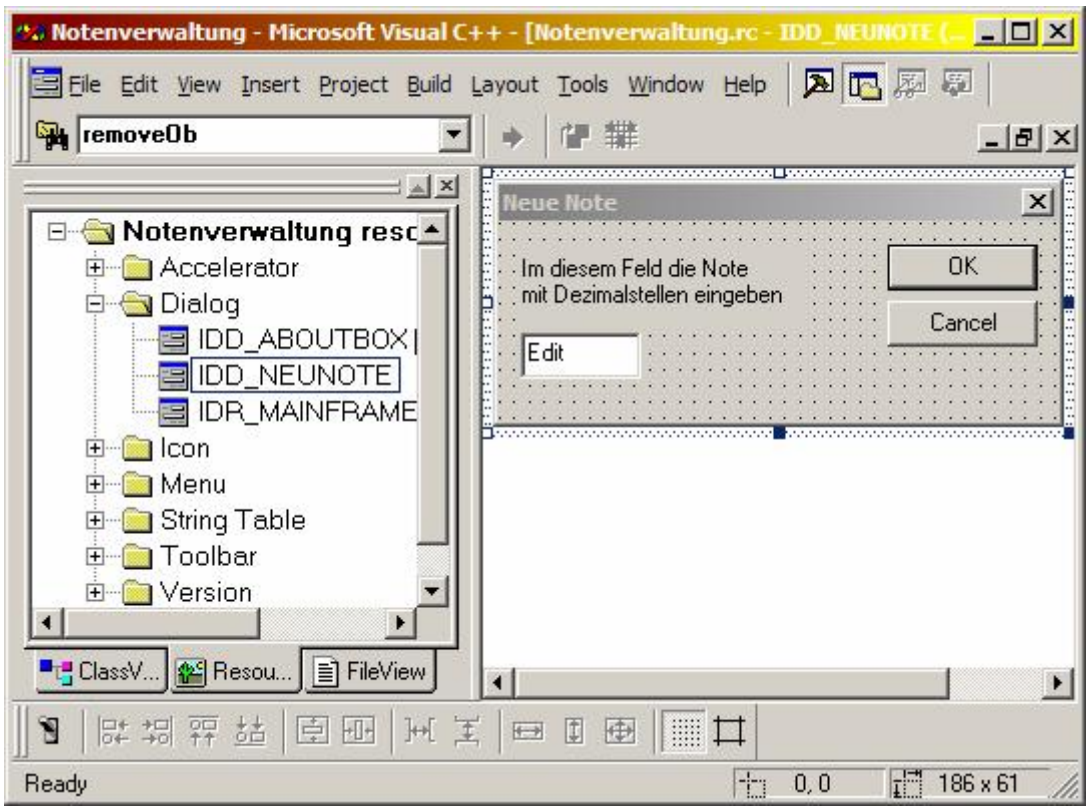
Vergiss auch nicht das DECLARE_SERIAL Makro in der Headerdatei und das IMPLEMENT_SERIAL in der Quellcode-Datei. Wie es kommt, dass diese Methode aufgerufen wird, sehen wir gleich.

Array von Noten

Wir können nun ein Array erzeugen und dieses in die CDocument Klasse einfügen. Dabei haben wir die Möglichkeit eine Klasse aus der Standardlibrary wie vector oder list zu verwenden, oder wir beugen uns wieder dem Diktat von Microsoft und verwenden eine Kollektion aus der MFC, die uns beim Serialisieren hilft.

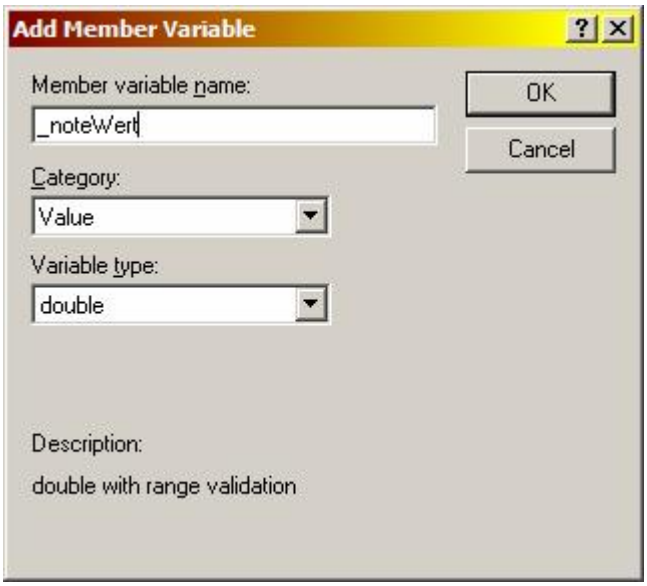
Erzeugen von Noten mit Dialog

Zuerst wollen wir eine Möglichkeit haben, einzelne Noten einzugeben. Der einfachste aber nicht der eleganteste Weg ist es ein Dialog zu erzeugen. Erzeuge nun also einen neuen Dialog (z.B. mit der rechten Maustaste in der Resourcenansicht).



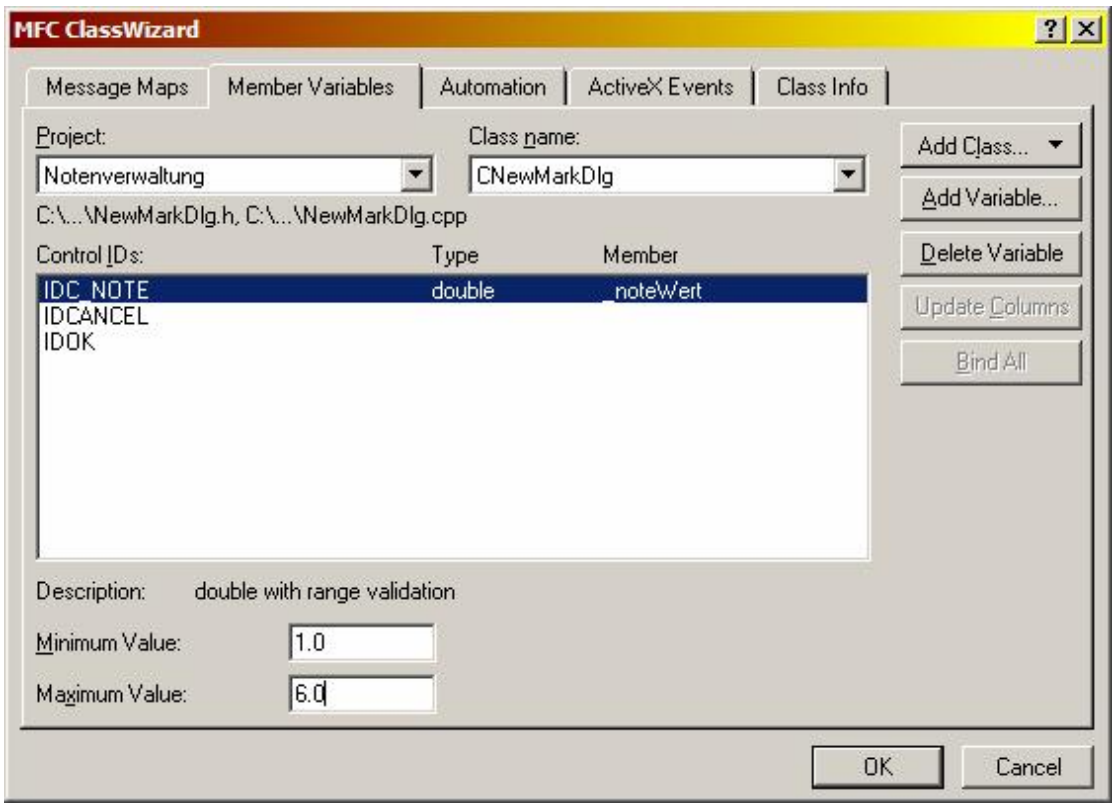
So könnte ein Dialog zum eingeben von Noten aussehen

Durch Doppelclick auf den Dialog (nicht ein Control auf dem Dialog doppelklicken) wird der Klassenassistent gestartet, mit dem man eine Klasse für den Dialog erzeugen kann.
Mit dem Klassenassistenten können wir auch ein Datenelement in die Dialogklasse einfügen, die dem double Wert aus dem Edit Feld entspricht. Im Gegensatz zu der Übung mit den Controls wählen wir als Datentypen für das Feld double wählen.



Mit dem Klassenassistenten fügen wir ein Datenelement in die neue Dialog-Klasse ein

Wir haben sogar die Möglichkeit Grenzen für diesen Wert einzugeben.

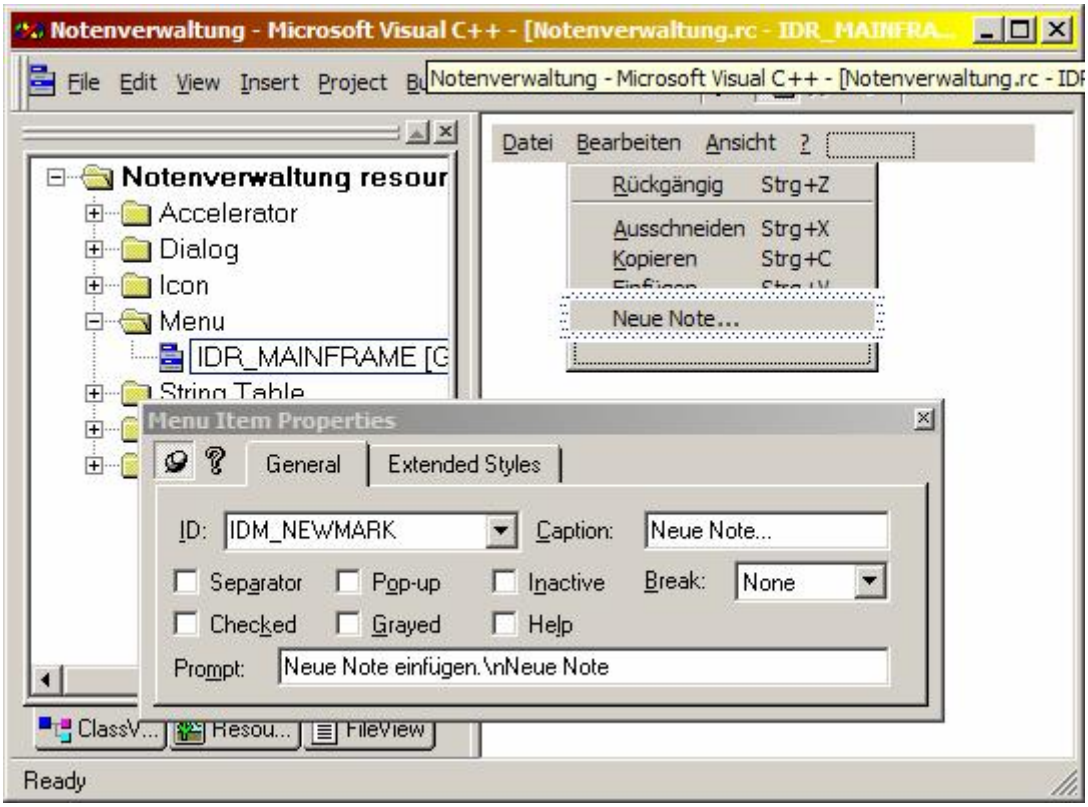


Feld mit Grenzen

Ergänze in der Dialogklasse (hier CNewMarkDlg) eine Methode getNote.

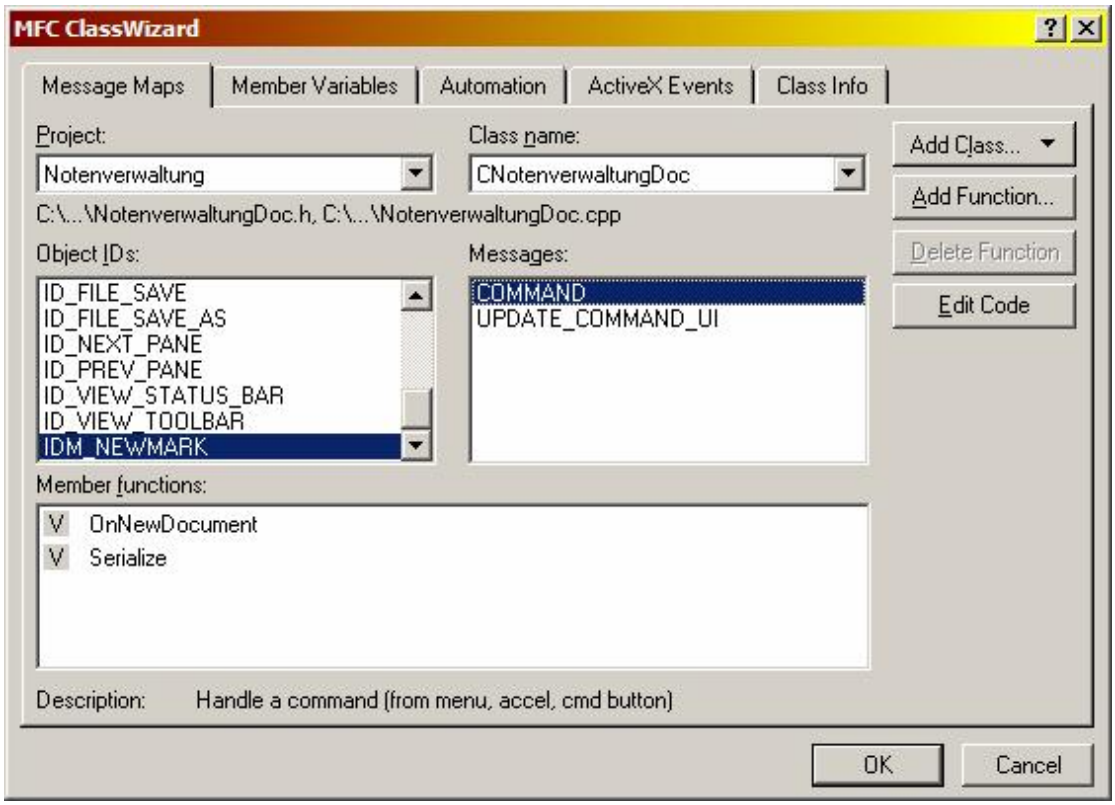
Dialog mit Menupunkt aufrufen

Diesen Dialog können wir nun aufrufen indem wir ein Objekt dieser Klasse (hier CNewMarkDlg) erzeugen und die Methode *DoModal* aufrufen.
Wir werden diesen Dialog aufgrund eines Menubefehls anzeigen lassen. Zuerst fügen wir im Resourceneditor einen Menupunkt ein.



So sieht der neue Menupunkt aus

Es ist nicht schwer eine Bearbeitungsmethode für diesen Menupunkt erzeugen zu lassen, der Assistent wird das für uns übernehmen. Schwieriger ist die Frage zu welcher Klasse wir diese Methode hinzufügen. Wir haben die Möglichkeit diese Methode von der CMainFrame-, von der CView- oder von der CDocument-Klasse bearbeiten zu lassen.
Da der Dialog dazu dient direkt eine Note zu erzeugen und diese schlussendlich in die CDocument-Klasse einzufügen, kann sich das CDocument Objekt selber darum kümmern. Der Klassenassistent bietet diese Möglichkeit an (Ctrl+W):



Klassenassistent vor dem Einfügen des Menubehandlers

Die Methode in der meiner CXXXDocument-Klasse sieht so aus :

```
void CNotenverwaltungDoc::OnNewmark()
{
    // Dialog erzeugen
    CNewMarkDlg dlg;

    // der Dialog gibt IDOK zurück
    // falls die OK Taste gedrückt wird
    if(IDOK == dlg.DoModal())
    {
        // holen wir uns den Notenwert
        double notenWert = dlg.getMark();
        // Wir erzeugen ein Notenobjekt
        // merken uns aber nur den CObject*
        CObject* notenObject = new CNote(notenWert);
        // Diesen Zeiger fügen wir in
        // unser Array ein
        _notenArray.Add(notenObject);
        // wir markieren das Dokument
        // als modifiziert
        SetModifiedFlag();    }
}
```

Vergiss nicht die nötigen #include - Direktiven (für die Dialog Klasse und für die Noten Klasse) einzufügen.

Was auch wichtig ist, dass wir im Destruktor die erzeugten Objekte wieder löschen.

```
CNotenverwaltungDoc::~CNotenverwaltungDoc()
{
    int size = _notenArray.GetSize();
    for(int i = 0; i < size; ++i)
    {
        CObject* object = _notenArray[i];
        delete object;
    }
    _notenArray.RemoveAll();
}
```

Auch hier verwenden wir nur die CObject Zeiger. Da der Destruktor virtuell ist, wird der Destruktor vom Notenobjekt trotzdem aufgerufen.

Die Noten serialisieren

Die Klasse CObArray, die wir hier verwenden weiss selber, wie sie sich archivieren muss. Es genügt in der CXXXDocument den folgenden Code zu ergänzen:

```
void CNotenverwaltungDoc::Serialize(CArchive& ar)
{
    _notenArray.Serialize(ar);

      if (ar.IsStoring())
      {
            // TODO: add storing code here
      }
      else
      {
            // TODO: add loading code here
      }
}
```

Die Aufgabe der View

Wir haben eine von CView abgeleitete Klasse, die wir im Moment noch nicht verwenden. Da diese Klasse von CListView abgeleitet ist, eignet sie sich ausgezeichnet um ein Array von Daten darzustellen.

Die Note als String

Zur Anzeige werden wir die Noten als String brauchen. Wir fügen also eine Methode zu der Klasse CNote hinzu:

```
CString CNote::asString() const
{
    CString valueString;
    valueString.Format("%.2f", _value);
    return valueString;
}
```

Hier sehen wir die Anwendung der Methode *Format* der Klasse CString.

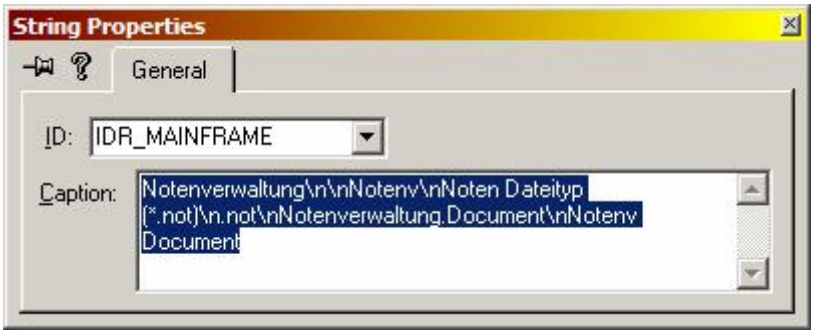
Die OnUpdate – Methode der View

Die OnUpdate Methode der View wird aufgerufen wenn die Methode UpdateAllViews der CDocument-Klasse aufgerufen wird. Werden die Daten also so geändert, dass die View's benachrichtigt werden müssen, genügt es UpdateAllViews aufzurufen. Dieser Aufruf kann in der Methode, in der wir den Notendialog darstellen ergänzt werden.
Die Methode OnUpdate muss in unserer CView – Klasse noch eingefügt werden:

```
void CNotenverwaltungView::OnUpdate(CView* pSender,
                                    LPARAM lHint,
                                    CObject* pHint)
{
    // zuerst einen Zeiger auf das Document Objekt holen
    CNotenverwaltungDoc* doc = GetDocument();
    // holen wir uns das Array der Noten
    CObArray& array = doc->getNotenArray();
    // Die Klasse CListView enthält ein ListCtrl
    CListCtrl& list = GetListCtrl();
    // Liste löschen
    list.DeleteAllItems();
    // Schleife über alle Objekte
    const int count = array.GetSize();
    for(long i = 0; i < count; ++i)
    {
        // hier erhalten wir nur einen
        // CObject Zeiger
        CObject* object = array[i];
        if(object->IsKindOf(RUNTIME_CLASS(CNote)))
        {
            // das Objekt ist von der richtigen
            // Klasse, darum wird ein typecast
            // riskiert
            CNote* note = (CNote*)object;
            CString data = note->asString();
            // in liste einfügen
            list.InsertItem(i, data);
        }
    }
}
```

Verbinden mit einer Dateiendung

Man kann beim Erzeugen des Projektes eine Datei- Endung definieren. Falls du das verpasst hast, lässt sich das „nachrüsten". Es gibt in den Resourcen eine Textresource mit der ID IDR_MAINFRAME. Die sollte etwas so geändert werden:



Hier wird die Datei – Endung ergänzt

Damit das Programm auch gestartet wird, wenn eine Datei mit dieser Endung (hier .not) doppelt angeklickt wird, müssen wir noch folgende zwei Zeilen in der InitInstance Methode der Applikationsklasse:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
      IDR_MAINFRAME,
      RUNTIME_CLASS(CNotenverwaltungDoc),
      RUNTIME_CLASS(CMainFrame),
      RUNTIME_CLASS(CNotenverwaltungView));
AddDocTemplate(pDocTemplate);

// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);

CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
```