Visuelle Zauberei oder Visual C++

Silke Seehusen, Fachhochschule Lübeck

Visual C++ dient der einfachen Entwicklung und teilweisen Generierung von Anwendungen mit grafischen Benutzungsoberflächen für Systeme von Microsoft wie Windows 95 und Windows NT. Es wird untersucht, wie leicht und visuell intuitiv es wirklich ist, eine solche Anwendung zu erstellen, und wie gut der erzeugte Quellcode weiterverwendet werden kann.

Nachdem wir uns mit C++Bibliotheken beschäftigt haben und auch beispielhaft die Implementierung von Software nach Entwurfsmustern demonstriert haben, soll es in diesem Artikel um die Benutzung eines bekannten Gerüstes gehen, des Frameworks, das mit Visual C++ von Microsoft ausgeliefert wird. Es soll nicht im Detail darum gehen, wie durch welchen Mausklick welche Aktion hervorgerufen wird, sondern eher darum, was für Code erzeugt wird, wie einfach es ist, im erzeugten Code die Stellen mit Leben zu füllen, damit es die gewünschte Anwendung wird, wie änderungsfreundlich der erzeugte Code ist und welche Anwendungen sinnvoll mit dem Framework implementiert werden können.

Der Aspekt, daß Visual C++ eine Entwicklungsumgebung zur Entwicklung allgemeiner C++Programme ist, wird hier nicht näher beschrieben. Details können in der Begleitdokumentation von Visual C++ oder auch z.B. in [1] nachgeschlagen werden.

Eine leere Applikation

Das Gerüst, das zur Verfügung gestellt wird, erlaubt es, sehr schnell eine Anwendung in C++ zu entwickeln, die ein Fenster mit normalen Voreinstellungen, z.B. unter Windows 95, öffnet und schon auf ein paar Benutzeraktionen vernünftig reagiert.

Zunächst, so lernen wir es auch in der Softwaretechnik, definieren wir dazu ein neues Projekt und nennen es *Leer*. Der Entwicklungsumgebung von Visual C++ müssen wir dazu den Namen des Projekts sagen und dann erst einmal mit allen Voreinstellungen einverstanden sein und viermal den richtigen Ok-Knopf (manchmal heißt er *Create* oder *Next* und zum Schluß *Finish*) drücken.

Voreingestellt wird ein Projekt vom Typ *MFCAppWizard(exe)* erzeugt und sofort Quellcode zu der Anwendung generiert, die vom Typ ein *Multiple Document Interface* (MDI) ist. Werden dann in der Entwicklungsumgebung die Knöpfe zum Übersetzen und Binden der Anwendung und dann zum Ausführen gedrückt, wird die voll lauffähige Anwendung *Leer* ausgeführt und erscheint auf dem Bildschirm wie in Abbildung 1.

Abbildung 1: Lauffähige Anwendung

In dem Hauptfenster der Applikation können mehrere Dokumente dargestellt werden, in Abbildung 1 wird genau eins dargestellt. Jedes Dokument kann eine oder mehrere Sichten besitzen. In Abbildung 1 hat es eine Sicht.

Die entsprechenden Menüpunkte und auch die Funktionstasten, Symbolleisten und Statusanzeigen sind sinnvoll vorbelegt. Die Ansicht kann verändert werden, es gibt ein voreingestelltes "About" hinter dem Fragezeichen.

Das dargestellte Dokument kann abgespeichert werden. Es wird dann eine leere Datei erzeugt. Das Dokument kann sogar gedruckt werden, und dabei kommt aus dem Drucker eine leere Seite raus. Semantisch korrekt!

Die generierten Klassen

Gezaubert wurde die Anwendung vom sogenannten *AppWizard*. Erzeugt wurden fünf Klassen:

CLeer	Anwendung
CMainFrame	Hauptfenster
CLeerDoc	Dokument
CLeerView	Sicht des Dokuments
CAboutDlg	About-Dialog

Die Klassen befinden sich in Dateien, deren Namen aus dem Klassennamen abgeleitet werden, wie die Klassen *CLeer* und *CAboutDlg* in *Leer.h* und *Leer.cpp*.

Bei den Dateinamen wird das beginnende C der Klassennamen weggelassen. Die Namenskonvention, daß alle Klassennamen mit einem großen C beginnen, auch als ungarische Namenskonvention tituliert, zieht sich durch den gesamten generierten Quellcode und auch durch die benutzte Klassenbibliothek, die *Microsoft Foundation Classes* (MFC), die eine zentrale Rolle in Visual C++ spielen.

Des weiteren wurden ein *Makefile* und Ressourcendateien für Ikonen, die Symbolleiste (*toolbar*) und den About-Dialog generiert.

Eine solche leere Multi-Dokument-Applikation besteht aus 56 KB erzeugtem Quellcode in 18 Dateien inklusive Ressourcen-Beschreibungen und Standardtexten wie About. Ausführbar mit Debug-Informationen werden es 4,243 MB, sogar komprimiert zu groß für eine 1,4 MB Diskette. Aber es tut ja auch schon einiges.

Die Stellen im erzeugten Quellcode, an denen die "Inhalte" oder die Verbindung zu den Inhalten eingefügt werden sollen, sind durch Kommentar, der mit *TODO* beginnt, gekennzeichnet, meistens noch mit einer kleinen Erklärung wie z.B.

```
CLeerApp::CLeerApp()
{
 /*
 TODO: add construction
 code here, Place
 all significant initia—
 lization in InitInstance
 */
}
```

Auf den netten Hinweis kommen wir weiter unten noch zurück.

Hierarchie der Basisklassen

Der Quellcode der erzeugten Klassen selbst ist relativ klein und übersichtlich, denn das meiste wird aus Basisklassen abgeleitet. Und damit endet ein wenig die Übersicht. Denn die Eigenschaften der erzeugten Klassen können erst erahnt werden, wenn auch die Eigenschaften der Basisklassen bekannt sind. Eine Ableitungshierarchie der erzeugten Klassen, die letztendlich alle von der Klasse *CObject* aus der MFC abgeleitet sind, läßt sich leicht mit der Entwicklungsumgebung erstellen und ist in Abbildung 2 dargestellt.

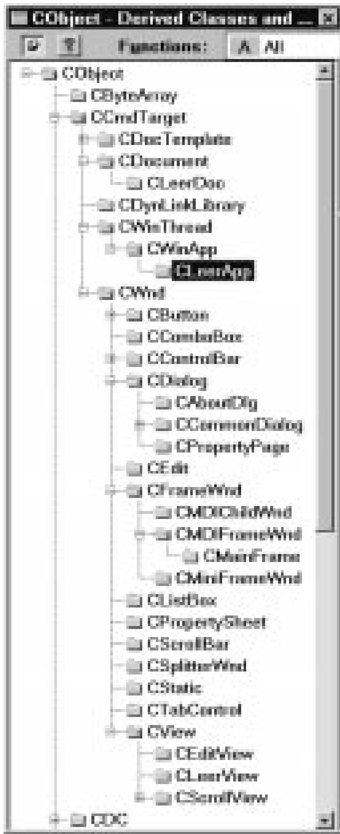

Abbildung 2: Basisklassen der erzeugten Klassen

Die Anwendung *CLeerApp* wird von *CWinApp* abgeleitet, die die gesamte Funktionalität enthält, die das äußere Fenster aus Abbildung 1 darstellt.

Die Rolle des Hauptprogramms spielt die Datei, in der die Klasse der Anwendung, hier *CLeerApp*, implementiert wird. Dort wird auch genau ein Objekt der Klasse erzeugt und treffend *theApp* genannt:

```
// The one and only
// CLeerApp object
CLeerApp theApp;
```

Soll darauf auch aus anderen Dateien zugegriffen werden, so muß sie per Hand in der entsprechenden Definitionsdatei als *extern* deklariert werden. Da der Code lesbar ist, sind solche kleinen Änderungen kein Problem.

Aus der Klassenhierarchie erkennen wir die Rollen der erzeugten Klassen, Hauptrahmen, Anwendung, Dokument, Sicht und About-Dialog.

Dokument-Sicht-Architektur

Der *AppWizard* benutzt ein Programmgerüst, ein Framework, und erzeugt die wichtigsten anwendungsspezifischen Klassen, in die dann der Code der Anwendung eingebettet oder angehängt wird. Das Gerüst wird durch die *Dokument-Sicht-Architektur* (*Document-View-Architecture*) beschrieben:

Eine Anwendung besitzt ein Dokument zur Datenhaltung und eine oder mehrere Sichten des Dokuments bzw. der Daten. Diese Architektur wird *Single Document Interface* (SDI) genannt. Besitzt eine Anwendung mehrere Dokumente, die wiederum jeweils eine oder mehrere Sichten haben können, so entspricht sie dem *Multiple Document Interface* (MDI). Die oben erzeugte Anwendung *Leer* ist von diesem Typ. Aber insbesondere sind sehr viele Windows-Anwendungen von diesem Typ, z.B. die Entwicklungsumgebung von Visual C++, der Acrobat Reader oder ein HTML-Browser wie Netscape.

Der *AppWizard* kann als dritte Art eine dialogbasierte Anwendung erzeugen, bei der die Anwendung aus einem Dialog besteht. Wenn Sie eine andere Art von Anwendung erstellen wollen, müssen Sie auf die Generierung des Programmgerüstes verzichten.

Und wieder Hallo

Wir erzeugen ein neues Projekt *Hallo*. An diesem Beispiel sollen triviale Ergänzungen vorgenommen werden, um die Dokument-Sicht-Architektur zu illustrieren, das aber von der Anzahl Ergänzungen her noch sehr übersichtlich ist.

Zunächst soll jedes Dokument, das geöffnet wird, uns mit Hallo begrüßen. Eine schon erweiterte Version der Anwendung ist in Abbildung 3 dargestellt.

Abbildung 3: Hallo

Das Dokument respektive die Klasse *CHalloDoc* bekommt ein Elementobjekt mit dem Namen *m_data* vom Typ *CString* aus der MFC. Nach Microsoft-Stil beginnen (fast) alle Elementobjekte mit *m_* und sind öffentlich. In der Definitionsdatei *HalloDoc.h* muß somit an geeigneter Stelle die Zeile

```
CString m_data;
```

ergänzt werden. *CString* ist das MFC-Äquivalent zur Klasse *string* der Standard-C++-Bibliothek.

Der nächste Schritt sollte sein, *m_data* im Konstruktor von *CHalloDoc* zu initialisieren. Nach der Idee der Konstruktoren in C++ sollte es so sein, aber bei Microsoft besser nicht. Die vom *AppWizard* generierten Kommentare deuten auch schon darauf hin. Der Grund liegt wohl darin, wie ich bei einem größeren Beispiel schmerzlich lernen mußte, daß die Reihenfolge der Aufrufe der verschiedenen Konstruktoren der Klassen des MDI-Gerüstes in der Bibliothek versteckt und auch nicht beschrieben ist, somit kann sich die Reihenfolge bei jeder neuen Version ändern.

Dafür gibt es bei fast jeder Klasse eine Methode zur Initialisierung, die irgendwann nach dem Konstruktor aufgerufen wird. Bei einem Dokument ist es die Methode *OnNewDokument()*, die wir jetzt um eine Zeile ergänzen, so daß sie folgendermaßen aussieht:

```
BOOL CHalloDoc::
OnNewDocument()
{
 if ( !CDocument::
       OnNewDocument() )
        return FALSE;
/*
TODO: add reinitialization
code here
(SDI documents will
reuse this document)
*/
 m_data =
      "Hallo"; //ergaenzt
 return TRUE;
```

```
}
```

Diese Methode zeigt, wie typischerweise Code erzeugt und durch Kommentare darauf hingewiesen wird, was wo eingefügt werden sollte.

Das Dokument enthält jetzt die Daten, dargestellt werden sie von der dazugehörigen Sicht. Diese ist in der generierten Klasse *CHalloView* realisiert. Immer, wenn eine Sicht etwas darstellen soll, wird ihre Methode *OnDraw()* aufgerufen. Diese ergänzen wir um die Ausgabe des Elementobjekts *m_data* des dazugehörigen Dokuments:

```
void CHalloView::
OnDraw(CDC* pDC)
{
 CHalloDoc* pDoc =
            GetDocument();
 ASSERT_VALID(pDoc);

 // TODO: add draw code ..
 CRect r;
 GetClientRect(r);
 pDC->SetTextAlign
  (TA_BASELINE|TA_CENTER);
 pDC->
   SetBkMode(TRANSPARENT);
 pDC->TextOut(r.Width()/2,
   r.Height()/2,
   pDoc->m_data,
   pDoc->
      m_data.GetLength());
}
```

Diese Ergänzung wird übrigens vom sogenannten *ClassWizard* unterstützt, da die Methode schon generiert war und nur editiert wird. Was jedoch ein Device Context ist und wie er benutzt wird, muß im Handbuch (z.B. online) zur Klasse *CDC* nachgelesen werden. Hier hört die visuelle Unterstützung auf.

Damit sind unsere Dokumente nicht mehr leer wie in Abbildung 1, sondern tragen alle in ihrer Mitte die Zeichenkette Hallo, so wie das erste Dokument in Abbildung 3.

Dialoge

So richtig visuell wird Visual C++ erst, wenn Ressourcen wie Dialogboxen erstellt werden. Weil das Hallo nur am Anfang nett ist, soll die Benutzerin den Text, der im Dokument dargestellt wird, durch einen Dialog ändern können. Der Dialog ist in Abbildung 4 dargestellt und der Effekt nach drei Dialogen zu jeweils einem Dokument in Abbildung 3.

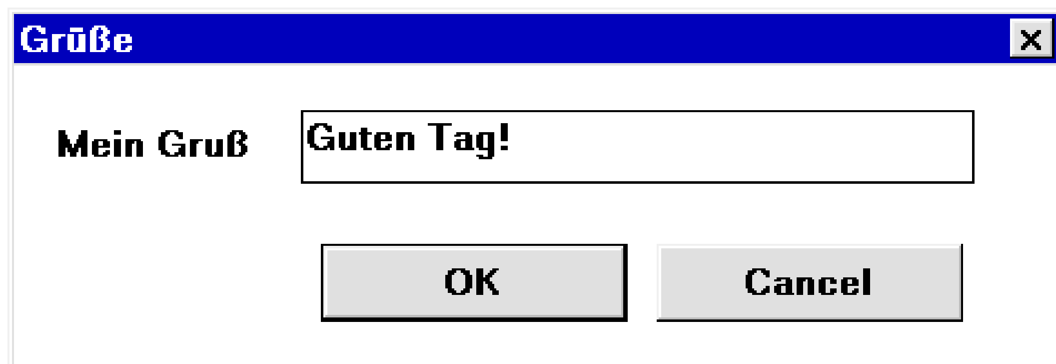

Abbildung 4: Dialog

Das Aussehen einer Dialogbox kann wirklich visuell, d.h. mit *Drag&Drop* erstellt werden und stellt dann eine weitere Ressource der Anwendung dar. Es kann im Aussehen später leicht verändert werden.

Die Dialogbox muß natürlich eingebunden werden, dazu wird in der Regel eine entsprechende Klasse, eine sogenannte Dialogklasse erzeugt, und dabei hilft der *ClassWizard*. Angegeben wird nur der Klassenname, in unserem Beispiel *GrussDialog*, die Dialog-ressource, die vorher erstellt worden ist, und die Verbindung von Eingabefeldern des Dialogs zu Elementobjekten der Klasse. Wir haben ein Elementobjekt *m_gruss* in der Klasse *GrussDialog* eingefügt, das an das Eingabefeld über dessen Namen gebunden wird. Da der *ClassWizard* bei der Anbindung alle existierenden Namen zeigt, werden hier zumindest keine Flüchtigkeitsfehler gemacht.

Was jetzt noch fehlt, ist der Aufruf des Dialogs und das Weiterverarbeiten des Resultats. In unserem Beispiel soll der Dialog durch einen Menüeintrag aufgerufen werden. Die Menüleiste ist auch eine Ressource, die mit dem Ressourcen-Editor ähnlich komfortabel wie die Dialogbox erstellt werden kann. Durch Anklicken des neu eingefügten Eintrags *Ändere Gruß* unter *Bearbeiten* soll die Methode *OnEditAenderegruss()* aufgerufen werden:

```
void CHalloDoc::
OnEditAenderegruss()
{
 // TODO: Add your ...
 GrussDialog d;
 d.m_gruss = m_data;
 if (d.DoModal()!=IDOK)
         return;
 m_data = d.m_gruss;
 UpdateAllViews(NULL);
}
```

Diese Methode erzeugt einen Grußdialog *d*, belegt den Gruß des Dialogfelds *m_gruss* mit dem Gruß des Dokuments *m_data* vor und startet den Dialog mit *DoModal()*. Diese Art des Dialogs kann nicht von der Anwendung unterbrochen werden, und die Methode *DoModal()* endet erst, wenn die Dialogbox geschlossen wird, z.B. durch Drücken von Ok. Über den Rückgabewert wird in unserem Beispiel angezeigt, ob der neue Gruß übernommen werden soll.

Da das Dokument verändert worden ist, werden alle Sichten dieses Dokuments mit *Update-AllViews()* aktualisiert.

Die Dokumente spielen u.a. eine ähnliche Rolle wie die Modelle nach dem Entwurfsmuster Model-View-Controller (siehe z.B. [2]), leider aber auch gleichzeitig die Rolle des Controllers.

Message-Maps

Die Anwendung ist jetzt vollständig. Denn der *ClassWizard* hat die sogenannte Message-Map der Klasse *CHalloDoc* um den Eintrag zum Aufruf der Methode *OnEditAenderegruss()* erweitert, wobei wir im Ressourcen-Editor die Verbindung zum Ereignis der Auswahl des Menüeintrags vorgenommen haben.

Die Message-Maps übernehmen die Abbildung von Ereignissen, die z.B. durch Mausklick verursacht werden, auf die Funktionen, die das Ereignis bearbeiten sollen. Jede Klasse, die von *CCmdTarget* abgeleitet ist (siehe Abbildung 2), hat eine Message-Map.

Eine Windows-Anwendung, die von *CWinApp* abgeleitet ist, hat ein Hauptprogramm, das nach der Initialisierung die Ereignisschleife (*event loop*) ausführt, so wie auch alle anderen normalen fensterbasierten Anwendungen. Bei der Bearbeitung eines Ereignisses wird zunächst der Empfänger ermittelt und dann ein entsprechender Eintrag in dessen Message-Map gesucht und die gefundene Funktion ausgeführt, nicht ohne vorher zu übergebende Daten zu hinterlegen.

Da in den Message-Maps quasi Methodenadressen notiert sind, wird dadurch die C++-spezifische dynamische Bindung von virtuellen Methoden außer Kraft gesetzt. Da das

Überschreiben von Methoden aber eine notwendige Voraussetzung für eine vereinfachte Programmierung von Fensteroberflächen ist, und auch die MFC benutzt es extensiv, wird ähnlich dem Suchen einer Methode in der Tabelle der virtuellen Methoden einer Klasse und deren Basisklassen in den Message-Maps einer Klasse und ihrer Basisklassen nach einem passenden Eintrag gesucht.

In der Beschreibung ist zu lesen, daß dabei wohl nicht strikt nach der Vererbungshierarchie vorgegangen wird. Vor Überraschungen sind wir nicht sicher.

Angenehm ist, daß die Erzeugung und Aktualisierung der Message-Maps vom *ClassWizard* sehr gut unterstützt werden.

Einfache Personalverwaltung

Als letztes Beispiel soll eine einfache Personalverwaltung untersucht werden, die nach dem Entwurfsmuster Model-View-Controller entworfen und schon mal unter Benutzung von Tcl/Tk implementiert wurde. Die Idee ist, daß nur die Teile View und Controller neu implementiert werden müssen. Fast richtig.

Da Visual C++ das Programmgerüst vorgibt, muß erst ein Projekt mit einer Anwendung erzeugt werden, wie z.B. oben die leere Anwendung kreiert wurde. Dann muß das Modell, das Personal, an ein Dokument gebunden werden wie z.B. *m_data* an das Hallo-Dokument. Als View wird die erzeugte *CPersView* mit einem Controller verbunden. Das Gerüst des Controllers wird als Dialog mit dem Ressourcen-Editor und dem *ClassWizard* erzeugt. Das Optische ist in Abbildung 5 anzuschauen. Die Anbindung der eingetragenen Werte an Elementobjekte wird mit dem *ClassWizard* vorbereitet, so daß jeweils nur ein *UpdateData()* aufgerufen werden muß, um die Werte auszutauschen.

Abbildung 5: Personalverwaltung

Wenn Sie sich damit abgefunden haben, daß sowohl Dokument als auch View als auch Dialog jeweils Teile des Controllers übernehmen, geht die Produktion der Anwendung recht schnell.

Leider fehlt in den generierten Definitionsdateien die Klammerung durch eine *#ifdef*-Anweisung, damit sie nur einmal vom Präprozessor als Text geladen werden. Das ist bei nicht leeren Anwendungen fast unabdingbar und per Hand nachzutragen.

Im konkreten Beispiel hatte die alte Anwendung die C++-Standard-Bibliothek benutzt, u.a. Container-Templates wie *list* und *map*. In den MFC werden jedoch Klassen wie *CMap* und *CList* verwendet, und die Methoden sehen natürlich auch leicht anders aus. Des weiteren wird nicht der primitive Datentyp *bool* benutzt, sondern mit der in C üblichen

Hilfskonstruktion *typedef int BOOL* gearbeitet.

Wie bereits erwähnt, war die Reihenfolge der Konstruktoraufrufe gewöhnungsbedürftig, wobei der Debugger zur Klärung konsultiert wurde. Initialisierungen, z.B. von Assoziationen zwischen Objekten, die durch Zeiger realisiert sind, mußten in die entsprechenden Initialisierungsmethoden verlagert werden.

Bei der Erstellung der Ressourcen für die grafische Benutzeroberfläche werden im Prinzip alle Elemente unterstützt, die aus Microsoft-Anwendungen bekannt sind, einschließlich Standarddialogboxen zur Dateiauswahl und Anbindungen an ODBC und OLE.

Da der Kern der Anwendung klassisch in C++ programmiert werden muß bzw. darf, wird er auch von keinem *Wizard* unterstützt. Es ist also trotz Visual ... notwendig, C++ zu beherrschen, allein schon, um die entsprechenden Zeilen im generierten Code ergänzen zu können. Die Teilmenge C reicht keinesfalls aus.

Zusammenfassung

- Der *AppWizard* ist ein gutes Werkzeug, das ein Programmgerüst nach einer der Standardarchitekturen erzeugt.
- Der *ClassWizard* ist für die Generierung von Klassen, die in das Gerüst passen, ein sehr mächtiges Werkzeug.
- Der erzeugte Code ist gut lesbar. Er nutzt viele Konzepte der Programmiersprache C++ wie Vererbung und virtuelle Funktionen.
- Ressourcen wie Dialoge, Ikonen, Symbolleisten und Menüs können wirklich visuell mit dem Ressourcen-Editor erstellt werden. Die Anbindung an das Programm wird durch den *ClassWizard* relativ einfach.
- Wenn Ihre Anwendung einem der Gerüste entspricht, wird deren Erstellung insgesamt relativ einfach.
- Die Trennung der Kontrollelemente von einer Sicht ist nach dem Gerüst kaum machbar.
- Die Message-Maps sind eine etwas eigenwillige und sicher historisch gewachsene Einrichtung. Irgend eine Art von Message-Maps braucht jedes Fenstersystem, aber es sollte konform mit dem Überschreiben von virtuellen Methoden gehen.
- Wird die Anwendung per Hand erzeugt, erscheint mir die Benutzung der vielen Elemente aus der MFC schwierig, da dann extreme Detailkenntnisse erforderlich sind.
- Abweichungen von einem Gerüst müssen mit erheblichem zusätzlichen Einarbeitungsaufwand bezahlt werden, da dann früher oder später Teile konventionell mit Aufrufen aus der Microsoft Win32 API oder der MFC programmiert werden müssen.
- Meiner Meinung nach ist es eine Entwicklungsumgebung für “Einzelprogrammierer”, denn Zugriffsberechtigungsschutz kann es aufgrund von Windows nicht geben, und eine Versionskontrolle fehlt auch.
- Die Programmiersprache C++ muß beherrscht werden, um Visual C++ sinnvoll nutzen zu können.

Insgesamt können Sie Visual C++ gut anwenden, wenn Ihre Programme nur unter Systemen von Microsoft laufen sollen. Plattformneutralität ist ein Fremdwort.

Literatur

[1] F. Heimann, G. Krüger, N. Turianskyj *Einführung in Visual C++ 4.x* Addison-Wesley, Bonn, 1996.

[2] F. Buschmann, H. Rohnert, P. Sommerlad, M. Stal *Pattern-Oriented Software Architecture — A System of Patterns* Wiley and Sons Ltd., 1996.