Document-View mit dem Borland C++ Builder (Version: 2004.03.11)

Prolog

Ich möchte dieses Tutorial auf keinen Fall unkommentiert wissen. Wer meine jQT kennt, der Weiss, dass ich immer ein Vorwort schreibe, welches den Rahmen des Tutorials festlegt. Im Rahmen dieses Prologs möchte ich erklären, was dieses Tutorial ist und – ganz wichtig – was es nicht ist.

Das Tutorial ist die Niederschrift meines Wissens sowie einiger Gedanken und Problemlösungen rund um Dokument/View. Es enthält Ideen, Theorien und Lösungen welche mir als interessant und lesenswert erscheinen. Diese Ideen und Theorien sind keine absolute Lösung und schon gar nicht sind die ganzen Konstruktionen hier ausgegoren und 1:1 auf jedes Problem applizierbar. Vielmehr soll dieses Tutorial den Grundgedanken von Doc/View vermitteln und, damit einhergehend, die Gedankenwelt der Programmierer etwas anregen. Es ist also quasi als „kreativer Input" zu verstehen.

Ja, ich konstruiere im Rahmen dieses Tutorials ein Document/View-Framework. Man sollte sich allerdings bewusst sein, dass dieses Framework lediglich zur Veranschaulichung der hier beschriebenen Methoden dient. Es hat massenhaft schwächen und ist unausgereift. Es ist also höchstens als Basis für eine **Neuentwicklung des Frameworks** kaum aber als Basis für die Entwicklung einer Applikation geeignet. (und wehe ich erwische einen der dieses Framework 1:1 für seine Applikation vergewaltigt (o;) Das sollte sich jeder hier bewusst werden.

Meine gute Güte. Ich schreib hier den Prolog und schon ist die halbe Seite voll. Ich hätte doch noch soviel zu sagen… Naja, wer sich für den Rest interessiert, soll den Epilog lesen (o:

Die anderen (undankbares Pack (o;) möchte ich nicht liänger vom eigentlichen Inhalt abhalten. In diesem Sinne lasst uns mit dem eigentlichen Thema beginnnen…

-junix

Wieso Document-View?

Von Daten und deren Anzeige

Wem ist das nicht schon passiert? Eine Anwendung ist eigentlich fertig, und schon kommt noch diese kleine Änderung, jener Dialog, dieses zusätzliche Anzeigeelement und vielleicht will der Chef plötzlich noch eine Anzeige für eine Sollgeschwindigkeit lieber doch auch noch zusätzlich in m/s statt nur in km/h.

Dabei sollten beide Felder natürlich editierbar sein, denn man sollte das soll ja jederzeit ändern können. Und am liebsten so, dass man gar nichts überlegen, geschweige denn umrechnen muss. Natürlich ist das kein Problem, das kann man ganz einfach einbauen. Ändert sich der Wert im einen Feld, so fängt man das Ereignis ab und passt die Daten im anderen Feld entsprechend an und umgekehrt. – Alle sind zufrieden.

Im Verlauf der Beta-Phase in der Anwender auf die Applikation losgelassen werden, passierts: Da kommt ein vielfacher Wunsch, dieses und jenes Anzeige-Element auf einen extra-Dialog auszulagern, den man – unabhängig von der Hauptanwendung – so platzieren kann, dass man immer den Überblick über die wichtigsten Werte hat. Der Wunsch ist da, der Auftrag erteilt. Und damit fängt der Spass erst richtig an.

Ist dezentral bedeutet nicht zwingend "besser"

Gerade im C++ Builder neigt man als Entwickler dazu, die Daten gleich in den zugehörigen Dialogen (Formularen) zu belassen und bei Bedarf durch Zugriff auf das passende Formular die Daten "aufzutreiben". Zwar trägt in vielen Dingen eine dezentrale Struktur zur Ausfallsicherheit bei, ier schiessen wir uns allerdings selbst ins Knie:

Wenn wir das obige Szenario genauer betrachten, so bedeutet dies konkret, dass wir gezwungen sind, im zusätzlichen Dialog zunächst mal alle Dialoge bekannt zu machen, in welchen die gewünschten Daten vorhanden sind, zum Anderen die Daten aus den entsprechenden Dialogen so aufzubereiten, dass sie unserer Darstellungsart, wie wir sie gerne hätten entsprechen.
Es müssen also folgende Punkte bekannt sein:

- Jedes Formular, welches Daten liefert die wir benötigen
- Das Objekt innerhalb des Dialogs, welches die Daten enthält (den "Datenlieferanten")
- Das Datenformat des jeweiligen Datenlieferanten (war die Geschwindigkeit denn jetzt in m/s oder km/h?)

Noch verheerender wird es, wenn es darum geht, bestimmte Einstellungen und Werte die in verschiedenen Formularen vorkommen zu speichern. Wie realisiert man das? In welchem Dialog/in welcher Klasse implementiert man nun die „Speicherfunktion"? Im Hauptdialog, weil da sowieso schon ein Haufen globaler „Müll" abgelagert ist? Oder vielleicht lieber in einem Nebendialog?
Klar. In einem kleinen Projekt, stellt das alles kein Problem dar. Oder etwa doch? Nehmen wir obiges Fallbeispiel mal etwas genauer unter die Lupe.

Beziehungskisten I

Machen wir zunächst eine Bestandesaufnahme: Zunächst hatten wir auf dem Hauptformular (MainForm) ein Objekt welches eine Sollgeschwindigkeit in km/h anzuzeigen hatte (Speed_TEp). Anschliessend auf dem Selben Formular noch ein Anzeigeelement das die Sollgeschwindigkeit in m/s aufzeigen sollte. Nennen wir dieses Element „Speedms_TLp". Dazu kommt ein Dialog, der ebenfalls die Sollgeschwindigkeit anzeigen soll (ControlForm), auf dem ein weiteres Edit-Control sitzt (TargetSpeed_TEp). Zeichnen wir nun die Beziehungen die sich durch die Notwendigkeit des gegenseitigen Updates ergeben ein, könnte das etwa so aussehen wie in Abbildung 1.

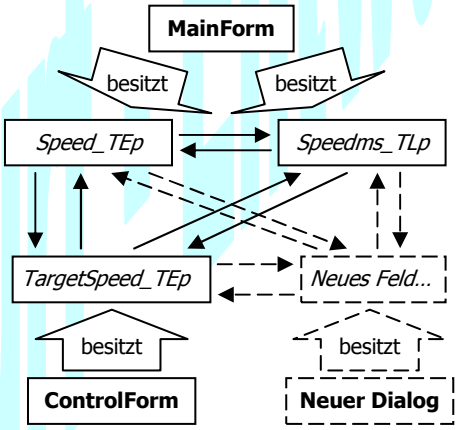


Abb. 1: Beim Hinzufügen eines elements, nehmen .

Was hier direkt ins Auge sticht: Ändert sich an einem Control z.B. die Einheit (z.B. statt m/s in m/min), ist es nötig, an 3 Stellen eine Baustelle zu öffnen. (Überall muss die Konvertierung geändert werden.) Ein unhaltbarer Zustand, zumal sich mit jedem weiteren Element oder Dialog die Zahl der Verbindungen vermehrt. Ausserdem sind die Objekte (hier ControlFrom und MainForm) nun - entgegen jeglichen OO-Ansatzes – nichtmehr selbstständige, in sich geschlossene Objekte sind. Es ist also nicht möglich, ControlForm oder MainForm ohne weitere Modifikationen in einem neuen Projekt zu verwenden.

Was ist Document-View?

Zentralistisch ist doch nicht immer schlecht…

Was in der Einleitung erklärt wurde, ist allerdings kein Grund entnervt aufzugeben und den Softwareentwickler-Job an den Nagel zu hängen. Natürlich bin ich nicht der Erste der über solche Probleme gestolpert ist, und entsprechend gibt es auch Lösungskonzepte. In dem von mir skizzierten Fall drängt sich eine Lösung auf: Das *Document-View-Konzept.* In einem Satz ausgedrückt, bedeutet Document/View die Trennung von Datenspeicher- und Anzeigeobjekten. Dieses Konzept geht von einer zentralen Klasse (Document) aus, welche die Daten verwaltet. Alle Dialoge, welche Daten aus dem Dokument anzeigen möchten (Views), müssen sich bei diesem Dokument „registrieren", damit die Views auf Änderungen im Dokument reagieren können, bzw. einen Update der Anzeige veranlassen können.
Die zentralistische Datenspeicherung hat aber abgesehen von der flexibleren Architektur noch weitere Vorteile. Das Laden und Speichern von Daten wird plötzlich erheblich vereinfacht. Idealerweise stellt nämlich das Dokument Methoden zum Speicher und Laden von Daten zur Verfügung. (Diese müssen jeweils selbstverständlich im jeweiligen Dokument ausprogrammiert werden.)

Beziehungskisten II

Durch die zentralistische Datenhaltung verringert sich die Anzahl der Beziehungen zwischen den Objekten drastisch. Zumindest aus Sicht der Views. Anstatt, dass die Views untereinander den Update auslösen, werden geänderte Werte in das Dokument geschrieben. Das Dokument seinerseits sorgt dann dafür, dass jeder der registrierten Views sich wieder den aktuellen Wert aus dem Dokument zieht.

Auch das Erweitern des Systems um weitere Dialoge vermehrt die Zahl der Beziehungen lediglich um 2, von denen lediglich der neue Dialog und – minimal – das Dokument betroffen sind. (Siehe Abb. 2)

Dieses veränderte Konzept erfordert natürlich auch eine veränderte Denkensweise. Im Gegensatz zu vorher, wo wir die Defaultwerte in den Objekten eingetragen haben, werden Defaultwerte hier ins Dokument geladen und dadurch ein Update der Views erzwungen.

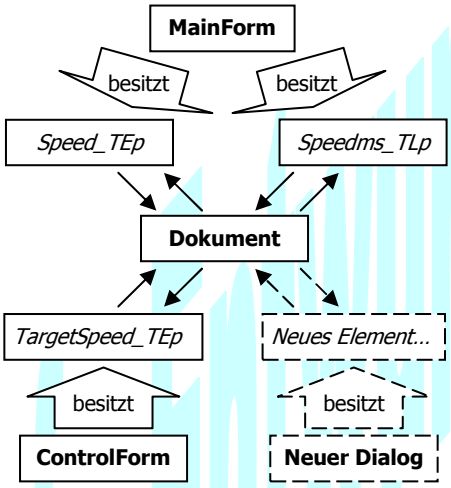


Abb. 2: Beim Hinzufügen eines Elements, nimmt die Zahl der Abhängigkeiten linear zu.

Und wie wird das nun implementiert?

Schon Lust bekommen, dieses Konzept in die eigene Anwendung einzubauen? Bestimmt stellt sich jetzt eine Frage: „Wie kann ich diese Theorie nun in die Praxis umsetzen?". Der übliche Weg führt über 2 Basisklassen im Framework, welche ein, zwei Funktionen die benötigt werden implementieren. Zu Einen wäre dies TDocument, zum Andern TView.

TDocument ist dabei der grössere Brocken. In TDocument sind für gewöhnlich Funktionen implementiert, die zumindest das Anhängen und Lösen von Views an/vom Dokument ermöglichen. Häufig sind aber auch noch spezielle Funktionen bereits abstrakt bereitgestellt, welche es ermöglichen, die Daten aus dem Dokument zu speichern, bzw. in das Dokument zu laden, Funktionen welche es ermöglichen das Dokument in seinen Ausganszustand zurückzusetzen, etc.

Im Vergleich zu TDocument ist TView eigentlich richtig primitiv. Alles was TView mitbringen muss, ist eine definierte aber abstrakte Funktion, welche das Erzwingen eines Updates der Daten auf dem Dialog ermöglicht. Während TDocument allerdings eine eigenständige Klasse ist, muss TView abgeleitet werden. Der Mechanismus ist der, dass wir eine Zwischenschicht zwischen (aus dem Framework abgeleiteten) Dialogfeld und Framework schieben, die uns die passende Funktion zur Verfügung stellt.

Ein Document/View-Framework

Die View-Klasse kurz erklärt

Das Problem das wir haben ist folgendes: Die Dokumentenklasse benötigt eine Funktion, welche sie aufrufen kann um einen Update der Daten zu erzwingen. Da das VCL Framework keine Events bietet, die man dafür „vergewaltigen" könnte, ohne unangenehme Nebeneffekte zu erzielen, müssen wir diese Funktionalität nachträglich einbauen.

Trotzdem ist das Erstellen der View-Klasse der eigentlich einfachste Teil des Frameworks. Immerhin ist es nur nötig eine weitere (rein) virtuelle Funktion einzufügen. Implementiert wird diese Funktion jeweils erst Dialog spezifisch. Da es unumgänglich ist, dass sich die View-Klasse in das VCL-Framework einfügt, müssen wir das View von einer Klasse ableiten, welche für möglichst viele Klassen, welche als Anzeige dienen könnten, als Basisklasse dient. Da wir in der Regel keine Chance haben, die Frameworkklassen zu verändern, ist es nötigt, das View als Erbe von TForm anzusiedeln. Das ermöglicht es uns, dass wir Views relativ einfach erstellen können, in dem wir einfach die Vorfahrenklasse der mit dem Builder erstellten Formular-Klasse von TForm auf z.B. TView ändern.

Die View-Klasse als Deklarationsbeispiel
Die Klasse TView könnte also in Etwa so aussehen:

```
class TView : public TForm
{
public:
    /* Konstruktion */
    __fastcall TView(TComponent* Owner);

    /* Update/Renderfunktion */
    virtual void __fastcall UpdateData(TDocument *Sender_TDp) = 0;
};
```

Dabei entspricht der Konstruktor von TView exakt dem Konstruktor von TForm um Kompatibilitätsprobleme zu vermeiden. Ausserdem wird eine rein virtuelle Methode UpdateData deklariert, die später in der Ableitung aus TView implementiert wird und für das Rendern der Daten aus dem Dokument verantwortlich ist.

Die Dokumenten-Klasse erklärt
Die Dokumentenklasse wird ein Wenig komplexer als die View-Klasse. Da wir diese Klasse nirgends in das Framework der VCL einzufügen brauchen, können wir hier eine eigenständige Basisklasse erstellen. Diese Klasse stellt das Interface für das spätere Dokument bereit. Hierzu gehören im Wesentlichen Funktionen zum

- Speicher und Laden von Daten innerhalb des Dokuments
- Laden von Default-Werten um das Dokument in den Grundzustand zu versetzen

Bei diesen Funktionalitäten macht es wenig sinn, diese in der Basisklasse zu implementieren. Immerhin kennt erst die Spezialisierung dieser Klasse, also der Erbe, die echte Funktionalität die hier geboten werden muss. Es handelt sich daher um rein virtuelle Funktionen.

Funktionen die bereits in der Basisklasse implementiert werden, sind Funktionen welche es erlauben,

- Views an ein Dokument zu binden und die Bindung zu lösen
- Einen Update aller registrierter Views zu erzwingen
- Einen Update-Event vorübergehend zu blockieren um grosse Datenmengen zu aktualisieren.

Im Gegensatz zum obigen rein virtuellen Interface, können und müssen diese Funktionen bereits in der Basisklasse vorhanden sein. Immerhin ist es die Kernfunktionalität des Dokuments.

Die Dokumenten-Klasse als Deklarationsbeispiel
Unsere TDocument-Klasse hätte demnach in Etwa folgendes Outfit:

```
class TDocument
{
public:
    /* Kon-/Destruktoren */
    __fastcall TDocument();
    virtual __fastcall ~TDocument();

    /* Servicefunktionen für die View Registration. */
    void __fastcall RegisterView(TView *View_TVp);
    void __fastcall UnregisterView(TView * View_TVp);

    /* Rein virtuelle Funktionen */
    virtual bool __fastcall LoadDocument(AnsiString Filename_AS) = 0;
    virtual void __fastcall ResetDocument(void); //Laden von Default werten
    virtual bool __fastcall SaveDocument(AnsiString Filename_AS);

    /* Funktionen zum vorübergehenden Blockieren des Updates */
    void __fastcall BeginUpdate(void);
    void __fastcall EndUpdate(void);
```

```
protected:
    void __fastcall UpdateAllViews(void);

private: // Get-/Set-Methoden
    void __fastcall   SetBeginUpdateCounter(unsigned int value);
    unsigned int      __fastcall GetBeginUpdateCounter();

protected:
    __property unsigned int BeginUpdateCounter  =
{ read=GetBeginUpdateCounter,

write=SetBeginUpdateCounter };

private:
    unsigned int      FBeginUpdateCounter_uint; // Feld für BeginUpdate
                                                // Counter
                                                // Property.
    TCriticalSection *FUpdateCounterSync_TCSp;  // Sync.obj. für Update-
                                                // Counter
    TCriticalSection *FViewListSync_TCSp;       // Sync.obj. für die View-
                                                // Liste
    TList            *FViews_TLp;               // Liste für die reg. Views
};
```

Die Klasse besitzt ein, drei Spezialitäten, die das Leben erheblich erleichtern:

- **Die Basisklasse wurde bereits Threadsicher gehalten.** Dies bedeutet, dass bereits kritische Teile des Codes durch Synchronisationsobjekte wie TCriticalSection geschützt werden. Abkömmlinge sind allerdings für die Threadsicherheit ihrer Daten selbst verantwortlich.
- **Die Ausführung von UpdateAllViews() kann verhindert werden** wenn zum Beispiel mehrere Attribute der abgeleiteten Klasse geändert werden, indem BeginUpdate() vor dem Aktualisieren der Werte bzw. EndUpdate() nach dem Aktualisieren aufgerufen wird. Dadurch wird die Zeit, die die Views zum Updaten benötigen verkürzt.
- **BeginUpdate()/EndUpdate() sind verschachtelungsfähig.** Dadurch, dass in Begin-/EndUpdate nicht mit einem einfachen Flag sondern mittels einer Zählvariable gearbeitet wird, ist es möglich, dass an verschiedenen Stellen im Code gleichzeitig BeginUpdate() aufgerufen werden kann, ohne dass der Aufruf von EndUpdate() den Ablauf durcheinander bringt.

Eine Beispielanwendung

Wie ist das Framework anzuwenden?

Na bisher klingt das Ganze ja einigermassen verlockend. Doch wie benutze ich das Framework nun im Builder? Bauen wir Schritt für Schritt eine Anwendung mit dem Framework auf und schauen, wie wir das Framework einsetzen.

Als Beispiel bauen wir mit dem Builder einen kleinen Texteditor mit einem eigenständigen Fenster, welches ein-zwei Informationen über das in Bearbeitung befindliche Dokument anzeigt. (Für den Anfang mal einfach die Anzahl Zeilen im Editor, später dann auch noch die Anzahl Wörter.

Womit fangen wir an?

Am Besten, wir implementieren zunächst einen rudimentären Editor. Ein Form, ein Memo auf das Form packen, einen Öffnen- und einen Speicherndialog dazu, eine Menüleiste, in der wir all diese Funktionen zugreifbar machen. Ich denke nicht, dass ich dazu noch was sagen muss. Falls doch, so ist es empfohlen, unter http://bcb-tutorial.c-plusplus.de den Einstieg in den C++ Builder zu suchen.

Um nun aber das Document/View-Framework von oben zu benutzen, ist es nötig, dass wir die Dateien document.cpp/h und view.cpp/h dem Projekt hinzufügen. Dies geschieht über das „Projekt"-Menü. Ein wesentlich eleganterer Weg, diese Units verfügbar zu machen ist allerdings das Vorcompilieren und

entsprechend abzulegen. Wie man Units vorkompiliert zur Verfügung stellt, werde ich in einem anderen Tutorial ausführlich erklären. (Es lohnt sich also, die Tutorial-Seite http://www.junix.ch/bcb/help häufiger zu frequentieren. (o;)

Für unser Beispiel allerdings, kopieren wir – der Einfachheit halber – einfach die beiden Units in das Projekt-Verzeichnis und fügen sie dem Projekt hinzu. Der Unterschied ist minimal, trägt einfach den Nachteil mit sich, dass wir jedem Projekt diese Dateien hinzufügen müssen, wenn wir Document/View einsetzen wollen.

Ein kleiner Betrug am Rande
Nun haben wir zwar einen Editor, aber noch kein Document-View System dahinter. Dem Einen oder Anderen mögen im Moment auch Zweifel aufkommen, dass das Sinn machen wird, und den Zweiflern muss ich Recht geben – wenn auch nur vorläufig. Um das Zerstreuen des Zweifels kümmere ich mich zu einem späteren Zeitpunkt. Für den Moment gehen wir einfach davon aus, dass es sinn macht.

Wie gesagt, haben wir keine Document/View- Architektur in unserem Editor. Unser Hauptdialog ist nach wie Vor vom Typ TForm. Dadurch haben ist die von uns in die Klasse TView eingefügte „Renderfunktion" (UpdateData) ebenfalls nicht verfügbar, und es ist nicht möglich, Update-Ereignisse des Dokuments zu empfangen. Um dieses Manko zunächst zu überbrücken, müssen wir einen kleinen Eingriff am Code den der C++ Builder generiert hat vornehmen:
Neu erbt unser Hauptdialog nicht mehr von TForm sondern von TView. Diese Modifikation schränkt uns in keiner Weise ein, denn wir haben ja TView seinerseits wieder von TForm abgeleitet. Wir schieben also, wie im vorherigen Teil beschrieben einfach eine Schicht zwischen unserem Hauptdialog und TForm. (das Includen der View-Unit nicht vergessen! (o:)

```
class TMain_TFp : public TView
{
__published:    // IDE-managed Components
...
```

Für das VCL-Framework sieht es somit aus, als hätten wir noch immer ein TForm, für unser auf die VCL aufgesetztes Framework allerdings, besteht nun die Möglichkeit, dass Document und View miteinander interagieren.

Ohne Document, kein View
Die Tatsache, dass nun ein View in der Anwendung sitzt, hilft uns noch nicht wirklich weiter, denn es fehlt ja noch das Gegenstück, das Document, welches überhaupt erst das View dazu bringt, seine Aufgabe zu übernehmen und die Daten – in welcher Form auch immer – darzustellen. Als Nächstes müssen wir also ein Dokument implementieren.
Dazu leiten wir von unserer Basisklasse TDocument unsere eigene Klasse (TMyDocument) ab.

Was gehört denn nun alles ins Dokument? Primär soll das Dokument ja Daten zur Verfügung stellen. Diese implementieren wir mittels Eigenschaften welche als Public deklariert sind. In unserem Fall wäre das wohl primär der Text den wir speichern möchten. (Im Beispiel die Eigenschaft „Content")

```
class TMyDocument : public TDocument
{
/* Methoden */
public:
    __fastcall TMyDocument();

    virtual bool inline __fastcall LoadDocument(AnsiString Filename_AS);
    virtual void __fastcall ResetDocument(void);
    virtual bool __fastcall SaveDocument(AnsiString Filename_AS);

private:
```

```
    void __fastcall SetContent(AnsiString value);
    AnsiString __sfastcall GetContent();

/* Attribute */
public:
    __property AnsiString Content  = { read=GetContent,
                                       write=SetContent };
private:
    AnsiString FContent_AS;
 };
```

Da wir ja auch einen Speichern- und einen Öffnendialog haben, ist es nötig, diese beiden Methoden ebenfalls noch zu implementieren. Es macht sinn, die Speicher-Methode erst in der konkreten Klasse zu implementieren, denn hier sind ja die Datenformate bekannt. Das Selbe gilt selbstverständlich auch fürs Öffnen der Dateien. Die Methode „ResetDocument" ist ebenfalls noch zu implementieren. Damit laden wir in das Dokument einen Standard-Text. Diese Methode sollte unbedingt auch im Konstruktor des Dokuments aufgerufen werden, damit definierte Werte im Dokument vorhanden sind, bevor wir das erste View registrieren.

```
void inline __fastcall TMyDocument::ResetDocument(void)
{
    Content = "";
}
```

Die Methode „ResetContent" bringt das Dokument in einen definierten Zustand.

Nachdem wir nun ResetDocument, die Eigenschaft Content und das Speichern bzw. Öffnen implementiert haben, ist unser Dokument mal vorläufig fertig. Was wir nun noch brauchen ist eine global bekannte Instanz von unserem Dokument.

```
//---------------------------------------------
TMyDocument *Document_TMDp = NULL;
//---------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    Document_TMDp = new TMyDocument();

...

    delete Document_TMDp;Document_TMDp = NULL;

    return 0;
}
```

„Nach der Zerstörung eines globalen Objektes muss der Zeiger der globalen zwingend NULL gesetzt werden."

Diese Instanz erzeugen wir als globales Objekt. Auf dieses Objekt greifen dann alle Views zu. Wie für andere globale Variablen ist es auch hier unerlässlich, mit der Verwendung sorgfältig umzugehen. Dies bedeutet:

- Jeder Erzeugung oder Zerstörung muss peinlichst genau überprüft werden.
- Nach der Zerstörung eines globalen Objektes muss der Zeiger der globalen zwingend NULL gesetzt werden. (Was man eigentlich allgemein auf Zeiger anwenden sollte)
- Vor jedem Zugriff muss geprüft werden, ob das Objekt überhaupt noch vorhanden ist (Prüfung des Zeigers auf NULL)

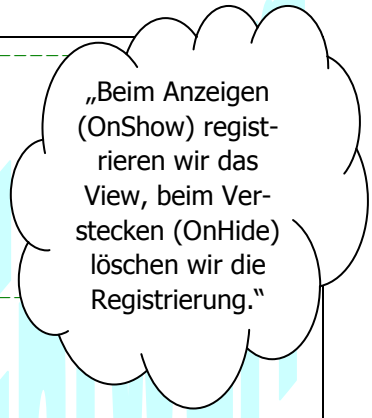
Die Verbindung vom Document zum View

Was uns nun noch fehlt, ist die Verbindung zwischen Document und View. Zu diesem Zweck verwenden wir die Ereignisse OnShow bzw. OnHide von TForm. Beim Anzeigen (OnShow) registrieren wir das Formular als View, wird das Formular vom Bildschirm verbannt (OnHide) entfernen wir das Formular wieder aus der Liste der zu aktualisierenden Views. Das hat den Vorteil, dass wir nur Rechenzeit für das Rendern der Daten aufbringen, welche auch wirklich angezeigt werden. Je kleiner die Liste der

Views und deren Render-Routine, desto schneller wird das ganze System. (Genaueres dazu erkläre ich noch im Rahmen des fünften Abschnitts, indem ich noch einige allgemeine Verwendungshinweise aufziege.)

```
//---------------------------------------------------------------------
void __fastcall TMain_TFp::FormShow(TObject *Sender)
{
    /* Sollte das Dokument nicht existieren,
       bringt diese Aktion keinen Sinn. */
    if (Document_TMDp != NULL)
        Document_TMDp->RegisterView(this);
}
//---------------------------------------------------------------------

void __fastcall TMain_TFp::FormHide(TObject *Sender)
{
    /* Sollte das Dokument nicht existieren,
       bringt diese Aktion keinen Sinn. */
    if (Document_TMDp != NULL)
        Document_TMDp->UnregisterView(this);
}
//---------------------------------------------------------------------
```



Soweit die Theorie. Wie machen wir das nun? Wie schon erwähnt, haben wir ja das Dokument als global bekannte Instanz von unserem Dokument implementiert. Um nun die Verbindung zwischen Dokument und View zu schaffen, rufen wir die Methode RegisterView() bzw. UnregisterView() von unserem Dokumenten-Objekt auf. Als Zeiger übergeben wir jeweils das zu registrierende Formular (in der Regel „this").

Diese Verbindung ermöglicht es nun, dass wir über Änderungen im Dokument informiert werden.

Daten rendern
Zwar haben wir jetzt ein View, ein Dokument und wir haben die beiden sogar verbunden. Das Alles bringt die Applikation aber noch nicht zum Laufen. Was fehlt?
Bisher laufen unsere Update-Benachrichtigungen einfachs ins Leere. Genau so wie der Text im Dokument nirgendwo gesichert wird. – Nicht wirklich das was wir uns vorgestellt hatten. Trotzdem ist das nahe liegend, dass es sich so verhält. Haben wir doch noch ein Kernstück völlig vergessen: Die Renderfunktion.

Irgendwo einleitend haben wir gesagt, das Dokument ruft bei jeder Änderung der Daten „UpdateData" aller registrierten Views auf. Der Schluss liegt also nahe, dass wir in unserem Fall noch die Funktion UpdateData „mit Leben füllen" müssen.
Grundsätzlich ist nichts weiter zu tun, als aus dem Aufrufenden Dokument die gewünschten Daten zu extrahieren.

```
void __fastcall TMain_TFp::UpdateData(TDocument *Sender_TDp)
{
    TMyDocument *Doc_TMDp = dynamic_cast<TMyDocument *>(Sender_TDp);

    if (Doc_TMDp != NULL)
        Content_TMp->Text = Doc_TMDp->Content;
}
```

In obigem Listing verwenden wir den mitgelieferten Zeiger „Sender_TDp", um an die Daten zu kommen. Die oben verwendete Variante mit dynamic_cast hat – speziell auch bei mehreren Dokumenten – den Vorteil, dass es uns möglich ist, selektiv die Daten zu rendern, ohne das komplette View aktua-

lisieren zu müssen. In unserem Fall jetzt ist die Renderfunktion relativ unspektakulär. Sie liest einfach den Textstring aus dem Dokument und trägt ihn ins Memo auf dem View ein.

Interessanter wäre hier die Renderfunktion unseres zweiten Dialogs. Ich will hier mal in die Zukunft vorgreifen:

```
//---------------------------------------------------------------------------
void __fastcall TDocData_TFp::UpdateData(TDocument * Sender_TDp)
{
    /* Die Dokumentendaten aufgrund der neuen Daten anpassen */
    TMyDocument *Doc_TMDp = dynamic_cast<TMyDocument *>(Sender_TDp);

    if (Doc_TMDp != NULL)
    {
        CharCount_TLp->Caption = CharCount_TLp->Caption.sprintf(
                                            Lbl_CharCount_cAS.c_str(),
                                            Doc_TMDp->Content.Length());

        WordCount_TLp->Caption = WordCount_TLp->Caption.sprintf(
                                            Lbl_WordCount_cAS.c_str(),
                                            CountWords(Doc_TMDp->Content));
    }
}
```

Diese Renderfunktion ist ein wenig Komplexer, denn immerhin müssen hier zwei Werte errechnet und dargestellt werden. An dieser Stelle ein kleiner Tip:

Je komplexer eine Renderfunktion ist, desto länger dauert deren Ausführung und desto länger dauert das aktualisieren eines Werts. Dies wiederum führt – mit der Zeit – zu einem harzigen reagieren der ganzen Anwendung. Es gilt hier also klar die Devise, möglichst kurze und einfache Berechnungen anzustellen. Eigentlich verstösst hier das Beispiel bereits gegen diese Regel. Aber aus Gründen der Einfachheit, wollen wir darüber einfach mal grosszügig hinwegsehen.

Wie kommen die Daten ins Dokument?
Schön. Nun haben wir also endlich eine Möglichkeit, den Inhalt des Dokuments darzustellen. Aber wie kommen denn nun die Änderungen auf dem View ins Dokument? Zunächst einmal müssen wir innerhalb vom view auf die Änderungen reagieren. Teil der Reaktion sollte dann das Aktualisieren des Dokumenteninhalts sein. In unserem Fall macht es wohl sinn, nach jedem Zeichen – also bei jeglicher Änderung im Memo – das Dokument zu aktualisieren. In anderen Fällen mag es sein, dass es geschickter ist, die Änderung erst nach einem Button-Klick oder einem Druck auf eine spezielle Taste an das Dokument zu übergeben. In meinem Beispiel fand ich es angemessen, bei jeglicher Änderung das Dokument zu aktualisieren. Hinter der Aktualisierung steckt auch nicht wirkliche Magie:

```
void __fastcall TMain_TFp::Content_TMpChange(TObject *Sender)
{
    /* Sollte das Dokument nicht existieren,
       bringt diese Aktion keinen Sinn. */
    if (Document_TMDp != NULL)
        Document_TMDp->Content = Content_TMp->Text;
}
```

Offensichtlich wird der Text im Memo-Feld einfach nur dem Inhalt des Dokuments zugewiesen. Durch die Zuweisung aktualisiert das Dokument seinen Inhalt und löst seinerseits wieder ein Update aller registrierten Views aus, was uns wiederum gleich zum nächsten Tipp bringt:

! Je häufiger das der Wert im Dokument geändert wird, desto Häufiger löst der Update-Event aus und desto häufiger kommt die Renderfunktion zur Ausführung. Je nach dem wie kurz die Renderfunktion gehalten ist, hat das unter Umständen ausgesprochen negative Auswirkungen auf das Laufzeitverhalten. Hier muss allerdings wohl jeder seine eigenen Erfahrungen sammeln.

Eine Variante – vor Allem bei Dokumenten mit vielen Werten die aktualisiert werden – die Rechenlast für das Rendern der Daten tiefer zu halten ist es, die Funktion „BeginUpdate" bzw. „EndUpdate" zu benutzen. Diese blockieren den Update nach dem Aufruf von BeginUpdate, bis die Funktion EndUpdate ausgeführt wurde. Dadurch lassen sich mehrere Werte aktualisieren, ohne dass die renderfunktion – unnötigerweise – nach jedem Wert aufgerufen wird.
Es gibt allerdings noch eine weitere kleine Sache, die es zu beachten gilt:

! Eine Gefahr am System gibt es allerdings: Es kommt hin und wieder vor, dass man Rekursivitäten übersieht. Nicht aus Schlamperei, sondern einfach, weil die Rekursivität nicht offensichtlich ist. Worauf ich genau hinaus will ist folgendes: Ein Event (A) wird ausgelöst. Dieser wiederum löst einige weitere Events aus (B, C und D). Dabei ruft Event D eine Funktion E auf, welche ihrerseits wiederum implizit Event A auslöst. Konkret auf unser System übertragen könnte dies folgendes bedeuten: Ein Wert in einem Contol ändert sich. Wie vorgesehen, löst dieses Control einen Update des dokuments aus. Innerhalb der Renderfunktion eines Views wird durch das Aktualisieren einer Anzeige implizit wieder ein Event ausgelöst, welcher schliesslich dazu führt, dass wieder ein Update-Event ausgelöst wird.
Dadurch erhalten wir eine Endlos-Schleife welche schlussendlich in einem Stackoverflow endet.

Einen Weg diese Schleife prinzipiell zu durchbrechen ist es, wenn man folgendes Konstrukt in die Setz-Methoden des Dokuments einfügt:

```
void __fastcall TMyDocument::SetContent(AnsiString value)
{
    if(FContent_AS != value)
    {
        FContent_AS = value;
        UpdateAllViews();  //-- Update erzwingen
    }
}
```

Dank diesem Konstrukt wird der Update der Views nur dann ausgelöst, wenn der neue Wert wirklich vom alten Wert abweicht. Für eine obig beschriebene Endlos-Schleife bedeutet dies, dass bereits im ersten Durchlauf die schleife durchbrochen wird.

Wie kann man nun Daten speichern oder laden?
Das ist nun alles schön und gut, wie können nun allerdings Daten in das Dokument geladen bzw. aus dem Dokument gespeichert werden?

Zu diesem Zweck dient die Implementation der Methoden Save- bzw. LoadDocument. Wer sonst, wenn nicht das Dokument weiss, welche Werte wie in Dateien gespeichert liegen bzw. wie die Werte in die Dateien gespeichert werden müssen.

Muss das Dokument nun die Werte speichern oder laden, übergibt man der Methode eifach den Dateinamen. Um den Rest kümmern sich die einzelnen Methoden.
Bei den Methoden Load-/SaveDocument macht es zum Beispiel auch Sinn, die Begin-/EndUpdate-Funktionen aufzurufen, um ein permanentes (unnötiges) Refreshen der Views zu unterbinden.

Document/View at it's best
Prinzipiell haben wir ja nun die Applikation fertig. Document/View wurde implementiert und ich habe noch immer ein Versprechen offen. Ich hatte nämlich vesprochen, anhand eines Beispiels die Vorzüü-

ge von Document/View in Sachen Erweiterbarkeit aufzuzeigen. Dies habe ich jetzt jedoch auf die lange Bank geschoben.

Jetzt, da wir unsere Basis geschaffen haben, wird es Zeit das Versprechen einzulösen. Stellen wir uns folgende Szene vor:
Jemand kommt und verlangt zu unserem Editor noch eine Anzeige der Anzahl Wörter und Zeichen innerhalb des Dokuments.

Lösung mittels konventionellem Applikationsdesign
Well, wie hätten wir das früher gemacht? Ein Formular gebaut, unser Hauptformular so umgebaut, dass das Formular darauf reagieren kann, wenn sich ein Wert ändert. Anschliessend hätten wir vom Hauptformular die Basisdaten gezogen um die Werte zu errechnen und irgendwie wäre das Gebilde wie in Abbildung 1 eingeläutet worden.

Lösung mit Document/View
Wie wäre die Lösung wohl mit Document/View ausgefallen? Ganz einfach: Wir designen ein neues Formular, welches in der Funktion UpdateData() aus dem Dokument die nötigen Daten extrahiert und anzeigt. Das ermöglicht uns einen erheblichen Unterschied zu obigem Weg: Das (alte) Hauptformular bleibt unangetastet. Die Entwicklung konzentriert sich nur auf die wirklich zu ändernden Teile, in unserem Fall, das neue Formular.

Um das genannte nochmals zu verdeutlichen, hier nochmals das Listing aus dem Dokument-Eigenschaften Dialog, welches für das Rendern und Anzeigen der Daten zuständig ist.

```
void __fastcall TDocData_TFp::UpdateData(TDocument * Sender_TDp)
{
    /* Die Dokumentendaten aufgrund der neuen Daten anpassen */
    TMyDocument *Doc_TMDp = dynamic_cast<TMyDocument *>(Sender_TDp);

    if (Doc_TMDp != NULL)
    {
        CharCount_TLp->Caption = CharCount_TLp->Caption.sprintf(
...
```

Wir designen ein neues Formular, welches in der Funktion UpdateData() aus dem Dokument die nötigen Daten extrahiert und anzeigt.

Wir brauchen keines der alten Formulare anzugreifen oder zu ändern. Spätestens wenn sich die Views häufen, zahlt sich diese Methode umso stärker aus…

Epilog
So, das Tutorial umfasst unterdessen 11 Seiten. Mehr als ich eigentlich je schreiben wollte. Vermutlich ist es auch schon wieder zu lang und so umständlich geschrieben, dass 60% der Leser bereits in der Hälfte aufgegeben hat. Solltest du eine der Ausnahmen darstellen, welche sich bis hier her durchgekämpft hat, so möchte ich dir zunächst mal für dein Interesse danken.

Wie ich im Rahmen des Prologs schon gesagt habe, ist dieses Tutorial kein absolutes Kochrezept, sondern vielmehr ein Denkanstoss. – Nein, das wird jetzt nicht die Wiederholung des Prologs. Ich halte euch alle für so intelligent, dass ihr das schon dort begriffen habt (o;
Wer sich bereits an ein - zwei Gedanken zu dem Thema Document/View gestossen hat, dem empfehle ich die weitere Lektüre von Informationen rund ums Thema MVC. MVC ist eigentlich der Ursprung von Document/View und bietet noch einige Möglichkeiten mehr als Doc/View. Leider lässt es der Umfang dieses Tutorials nicht zu, dass ich dieses Modell ebenfalls in diesem Tut behandle. Wer weiss, wenn ich die Musse habe, und mir meine 50min Arbeitsweg langweilig werden, kümmere ich mich mal um ein Tut um MVC welches auf diesem hier aufbaut. Bis dahin empfehle ich das PDF welches auf Hume-Sikkins Webpage (findet ihr in seiner Signatur im Forum auf c-plusplus.de) liegt als Lektüre.

Wer Freude und Lust hat, darf die Infos dort auch gerne so aufbereiten, dass man das Tutorial hier ergänzen kann (o:

So, Word zeigt nun 12 Seiten an. Ich denke das ist der Richtige Zeitpunkt aufzuhören. Wenn Ihr Fragen, Feedback, Lob oder Prügel (verbal natürlich *g*) habt, so benutzt doch die Adresse doc_view_tutorial@junix.ch um diese loszuwerden. Allfällige Updates dieses Artikels oder eine Beispielapplikation findet man unter:
http://www.junix.ch/bcb/help/doc_view

Ich wünsche euch allen frohes Coden und viel Spass mit allen neuen Erkenntnissen die Ihr (vielleicht) gewonnen habt.

Euer
junix