RE-ENGINEERING DER BASISKLASSEN FÜR DIE BENUTZEROBERFLÄCHE

Dokument zur Diplomarbeit
- Designphase -

Autoren	Thomas Kullmann, Günther Reinecker
Dokumentversion	1.0
Zustand	abgeschlossen
letzte Bearbeitung	21.08.02

Inhalt

I Überblick

Die Dokumentation wurde stark formalisiert, für die Layoutkonventionen siehe [1].

Die Implementation wurde so angelegt, dass sie sowohl unter Win16 (Windows® 3.x, 9x, Me) als auch Win32 (NT, 2000, XP, …) funktioniert – damit sind die Fenster auf allen derzeit verfügbaren Microsoft-Windows® - Betriebsystemen lauffähig. Die noch benötigten alten 16Bit-Funktionen werden jeweils durch Präprozessor-Anweisungen ausgeschlossen, wenn eine Win32-Umgebung vorliegt und diese Funktionen nicht mehr benötigt werden. Die Fenster wurden auch unter Microsoft Visual C++ ® erfolgreich getestet.

Durch das System von Windowsbotschaften sind die Entwurfsmöglichkeiten von neuen Fensterklassen nicht sehr groß. Das vereinfacht jedoch das Reverse-Engineering aller im XCTRL vorhandenen Fenster erheblich. Leider kommt dadurch ein Großteil der Windowsbotschaften (so genannte **Kommandos**) in einer einzigen Methode – `OnCommand` bzw. `Dlg_OnCommand` – an. Wir empfehlen die für diesen Anwendungsfall typische `switch`-Anweisung übersichtlich zu gestalten und für jede einzelne Ressourcen-Id (oder einer Klasse Ressourcen-Ids, mit gleichem oder ähnlichem Verhalten) eine Methode zu implementieren, und diese in den `case` –Zweigen aufzurufen.

Tipp: Das Suffix `On` vor einer solchen Methode erleichtert das Einlesen in die Codepassagen.

► Dokumentation des Istzustands

Die Dokumentation bezieht sich auf den Quellcode der unten aufgelisteten Dateien (Stand: 11.05.2003). Nachfolgende Änderungen oder Neuimplementationen, können nicht berücksichtigt werden!

h-Dateien (Deklaration)	cpp-Dateien (Implementation)
• SWINTRAC.H	• DLG_TPL.CPP

Tabelle 1 „Auflistung der zum Subsystem zugehörigen Dateien“ (Quelle: selbst)

■ Analyse und Beseitigung von Mängeln des alten Subsystems

Die zum Erstellen und Anzeigen der modalen und nicht modalen Dialoge verwendeten API[1]-Methoden müssen unter Win32 und Win16 verwendet werden können! Dadurch ist es jedoch nicht möglich, dass die Windowsbotschaften direkt an Objekte weitergeleitet werden, weil als Empfänger dieser “Nachrichten“ nur eine globale oder `static` Methode angegeben werden kann – i.d.R. heißt diese `DialogProc`. Wenn hier eine Botschaft ankommt, muss man sich um ihre Weiterleitung an das richtige Objekt selbst kümmern.

Das war im alten Subsystem fehlerhaft umgesetzt, weil bei mehr als einem nicht modalen Dialog alle Botschaften an das jeweils zuletzt erzeugte Fenster weitergeleitet wurden. Wohl deshalb wurde auf den Einsatz solcher Fenster im XCTRL fast völlig verzichtet.

Diese Probleme bestand auch bei den modalen Dialogen, weil hier aber nur genau ein Fenster gesteuert werden kann, führte dies zu keinen schwerwiegenden Fehlern. Trotzdem wurden alle Botschaften (auch der verdeckten Fenster) nur vom obersten Fenster verarbeitet!

Außerdem waren viele Methoden in `TModalDlg` und `TModelessDlg` identisch deklariert und z.T. auch doppelt implementiert.

■ die neue Vererbungshierarchie

`TBasicWindow` ist die Basis für alle Fensterklassen, hier erfolgt auch die Zuordnung zwischen dem jeweiligen **Fensterobjekt** und seinem **-handle** – fehlerfrei für “nahezu“ beliebig viele Fenster (das verwendete Prinzip wird unten näher erläutert und ist an **[3]** angelehnt). Die Eigenschaften, die im alten `TModalDlg` und `TModelessDlg` parallel vorhanden waren, befinden sich nun in `TBasicDialog`, weil die Signatur bei den MDI-Fenstern meist anders aussieht. `TBasicDialog` bietet zusätzlich die Funktionalität zum Verwalten des Ressourcenbezeichners vom Dialogfenster. Hier sind zahlreiche Methoden implementiert, die den Zugriff auf die Steuerelemente erleichtern.

Die Klasse `TModalDlg` implementiert das Anzeige- und Schließverwalten von modalen Dialogfenstern. Im Gegensatz dazu bietet `TModelessDlg` die Funktionalität für nicht modale Dialogfenster. Von `TModalDlg` und `TModelessDlg` können neue Dialogklassen abgeleitet werden. Beim Konstruktoraufruf kann die Bezeichnung einer Ressource angeben werden, die automatisch aus den Ressourcen geladen und angezeigt wird. Durch Überschreiben von virtuellen Methoden aus `TBasicDialog`, kann man das Verhalten des Fensters an die eigenen Bedürfnisse anpassen.

Die von `TBasicDialog` abgeleiteten Klassen müssen sich aber selbst um die, weiter unter erläuterte, Anmeldung und Unregistrierung der Fenster kümmern! `TModalDlg` und `TModelessDlg` implementieren bereits dieses Verhalten.

`TBasicMDIWindow` ist die einfachste Version eines MDI-Fensters und nimmt der, an ‚XCtl’ angepassten, Klasse `TMDIWindow` einen Teil der Funktionalität ab.

[1] **A**pplication **P**rogramming **I**nterface

■ **weiterleiten der Botschaften an das jeweilige Objekt (Methode `TBasicWindow::EventHandler`)**

Die von `TBasicWindow` abgeleiteten Klassen müssen die Botschaften der erzeugten Fenster an die `static`-Methode `TBasicWindow::EventHandler` umleiten[2]. Damit kommen die Windowsbotschaften aller so erzeugten Fenster jedoch bei nur einer Methode an!

Jede empfangene Windowsbotschaft enthält aber auch den Adressaten, der diese Botschaft empfangen soll, das *Fensterhandle*. Die Methode `Eventhandler` kennt, über den unten beschriebenen Mechanismus, die *Handle* von allen registrierten Objekten (Fenstern). So kann jede Botschaft, über das *Fensterhandle* identifiziert, an das richtige Objekt weitergeleitet und dort verarbeitet werden.

[2] Dies geschieht durch den Aufruf einer *API*-Methode die das Anzeigen eines Fensters vorbereitet – (modale Dialoge: `DialogBoxParam`; nicht modale Dialoge: `CreateDialog`).

■ **Anmelden und Registrieren von Objekten (für Methode `TBasicWindow::EventHandler`)**

Erst wenn das Fenster zum ersten Mal angezeigt wird, erhält es sein *Handle* per erster Windowsbotschaft, die (über den o.g. Mechanismus, siehe **Abbildung 1**, **A**) bereits an Methode `TBasicWindow::EventHandler` weitergeleitet wird. Das Objekt ist zu diesem Zeitpunkt noch nicht registriert, weil es erst durch diese Botschaft das *Handle* erhält und damit noch nicht registriert werden konnte!

Die Methode `EventHandler` versucht nun die Botschaft zu einem bereits registrierten Objekt weiterzuleiten. Das schlägt fehl (**C**), weil das *Fensterhandle* [noch] nicht registriert ist. Deshalb muss sich ein Fenster, bevor es zum ersten Mal angezeigt wird, bei `TBasicWindow` anmelden. Dazu setzt man das `static`-Attribut `s_WaitingWindow= this` (**B**). Wenn nun eine Botschaft für ein unbekanntes *Fensterhandle* empfangen wird, können Objekt (`s_WaitingWindow`) und Handle (der Botschaft) automatisch registriert werden (**D**), `s_WaitingWindow` wird wieder `0` gesetzt.

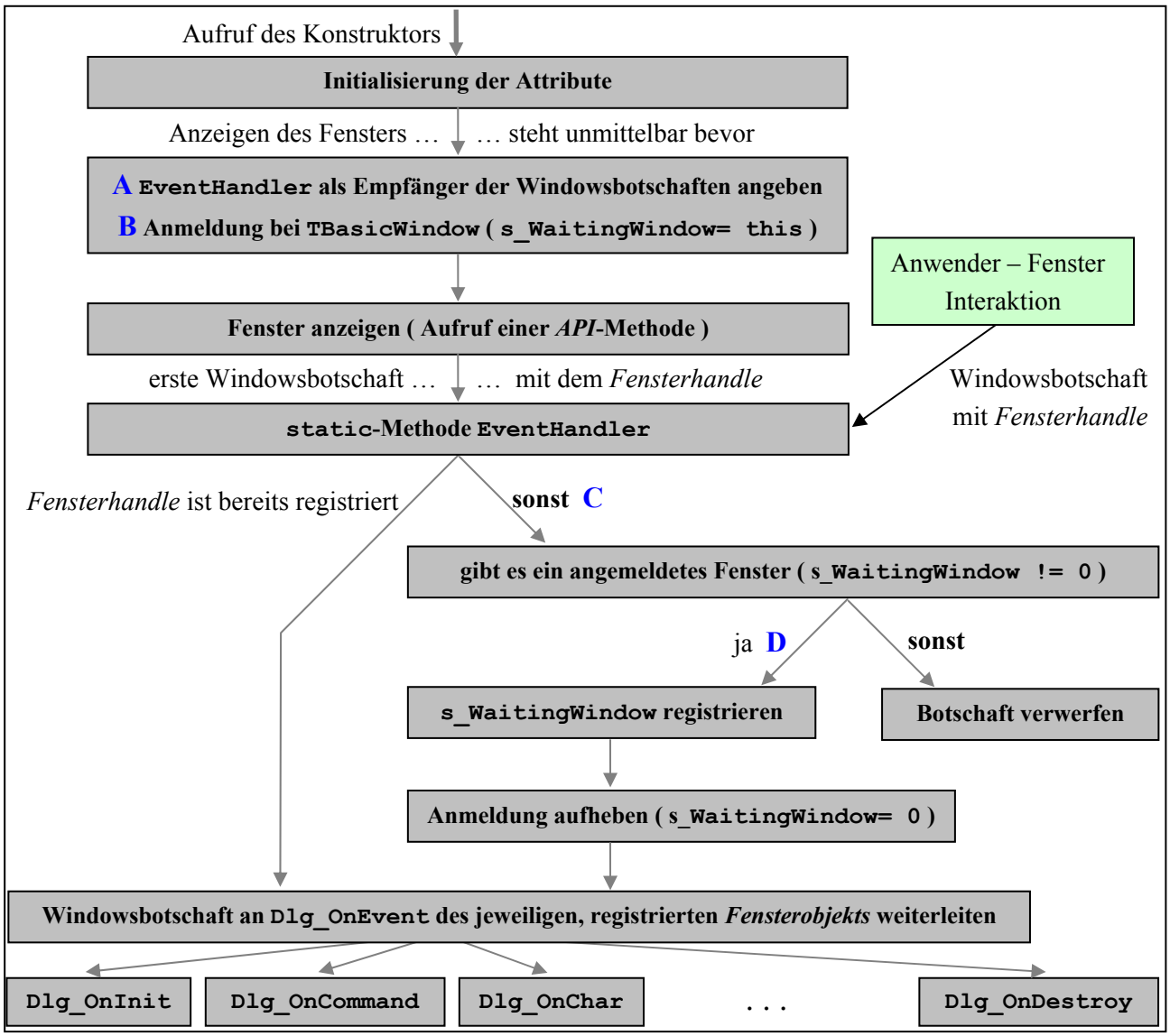

Abbildung 1 „Anmelden und Registrieren von Fenstern; weiterleiten der Windowsbotschaften durch `TBasicWindow`“ (Quelle. selbst)

■ **Unregistrierung (Aufheben der Registrierung, sobald ein Fenster sein *Handle* verliert)**
Nachdem ein Fenster endgültig geschlossen wurde und damit sein *Fensterhandle* verloren hat, muss die Registrierung wieder aufgehoben werden, damit es später nicht zu Fehlersituationen kommt, wenn das *Handle* an ein neues Fenster vergeben wird. Dazu muss jedes Objekt selbstständig Methode `UnregisterWindow()` aufrufen, wo Objekt und *Handle* wieder aus der Liste der registrierten Fenster entfernt werden.

■ **Verwaltung der registrierten Fenster**
Die Liste der registrierten Fenster muss, um für die `static`-Methode `TBasicWindow::EventHandler` erreichbar zu sein, ebenfalls `static` sein. `TBasicWindow::s_First` zeigt auf den Anfang der verkettete Liste, die alle registrierten Fenster enthält. Über das Attribut `m_Next` (das nicht `static` ist), kann man sich bis zum Ende der Liste "durchhangeln" – das Ende ist `0`-terminiert. Über das `private` Attribut `m_Window`, hat Methode `EventHandler` auch Zugriff auf das *Fensterhandle*.

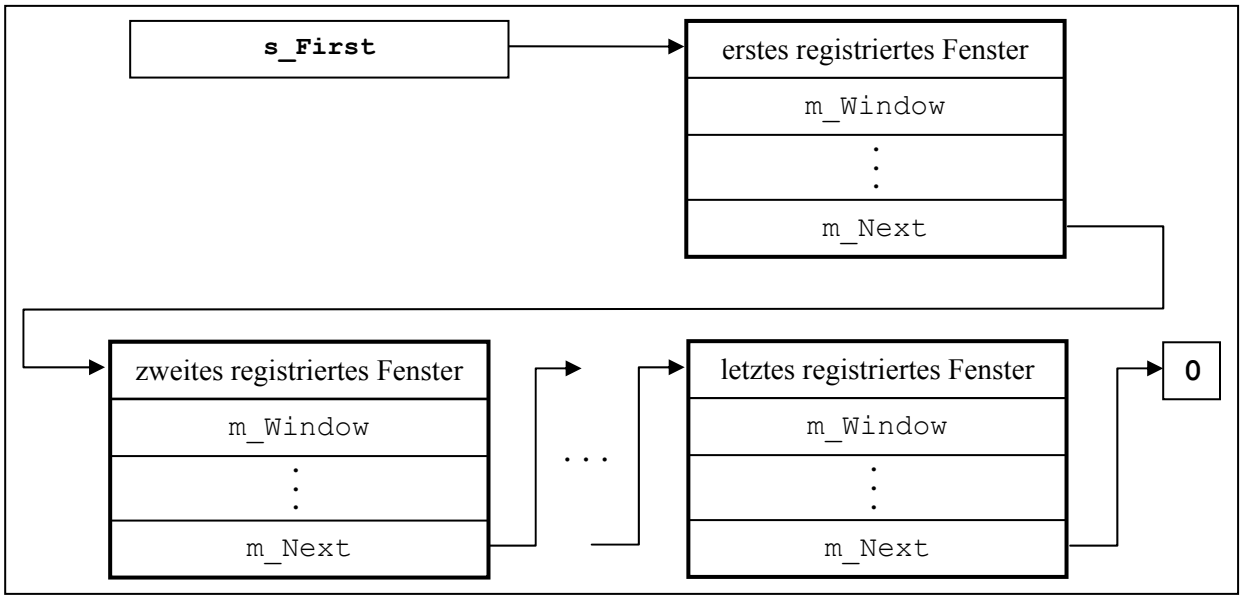

Abbildung 2 „Verwaltung der registrierten Fenster als verkettete Liste" (Quelle: selbst)

II Klassen

Abbildung 3 „UML-Klassendiagramm der Basisklassen für Dialogfenster“ (Quelle: Together®, Version 6.0)

II.1 Klasse TBasicWindow

Deklaration : SWINTRAC.H

Implementation : DLG_TPL.CPP

Es gibt keine abstrakten Methoden, d.h. Objekte von `TBasicWindow` könnten erzeugt werden. Da jedoch der Registrierungs-, Anmeldungs- und Unregistrierungsvorgang fehlen und nirgends ein Fenster erstellt wird, ist es nicht sinnvoll Objekte zu erstellen.

II.1.1 Attribute

▶ **`HWND m_Window`** **PROTECTED**

Handle des Fensters; `0` wenn das Fenster nicht registriert ist

▶ **`TBasicWindow *s_First`** **STATIC PRIVATE**

Zeiger auf den Anfang der Liste der registrierten Fenster; das nächste Listenelement ist jeweils über `m_Next` erreichbar – solange bis `0` das Ende der Liste kennzeichnet. siehe **Abbildung 2**

0 wenn es kein registriertes Fenster gibt.

▶ **`TBasicWindow *m_Next`** **PRIVATE**

Zeiger auf das nächste registrierte Fenster; `0` wenn dieses Fenster nicht registriert ist oder wenn dieses Fenster das letzte registrierte Fenster ist; `s_First` ist der Anfang der Liste; siehe **Abbildung 2**

▶ **`TBasicWindow *s_WaitingWindow`** **STATIC PROTECTED**

Zeiger auf das angemeldete Fenster, dass noch nicht registriert ist, in aber Kürze die erste Botschaft empfängt; zur Registrierung dieses Fenster benötigt; siehe **Abbildung 1**

II.1.2 Methoden

▶ **`TBasicWindow( void )`** **PUBLIC**

Im Konstruktor werden alle Attribute initialisiert (siehe **II.1.1 Attribute**).

▶ **`TBasicWindow( TBasicWindow& )`** **PRIVATE**

Der Copy-Konstruktor ist leer implementiert, um ungewünschte Kopienbildung zu verhindern.

▶ **`virtual ~TBasicWindow( void )`** **PUBLIC**

Der Destruktor unregistriert das Fenster, weil dieses beim Destruktoraufruf bereits das Handle verloren hat – siehe **Unregistrierung (Aufheben der Registrierung, sobald ein Fenster sein Handle verliert)**.

▶ **`HWND GetHandle( void )`** **PUBLIC**

gibt den Inhalt von `m_Window` (*Fensterhandle* mit dem dieses Objekt registriert) zurück oder `0`

▶ **`void RegisterWindow( HWND )`** **PROTECTED**

registriert dieses Objekt mit dem, als Parameter übergebenen, *Fensterhandle* und setzt `m_Window` auf dieses *Handle*

▶ **`void UnregisterWindow( void )`** **PROTECTED**

hebt die Registrierung für dieses Objekt auf und setzt `m_Window`= 0

► **BOOL CALLBACK EventHandler(HWND, UINT, WPARAM, LPARAM)** **STATIC PROTECTED**

empfängt die Windowsbotschaften aller (von `TBasicWindow` abgeleiteten) Fenster und leitet sie an das entsprechende Objekt, Methode `OnEvent`, weiter – Identifikation über das *Fensterhandle* im ersten Parameter;

Ist das Objekt noch nicht registriert, wird geprüft ob ein angemeldetes Fenster verfügbar ist. Wenn ja wird dieses registriert, `s_WaitingWindow= 0` gesetzt (damit gibt es kein angemeldetes Fenster mehr) und die Botschaft wird an das soeben registrierte Objekt weitergeleitet. siehe **Abbildung 1**.

► **virtual LRESULT OnEvent(HWND, UINT, WPARAM, LPARAM)** **PROTECTED**

empfängt die Windowsbotschaften, die, über das *Fensterhandle* identifiziert, an dieses Objekt weitergeleitet wurden; Entsprechend dem zweiten Parameter (Typ der Botschaft), kann wird die Botschaft eine spezielle Methode weitergeleitet werden (siehe `TBasicDialog`). Die letzten beiden Parameter werden botschaftsspezifisch belegt.

Wenn ein abgeleitetes Objekt eine Botschaft behandeln will, sollte diese Methode überschrieben werden – vorher ist zu prüfen, ob die Botschaft in einer anderen Basisklasse überschrieben und an eine Botschaftenbehandlungsroutine weitergeleitet wird!

► **BOOL SetActivateHotKey(const char, const BOOL, const BOOL, const BOOL, const BOOL)** **PUBLIC**

registriert die, in den Parametern angegebene, Tastenkombination; Der erste Parameter bezeichnet eine "normale" Taste – hier sollten nur „`VK_`"-Konstanten angegeben werden. Die folgenden Parameter bezeichnen (in dieser Reihenfolge), ob die [SHIFT], [CTRL], [ALT] und [<erweitert>] gedrückt werden sollen – wobei niemand weiß, was <erweitert> bedeutet. Der Rückgabewert gibt an, ob die Tastenkombination erfolgreich registriert wurde (`TRUE`) oder ob Fehler auftraten (`FALSE`).

Wenn die Tastenkombination gedrückt wird und das Fenster inaktiv ist, wird das Fenster aktiviert. Wenn das Fenster bereits aktiv ist, wird eine `WM_SYSCOMMAND`-Botschaft mit `wParam=SC_HOTKEY` gesendet.

Win16: Diese Art Tastenkombination funktioniert unter Win16 und Win32.

ACHTUNG: Von dieser Art Tastenkombination kann nur max. eine definiert werden – wenn bereits eine solche Tastenkombination definiert ist, wird diese überschrieben.

► **BOOL AddHotKey(const char, const BOOL, const BOOL, const BOOL, const BOOL)** **PUBLIC**

registriert die, in den Parametern angegebene, Tastenkombination; Der erste Parameter bezeichnet eine "normale" Taste – hier sollten nur „`VK_`"-Konstanten angegeben werden. Die folgenden Parameter bezeichnen (in dieser Reihenfolge), ob die [SHIFT], [CTRL], [ALT] und [<WINDOWSTASTE>].gedrückt werden sollen. Der Rückgabewert gibt an, ob die Tastenkombination erfolgreich registriert wurde (`TRUE`) oder ob Fehler auftraten (`FALSE`).

Win32: Diese Art Tastenkombination funktioniert nur unter Win32, hier können "nahezu" beliebig viele Tastenkombinationen registriert werden. Unter Win16 wird stets `FALSE` zurückgegeben.

Wenn die Tastenkombination gedrückt wurde, egal ob das Fenster aktiv oder inaktiv ist, wird eine `WM_HOTKEY`-Botschaft mit `lParam=<KeyCode>` gesendet. Um zu ermitteln, welche der so registrierten Tastenkombinationen gedrückt wurde, kann man Methode `IsHotKey ( …, …, …, …, …, … )` verwenden – dort wird der erste Parameter (`lParam`) mit dem `KeyCode` verglichen, der aus den übrigen Parametern berechnet wird – bei Übereinstimmung wird dort `TRUE` zurückgegeben, sonst `FALSE`.

► **UINT LoadHotKeys(LPCSTR) PUBLIC**

lädt die Accelerator-Tabelle (mit der Ressourcen-Id die im Parameter angegeben wurde) aus der rc-Datei der `hModuleInstance`-Instanz, und registriert diese als Tastenkombination für dieses Fenster; Für jede Tastenkombination kann dort eine *Kommando*-Id angegeben werden, die beim Drücken der Tastenkombination ans Fenster gesendet werden soll, wenn dieses aktiviert ist. Der Rückgabewert gibt an, wie viele Tastenkombinationen erfolgreich geladen wurden.

Wenn das Fenster inaktiv ist, oder die angegebene *Kommando*-Id `-1` ist, wird beim Drücken der Tastenkombination so verfahren, wie bei Tastenkombinationen, die mit Methode `AddHotKey( …, …, …, …, … )` registriert wurden. Dann muss auch hier mit Methode `IsHotKey ( …, …, …, …, …, … )` geprüft werden, welche Tastenkombination gedrückt wurde.

Win32: Diese Art Tastenkombination funktioniert nur unter Win32, hier können "nahezu" beliebig viele Tastenkombinationen registriert werden. Unter Win16 wird stets `0` zurückgegeben.

```
   BOOL  IsHotKey( LPARAM, const char, const BOOL,          PUBLIC
►                          const BOOL, const BOOL,
                           const BOOL )
```

prüft, ob der Keycode, der im ersten Parameter angegeben wurde, identisch ist mit der Tastenkombination die in den übrigen Parametern übergeben wurde. Ist dies der Fall, wird `TRUE` zurückgegeben, sonst `FALSE`. Der zweite Parameter bezeichnet eine "normale" Taste – hier sollten nur „`VK_`"-Konstanten angegeben werden. Die letzten Parameter bezeichnen (in dieser Reihenfolge), ob die [SHIFT], [CTRL], [ALT] und [<WINDOWSTASTE>].gedrückt werden sollen. Der Rückgabewert gibt an, ob die Tastenkombination erfolgreich registriert wurde (`TRUE`) oder ob Fehler auftraten (`FALSE`). siehe auch Methode `AddHotKey( …, …, …, …, … )` und Methode `LoadKey( … )`

Win32: Weil diese Art Tastenkombination nur unter Win32 funktioniert, wird unter Win16 nur `FALSE` zurückgegeben.

II.1.3 Bewertung

Metrik		Kennung (min, max)	Wert
Klasse			
	‚Lines Of Code' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	233
	‚LOC of Implementation'*	LOCI	145
	‚LOC of Declaration'*	LOCD	88
	‚Number Of Attributes'	NOA (0, 30)	5
	‚Number Of Operations'	NOO (0, 50)	11
	‚Number Of Members' Attribute + Methoden	NOM = NOA + NOO	16
	‚Number Of Constructors'	NOCON (0, 5)	2
	‚Number Of Overridden Methods'	NOOM (0, 10)	0
	‚Percentage of Private Members'	PPrivM	21
	‚Percentage of Protected Members'	PProtM (0, 10)	42
	‚Percentage of Public Members'	PPubM	37
	‚Weighted Methods Per Class'	WMPC1 (0, 30)	
Attribute			
	‚Attribute Complexity'	AC	45
Methoden			
	‚Maximum Number Of Parameters'	MNOP (0, 4)	6
	‚Cyclomatic Complexity'	CC	
Kommentare			
	‚Number Of Comments'*	NOC	
	‚True Comment Ratio'	TCR (5, 400)	74

Tabelle 2 „ausgewählte Metriken der Klasse `TBasicWindow`" (Quelle: Together® ‚Version 6.0)

* Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

II.2 Klasse TBasicDialog

Deklaration : SWINTRAC.H

Implementation : DLG_TPL.CPP

Es gibt keine abstrakten Methoden, d.h. Objekte von `TBasicDialog` könnten erzeugt werden. Da jedoch der Registrierungs-, Anmeldungs- und Unregistrierungsvorgang fehlen und nirgends ein Fenster erstellt wird, ist es nicht sinnvoll Objekte zu erstellen.

II.2.1 Attribute

► **`char *m_ResName`** **PRIVATE**

ist der Name der Ressource für das Dialogfenster; wird im Konstruktor mit dem Wert des ersten Parameters initialisiert

II.2.2 Methoden

► **`TBasicDialog( const char* )`** **PUBLIC**

Im Konstruktor werden alle Attribute initialisiert (siehe **II.2.1 Attribute**). Der Parameter ist der Name der Ressource für das anzuzeigende Dialogfenster. Dieser wird in das dynamisch erzeugte `m_ResName` kopiert.

▶ **virtual ~TBasicDialog(void) PUBLIC**

Der Destruktor gibt den Speicher vom dynamisch erzeugten `m_ResName` wieder frei.

▶ **const char *GetResName(void) PUBLIC**

gibt `m_ResName` (Namen der Ressource für das anzuzeigende Dialogfenster) zurück

▶ **UINT CtrlGetText(const int, char*, const int) PUBLIC**

liest den Inhalt des Steuerelements (dessen Ressourcen-Id im ersten Parameter angegeben wird) und gibt das Ergebnis im zweiten Parameter zurück, dort sollte mehr Platz für Zeichen reserviert sein, als der letzte Parameter angibt – sind mehr Zeichen enthalten, wird der Rest abgeschnitten; siehe [2]: `GetDlgItemText`

Der Rückgabewert gibt an, wie viele Zeichen zurückgegeben wurden oder `0`, wenn kein solches Steuerelement existiert.

▶ **BOOL CtrlSetText(const int, LPCSTR) PUBLIC**

trägt den (als zweiten Parameter übergebenen) Text in das Steuerelement (dessen Ressourcen-Id im ersten Parameter angegeben wurde) ein; siehe [2]: `SetDlgItemText`

Rückgabewert: `TRUE` ↔ Steuerelement ist vorhanden und Text wurde erfolgreich eingetragen

▶ **long CtrlGetLong(const int, BOOL&) PUBLIC**

liest den Inhalt eines Steuerelements (dessen Ressourcen-Id im ersten Parameter angegeben wird), konvertiert ihn in einen ganzzahligen `Long`-Wert und gibt das Ergebnis zurück; Wenn das Steuerelement vorhanden ist und der gelesene Wert erfolgreich konvertiert werden konnte, wird im letzten Parameter `TRUE` zurückgegeben, sonst `FALSE`.

Wenn Fehler während der Konvertierung auftreten, wird eine Fehlermeldung ausgegeben. Enthält der zu konvertierende Text Nachkommastellen, werden diese ignoriert (abgeschnitten).

▶ **long CtrlGetLong(const int, PUBLIC**
const long, const long, BOOL&)

wie Methode `CtrlGetLong( … , … )`, der Rückgabewert wird jedoch zusätzlich ins Intervall [<zweiter Parameter> … <dritter Parameter>] eingepasst.

▶ **BOOL CtrlSetLong(const int, const long) PUBLIC**

konvertiert den (im zweiten Parameter übergebenen) `Long`-Wert in Text und trägt diesen in das Steuerelement (dessen Ressourcen-Id im ersten Parameter angegeben wurde) ein

Rückgabewert: `TRUE` ↔ Steuerelement ist vorhanden und Wert erfolgreich konvertiert und eingetragen

▶ **double CtrlGetDouble(const int, const UINT, PUBLIC**
const char, BOOL&)

wie Methode `CtrlGetLong( …, … )`, allerdings werden hier `Double`-Werte akzeptiert deren Nachkommastellen im zweiten Parameter angegeben werden – sind mehr Stellen vorhanden, wird der Rest ignoriert (abgeschnitten). Das Trennzeichen (Komma) wird im dritten Parameter angegeben.

double CtrlGetDouble(const int, PUBLIC
▶ **const double, const double,**
const UINT, const char, BOOL&)

wie Methode `CtrlGetDouble( …, …, …, … )`, der Rückgabewert wird jedoch zusätzlich ins Intervall [<neuer zweiter Parameter> … <neuer dritter Parameter>] eingepasst.

▶ **BOOL CtrlSetDouble(const int, const double, const UINT, const char)** **PUBLIC**

konvertiert den (im zweiten Parameter übergebenen) `Double`-Wert in Text und trägt diesen in das Steuerelement (dessen Ressourcen-Id im ersten Parameter angegeben wurde) ein; der dritte Parameter gibt die Anzahl der Nachkommastellen an (bei zu vielen Nachkommastellen wird der Rest ignoriert – abgeschnitten). Das Trennzeichen (Komma) wird im letzten Parameter angeben.

Rückgabewert: `TRUE` ↔ Steuerelement ist vorhanden und Wert erfolgreich konvertiert und eingetragen

▶ **HWND Ctrl(const int)** **PUBLIC**

ermittelt das *Handle* des Steuerelements (dessen Ressourcen-Id der Parameter angibt); im Fehlerfall `0`; siehe **[2]**: `GetDlgItem`

▶ **BOOL CtrlSetFocus(const int)** **PUBLIC**

fokussiert das Steuerelement (dessen Ressourcen-Id der Parameter angibt); siehe **[2]**: `SetFocus`
Rückgabewert: `TRUE` ↔ Steuerelement ist vorhanden und wurde fokussiert

▶ **BOOL CtrlSetEnable(const int, const BOOL)** **PUBLIC**

je nachdem was der zweite Parameter angibt, wird das Steuerelement (dessen Ressourcen-Id der erste Parameter angibt) freigeben (`TRUE`) oder gesperrt und ausgegraut (`FALSE`); siehe **[2]**: `EnableWindow`

Rückgabewert: `TRUE` ↔ Steuerelement ist vorhanden und war bereits vor diesem Methodeaufruf gesperrt und ausgegraut (Rückgabewert von `EnableWindow`)

▶ **virtual LRESULT OnEvent(HWND, UINT, WPARAM, LPARAM)** **PROTECTED**

überschrieben um ausgewählte Botschaften an eine der folgenden `Dlg_On`<irgendwas>-Methode weiterzuleiten

▶ **virtual BOOL Dlg_OnInit(HWND, HWND, LPARAM)** **PROTECTED**

wird aufgerufen sobald eine `WM_INITDIALOG` Botschaft (als Initialisierungsaufforderung des Fensters – i.d.R. die erste Botschaft bei modalen und nicht modalen Dialogfenstern) empfangen wurde; ist leer implementiert

▶ **virtual void Dlg_OnCommand(HWND, int, HWND, UINT)** **PROTECTED**

wird aufgerufen sobald eine `WM_COMMAND` Botschaft (zur Behandlung von *Kommandos*) empfangen wurde; Der erste Parameter ist ein *Handle* auf das Fenster, dass die Botschaft ausgelöst hat, der zweite ist die Id des Steuerelements das das *Kommando* ausgelöst hat. Die letzten beiden Parameter werden kommandospezifisch belegt (siehe **[2]**). ist leer implementiert

▶ **virtual void Dlg_OnHScrollBar(HWND, HWND, UINT, int)** **PROTECTED**

wird aufgerufen sobald eine `WM_HSCROLL` Botschaft (Aktion einer horizontalen Bildlaufleiste) empfangen wurde; Der erste Parameter ist das *Fensterhandle*, das zweite ist das *Handle* der Bildlaufleiste; der dritte ist der Typ[3] der Botschaft und der letzte ist die neue Position des Bildlauffeldes. ist leer implementiert

[3] `SB_LINEDOWN`, `SB_PAGEDOWN`, `SB_BOTTOM`, `SB_THUMBPOSITION`, `SB_LINEUP`, `SB_PAGEUP`, `SB_TOP`, oder `SB_ENDSCROLL`

► **virtual void Dlg_OnVScrollBar(HWND, HWND, UINT, int) PROTECTED**
wie Methode `DlgOnHScrollBar`, nur für vertikale Bildlaufleisten – bei Empfang von `WM_VSCROLL`; ist ebenfalls leer implementiert

► **virtual void Dlg_OnTimer(HWND, UINT) PROTECTED**
wird aufgerufen sobald eine `WM_TIMER` Botschaft (Timerereignis ist eingetreten, das mit diesem Fenster verknüpft ist – API-Methode `StartTimer`) empfangen wurde; Der erste Parameter ist das *Fensterhandle*, der zweite die Identifikation des Timers. ist leer implementiert

► **virtual void Dlg_OnLButtonUp(HWND, int, int, UINT) PROTECTED**
wird gerufen sobald eine `WM_LBUTTONUP` Botschaft (linke Maustaste wurde losgelassen) empfangen wurde; ist leer implementiert

► **virtual void Dlg_OnLButtonDown(HWND, UINT, PROTECTED**
int, int, UINT)
wird gerufen sobald eine `WM_LBUTTONDOWN` Botschaft (linke Maustaste wurde gedrückt) empfangen wurde; ist leer implementiert

► **virtual void Dlg_OnRButtonDown(HWND, UINT, PROTECTED**
int, int, UINT)
wie Methode `Dlg_OnLButtonDown`, nur für die rechte Maustaste – bei Empfang von `WM_LBUTTONDOWN`; ist leer implementiert

► **virtual void Dlg_OnMouseMove(HWND, int, int, UINT) PROTECTED**
wird aufgerufen sobald eine `WM_MOUSEMOVE` Botschaft (Mauszeiger wurde bewegt) empfangen wurde; ist leer implementiert

► **virtual int Dlg_OnMouseActivate(HWND, HWND, PROTECTED**
UINT, UINT)
wird aufgerufen sobald eine `WM_MOUSEACTIVATE` Botschaft empfangen wurde; ist leer implementiert

► **virtual void Dlg_OnDestroy(HWND) PROTECTED**
wird aufgerufen sobald eine `WM_DESTROY` Botschaft empfangen wurde; ist leer implementiert

► **virtual void Dlg_OnSetFocus(HWND, HWND) PROTECTED**
wird aufgerufen sobald eine `WM_SETFOCUS` Botschaft (das Steuerelement mit dem – als zweiten Parameter übergebenen – *Fensterhandle* hat den Fokus verloren) empfangen wurde; ist leer implementiert

► **virtual BOOL Dlg_OnChar(HWND, WPARAM, LPARAM) PROTECTED**
wird aufgerufen sobald eine `WM_CHAR` Botschaft (im Fenster mit dem – als ersten Parameter übergebenen – *Fensterhandle* wurde eine Taste gedrückt und losgelassen) empfangen wurde; Der Rückgabewert gibt an, ob die Botschaft verarbeitet wurde. nach dem Loslassen der [ESC]-Taste wird Methode `Interrupt` aufzurufen; Rückgabewert stets `TRUE`

► **virtual void Interrupt(void) PROTECTED**
wird aufgerufen; wenn das spezielle Tastaturereignis `VK_ESCAPE` ([ESC]-Taste wurde gedrückt) empfangen wurde; ist leer implementiert

► **`virtual BOOL CanClose( void )`** **PROTECTED**

Der Rückgabewert gibt an, dass das Dialogfenster bei *Kommando* `IDOK` geschlossen werden darf. gibt hier stets `TRUE` (Fenster darf geschlossen werden) zurück

► **`virtual void LeaveDialog( void )`** **PROTECTED**

wird gerufen, wenn das Dialogfenster geschlossen werden soll; ist hier leer implementiert

II.2.3 Bewertung

Metrik		Kennung (min, max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	433
	‚**LOC** of **I**mplementation'*	LOCI	220
	‚**LOC** of **D**eclaration'*	LOCD	213
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	1
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	39
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	40
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	1
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	2
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	**43**
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	55
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	9
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	**6**
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'*	NOC	
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	55

Tabelle 3 „ausgewählte Metriken der Klasse `TBasicDialog`" (Quelle: Together® ‚Version 6.0)

* Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

II.3 Klasse TModalDlg

Deklaration : SWINTRAC.H

Implementation : DLG_TPL.CPP

Es gibt keine abstrakten Methoden, d.h. Objekte von `TModalDlg` könnten erzeugt werden. Das ist jedoch nicht empfehlenswert, weil nur das Schließungsverhalten implementiert ist.

II.3.1 Attribute

▶ **`FARPROC m_Handler`** **PROTECTED**

ist der Zeiger auf die `CALLBACK`-Funktion `TBasicWindow::EventHandler`; Muss beim Laden der Ressourcen als Parameter übergeben werden (aus [2]). wird im Konstruktor initialisiert (Win16: Muss zuvor in die Instanz der Anwendung aufgenommen und am Ende wieder entfernt werden)

II.3.2 Methoden

▶ **`TModalDlg( const char* )`** **PUBLIC**

Im Konstruktor werden alle Attribute initialisiert (siehe **II.3.1 Attribute**). Der Parameter ist der Name der Ressource für das anzuzeigende Dialogfenster.

▶ **`virtual ~TModalDlg( void )`** **PUBLIC**

Der Destruktor gibt den Zeiger auf die `CALLBACK`-Funktion wieder frei (nur bei Win16) und hebt die Registrierung des Fensters auf (siehe **Dokumentation des Istzustands**).

▶ **`virtual int ExecuteDialog( HWND )`** **PUBLIC**

erzeugt das Dialogfenster und zeigt es an; Der Rückgabewert gibt Auskunft, weshalb das Dialogfenster geschlossen wurde.

▶ **`virtual void Dlg_OnCommand( HWND, int, HWND, UINT )`** **PROTECTED**

wird überschrieben, um das Standardverhalten zum Schließen des Dialogfensters (`IDCANCEL`, `IDABORT`) zu implementieren und um bei `IDOK` zu prüfen, ob das Fenster geschlossen werden darf (Methode `CanClose`).

Wenn ein abgeleitetes Objekt auf ein hier nicht behandeltes *Kommando* reagieren will, sollte diese Methode überschrieben werden.

Tipp: Am Ende der überschriebenen Methode sollte diese Methode aufgerufen werden, um das Standardverhalten zum Schließen von Dialogfenstern auszulösen!

II.3.3 Bewertung

Metrik		Kennung (min, max)	Wert
Klasse			
	‚Lines Of Code' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	64
	‚LOC of Implementation'*	LOCI	37
	‚LOC of Declaration'*	LOCD	27
	‚Number Of Attributes'	NOA (0, 30)	1
	‚Number Of Operations'	NOO (0, 50)	2
	‚Number Of Members' Attribute + Methoden	NOM = NOA + NOO	3
	‚Number Of Constructors'	NOCON (0, 5)	1
	‚Number Of Overridden Methods'	NOOM (0, 10)	1
	‚Percentage of Private Members'	PPrivM	0
	‚Percentage of Protected Members'	PProtM (0, 10)	40
	‚Percentage of Public Members'	PPubM	60
	‚Weighted Methods Per Class'	WMPC1 (0, 30)	
Attribute			
	‚Attribute Complexity'	AC	9
Methoden			
	‚Maximum Number Of Parameters'	MNOP (0, 4)	4
	‚Cyclomatic Complexity'	CC	
Kommentare			
	‚Number Of Comments'*	NOC	31
	‚True Comment Ratio'	TCR (5, 400)	54

Tabelle 4 „ausgewählte Metriken der Klasse `TModalDlg`" (Quelle: Together® ‚Version 6.0)

* Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

II.4 Klasse TModelessDlg

Deklaration : SWINTRAC.H

Implementation : DLG_TPL.CPP

Es gibt keine abstrakten Methoden, d.h. Objekte von `TModelessDlg` könnten erzeugt werden. Das ist jedoch nicht empfehlenswert, weil nur das Schließungsverhalten implementiert ist.

II.4.1 Attribute

▶ **`DLGPROC m_Handler`** **PROTECTED**

ist der Zeiger auf die `CALLBACK`-Funktion `TBasicWindow::EventHandler`; Muss beim Laden der Ressourcen als Parameter übergeben werden (aus [2]). wird im Konstruktor initialisiert (Win16: Muss zuvor in die Instanz der Anwendung aufgenommen und am Ende wieder entfernt werden)

▶ **`TModelessDlg **m_Reference`** **PROTECTED**

Adresse auf die Variable, in der das Dialogfenster gespeichert wird; Wenn das Fenster geschlossen wird, wird das -objekt freigegeben und die Variable wird `0` gesetzt. Indem man prüft, ob die Variable == `0` ist, kann man ermitteln, ob das Fenster [nicht] sichtbar ist.

II.4.2 Methoden

▶ **`TModelessDlg( const char*, TModelessDlg** )`** **PUBLIC**

Im Konstruktor werden alle Attribute initialisiert (siehe **II.4.1 Attribute**). Der erste Parameter ist der Name der Ressource für das anzuzeigende Dialogfenster. Der zweite Parameter wird in `m_Reference` gespeichert.

▶ **`virtual ~TModelessDlg( void )`** **PUBLIC**

Wenn das Fenster beim Aufruf des Destruktors sichtbar ist, wird das Fenster zuerst geschlossen. Das ist die einzige Möglichkeit das Fenster selbst zu schließen.

Der Zeiger auf die `CALLBACK`-Funktion wird wieder freigegeben (nur bei Win16) und die Registrierung des Fensters wird aufgehoben (siehe **Dokumentation des Istzustands**). Außerdem wird die Variable, in der das Dialogfenster gespeichert wird, `0` gesetzt. Indem man prüft, ob die Variable == `0` ist, kann man ermitteln, ob das Fenster [nicht] sichtbar ist.

▶ **`BOOL Initialize( HINSTANCE= 0 )`** **PUBLIC**

erzeugt das Dialogfenster und zeigt es an; Der Rückgabewert gibt Auskunft, ob das Fenster angezeigt werden konnte, im Fehlerfall wird auch eine Meldung ausgegeben. Hier kann auch eine andere Instanz, als `::ModuleInstance` angegeben werden, wird der Parameter weggelassen oder 0 übergeben, wird `::ModuleInstance` verwendet.

▶ **`virtual void Dlg_OnCommand( HWND, int, HWND, UINT )`** **PROTECTED**

wird überschrieben, um das Standardverhalten zum Schließen des Dialogfensters (`IDCANCEL`, `IDABORT`) zu nutzen und um bei `IDOK` zu prüfen, ob das Fenster geschlossen werden darf (Methode `CanClose`).

Wenn ein abgeleitetes Objekt auf ein hier nicht behandeltes *Kommando* reagieren will, sollte diese Methode überschrieben werden.

Tipp: Am Ende der überschriebenen Methode sollte diese Methode aufgerufen werden, um das Schließungsverhalten zu implementieren!

II.4.3 Bewertung

Metrik		Kennung (min, max)	Wert
Klasse			
	‚Lines Of Code' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	72
	‚LOC of Implementation'*	LOCI	45
	‚LOC of Declaration'*	LOCD	27
	‚Number Of Attributes'	NOA (0, 30)	1
	‚Number Of Operations'	NOO (0, 50)	2
	‚Number Of Members' Attribute + Methoden	NOM = NOA + NOO	3
	‚Number Of Constructors'	NOCON (0, 5)	1
	‚Number Of Overridden Methods'	NOOM (0, 10)	1
	‚Percentage of Private Members'	PPrivM	0
	‚Percentage of Protected Members'	PProtM (0, 10)	40
	‚Percentage of Public Members'	PPubM	60
	‚Weighted Methods Per Class'	WMPC1 (0, 30)	
Attribute			
	‚Attribute Complexity'	AC	9
Methoden			
	‚Maximum Number Of Parameters'	MNOP (0, 4)	4
	‚Cyclomatic Complexity'	CC	
Kommentare			
	‚Number Of Comments'*	NOC	30
	‚True Comment Ratio'	TCR (5, 400)	52

Tabelle 5 „ausgewählte Metriken der Klasse `TModelessDlg`" (Quelle: Together® ‚Version 6.0)

* Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

II.5 Klasse TBasicMDIWindow

Deklaration : SWINTRAC.H

Implementation : DLG_TPL.CPP

Es gibt keine abstrakten Methoden, d.h. Objekte von `TBasicMDIWindow` könnten erzeugt werden. Das ist jedoch nicht empfehlenswert, weil nur das Schließungsverhalten implementiert ist. Ein würde ein leeres Fenster erscheinen.

II.5.1 Attribute

► **`char m_Title[256]`** **PROTECTED**

enthält die Zeichenkette, die mit Methode `SetTitle` in der Titelleiste angezeigt werden kann

II.5.2 Methoden

► **`TBasicMDIWindow( void )`** **PUBLIC**

initialisiert `m_Title` als Leerstring; zum Anzeigen Methode `Show` aufrufen

▶ **virtual ~TBasicMDIWindow (void)** **PUBLIC**
Wird beim Schließen des Fensters automatisch aufgerufen. Implementiert das Unregistrierungsverhalten für das Fenster

▶ **LRESULT Show (HWND, const int= -1, const int= -1, const int= -1, const int= -1)** **PUBLIC**
zeigt das Fenster an; der erste Parameter enthält das Elternfenster; die restlichen Parameter sind optional und geben die Koordinaten (x, y, Breite und Höhe) an

▶ **virtual BOOL CanOpen (void)** **PROTECTED**
Wird beim Anzeigen des Fensters aufgerufen. Wird hier `FALSE` zurückgeben, wird der Anzeigevorgang abgebrochen. hier nur Rückgabe von `TRUE` (Fenster kann angezeigt werden)

▶ **virtual BOOL CanClose (void)** **PROTECTED**
Wird beim Schließen des Fensters aufgerufen. Wird hier `FALSE` zurückgegeben, kann das Fenster nicht geschlossen werden, z.B. weil noch nicht alle Daten gespeichert wurden. hier nur Rückgabe von `TRUE` (Fenster kann geschlossen werden)

▶ **virtual HBRUSH GetBkColor (void)** **PROTECTED**
kann überschrieben werden, um die Hintergrundfarbe zu ändern; hier nur Rückgabe von `COLOR_WINDOW + 1`

▶ **virtual HICON GetIcon (void)** **PROTECTED**
kann überschrieben werden, um das Ikon der Titelleiste zu ändern; hier nur Rückgabe von `0`

▶ **virtual HMENU GetMenu (HWND)** **PROTECTED**
kann überschrieben werden, um ein Kontextmenü zu erstellen; im Parameter wird das Fensterhandle (identisch zu `GetHandle()`) übergeben; hier nur Rückgabe von `0`

▶ **virtual LPCSTR ClassName (void)** **PROTECTED**
kann überschrieben werden, um einen anderen Klassennamen zurückzugeben; hier nur konstanter Rückgabewert

▶ **virtual BOOL SetTitle (void)** **PROTECTED**
zeigt `m_Title` als Bezeichnung der Titelleiste an

▶ **LRESULT CALLBACK EventHandler (HWND, UINT, WPARAM, LPARAM)** **STATIC PROTECTED**
empfängt die Windowsbotschaften aller (von `TBasicMDIWindow` abgeleiteten) Fenster; behandet `WM_CREATE` und `WM_NCDESTROY` und leitet alle übrigen Botschaften, über Methode `TBasicWindow::EventHandler`, an das entsprechende Objekt, Methode `OnEvent`, weiter – Identifikation über das *Fensterhandle* im ersten Parameter;

▶ **virtual LRESULT OnEvent (HWND, UINT, WPARAM, LPARAM)** **PROTECTED**
überschrieben um ausgewählte Botschaften an die folgenden Methoden weiterzuleiten

▶ **virtual void OnKeyDown (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_KEYDOWN` (Taste wird gedrückt, noch nicht losgelassen) aufgerufen; hier leer implementiert

▶ **virtual void OnLButtonDown (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_LBUTTONDOWN` (linke Maustaste gedrückt, noch nicht losgelassen) aufgerufen; hier leer implementiert

▶ **virtual void OnLButtonUp (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_LBUTTONUP` (linke Maustaste losgelassen) aufgerufen; hier leer implementiert

▶ **virtual void OnRButtonDown (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_RBUTTONDOWN` (rechts Maustaste gedrückt, noch nicht losgelassen) aufgerufen; hier leer implementiert

▶ **virtual void OnMouseMove (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_MOUSEMOVE` (Mauscursor wurde bewegt) aufgerufen; hier leer implementiert

▶ **virtual LRESULT OnCommand (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_COMMAND` (Nutzerdefinierte Ereignisse) aufgerufen; hier nur Rückgabe von `0`

▶ **virtual void OnCreate (void)** **PROTECTED**
wird am Ende von `WM_CREATE` (Fenster wurde erzeugt) aufgerufen, wenn Methode `CanOpen` `TRUE` zurückgeben wurde; hier leer implementiert

▶ **virtual void OnPaint (void)** **PROTECTED**
wird bei `WM_PAINT` (Fenster soll gezeichnet werden) aufgerufen,; hier wird das Fenster vorbereitet und Methode `DoPaint` aufgerufen

▶ **virtual void DoPaint(HDC, PAINTSTRUCT*)** **PROTECTED**
wird zum zeichnen des Canvas (erster Parameter) aufgerufen; der zweite Parameter gibt Informationen über die zu zeichnende Fläche

▶ **virtual void OnFocus (void)** **PROTECTED**
wird bei `WM_SETFOCUS` (Fenster wurde fokussiert) aufgerufen; hier leer implementiert

▶ **virtual void OnSize (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_SIZE` (Fenstergröße wurde geändert) aufgerufen; hier leer implementiert

▶ **virtual void OnTimer (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_TIMER` (Timerereignis ist aufgetreten) aufgerufen; hier leer implementiert

▶ **virtual void OnRenderFormat (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_RENDERFORMAT` aufgerufen; hier leer implementiert

▶ **virtual void OnRenderAllFormats (WPARAM, LPARAM)** **PROTECTED**
wird bei `WM_RENDERALLFORMATS` aufgerufen; hier leer implementiert

II.5.3 Bewertung

Metrik		Kennung (min, max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	253
	‚**LOC** of **I**mplementation'*	LOCI	174
	‚**LOC** of **D**eclaration'*	LOCD	79
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	1
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	24
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	25
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	1
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	4
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	85
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	11
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	3
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	**5**
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'*	NOC	
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	29

Tabelle 6 „ausgewählte Metriken der Klasse `TBasicMDIWindow`" (Quelle: Together® ‚Version 6.0)

* Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

II.6 Klasse THotKey

Deklaration : SWINTRAC.H

Implementation : DLG_TPL.CPP

Es gibt keine abstrakten Methoden, d.h. Objekte von `THotKey` können erzeugt werden.

Win32: `THotKey` besitzt unter Win16 keine Funktionalität.

Tipp: Statt `THotKey` sollte Methode `AddHotKey( …, …, …, …, … )` oder `LoadHotKeys( … )` von `TBasicWindow` verwendet waren, weil die Tastenkombinationen dort einfacher verwaltet werden können.

II.6.1 Attribute

▶ **`int m_Cmd`** **PUBLIC**

Kommando-Id, die beim Drücken der Tastenkombination ausgelöst werden soll, wenn das Fenster aktiv ist; wenn die Tastenkombination mit einem, von `TBasicWindow` abgeleiteten, Fenster verknüpft ist; wird eine entsprechende `WM_COMMAND`-Botschaft an das Fenster verschickt

▶ **HWND m_Window** **PRIVATE**

Handle zu dem Fenster, das die `WM_HOTKEY`-Botschaft oder das *Kommando* empfangen soll

Win32: unter Win16 stets `0`

▶ **ATOM m_Atom** **PRIVATE**

eindeutiger Bezeichner, der für die Registrierung der Tastenkombination benötigt wird

Win32: unter Win16 stets `0`

▶ **BOOL m_Registered** **PRIVATE**

gibt an, ob die Tastenkombination erfolgreich registriert wurde

Win32: unter Win16 stets `FALSE`

▶ **LPARAM m_KeyCode** **PRIVATE**

enthält den KeyCode, der für diese Tastenkombination berechnet und registriert wurde

Win32: unter Win16 stets `0`

II.6.2 Methoden

▶ **THotKey(HWND, const char, const BOOL, const BOOL, const BOOL, const BOOL)** **PUBLIC**

registriert die (in den letzten fünf Parametern) angegebene Tastenkombination für das Fenster, dessen *Handle* im ersten Parameter angegeben wurde; Der zweite Parameter bezeichnet eine "normale" Taste – hier sollten nur „`VK_`"-Konstanten angegeben werden. Die folgenden Parameter bezeichnen (in dieser Reihenfolge), ob die [SHIFT], [CTRL], [ALT] und [<WINDOWSTASTE>].gedrückt werden sollen. Im Erfolgsfall wird `m_Registered` `TRUE` gesetzt, sonst `FALSE`.

Win32: Weil diese Art Tastenkombination nur unter Win32 funktioniert, ist dieser Konstruktor unter Win16 leer implementiert, dort wird `m_Registered` `FALSE` gesetzt – es werden keine Ressourcen belegt.

Beim Drücken der Tastenkombination wird, wenn `m_Cmd != -1` ist, ein *Kommando* ans Fenster gesendet – wenn dieses aktiviert ist. Wenn das Fenster inaktiv ist, oder die angegebene *Kommando*-Id `-1` ist, wird eine `WM_HOTKEY`-Botschaft mit `lParam=<KeyCode>` gesendet. Um zu ermitteln ob die Tastenkombination gedrückt wurde die dieses Objekt registriert hat, kann man Methode `IsEqual( … )` benutzen wo `lParam` als Parameter angegeben wird – bei Übereinstimmung wird dort `TRUE` zurückgegeben, sonst `FALSE`.

▶ **THotKey(THotKey&)** **PRIVATE**

Der Copy-Konstruktor ist leer implementiert, um ungewünschte Kopienbildung zu verhindern.

▶ **virtual ~THotKey(void)** **PUBLIC**

Der Destruktor gibt alle unter Win32 belegten Ressourcen wieder frei. Ist unter Win16 leer implementiert.

▶ **BOOL IsEqual(LPARAM)** **PUBLIC**

vergleicht den, im Parameter übergebenen, KeyCode mit dem KeyCode, der für diese Tastenkombination registriert wurde; Wenn die Tastenkombination nicht registriert wurde, wird FALSE zurückgegeben.

Win32: Unter Win16 wird stets `FALSE` zurückgegeben.

► **`BOOL IsRegistered( void )`** **PUBLIC**

gibt `m_Registered` (Tastenkombination erfolgreich registriert) zurück

Win32: Unter Win16 wird stets `FALSE` zurückgegeben.

► **`static UINT CalcModifier( const BOOL, const BOOL, const BOOL, const BOOL )`** **PUBLIC**

berechnet den ModifierCode für die [SHIFT], [CTRL], [ALT] und [<WINDOWSTASTE>], die in dieser Reihenfolge in den Parametern übergeben wurde

Win32: Unter Win16 wird stets `0` zurückgegeben.

► **`static LPARAM CalcKeyCode( const char, const BOOL, const BOOL, const BOOL )`** **PUBLIC**

berechnet den KeyCode für, die in den Parametern übergebene, Tastenkombination; Der erste Parameter bezeichnet eine "normale" Taste – hier sollten nur „`VK_`"-Konstanten angegeben werden. Die letzten Parameter bezeichnen (in dieser Reihenfolge), ob die [SHIFT], [CTRL], [ALT] und [<WINDOWSTASTE>].gedrückt werden sollen.

Win32: Unter Win16 wird stets `0` zurückgegeben.

II.6.3 Bewertung

Metrik		Kennung (min, max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	136
	‚**LOC** of **I**mplementation'*	LOCI	88
	‚**LOC** of **D**eclaration'*	LOCD	48
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	5
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	4
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	9
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	2
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	0
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	42
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	52
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	37
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	**6**
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'*	NOC	55
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	60

Tabelle 7 „ausgewählte Metriken der Klasse `THotKey`" (Quelle: Together® ‚Version 6.0)

* Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

III Attribute

► `extern HINSTANCE hModuleInstance` **GLOBAL**

um auf das in M_MAIN.CPP definierte Attribut auch in DLG_TPL.CPP zugreifen zu können; ist die eindeutige Kennung der Instanz der Anwendung

IV Methoden

► `BOOL SetActivateHotKey( HWND, const char, const BOOL, const BOOL, const BOOL, const BOOL )` **GLOBAL**

registriert die, in den Parametern angegebene, Tastenkombination für das im ersten Parameter angegebene *Fensterhandle*; Der zweite Parameter bezeichnet eine "normale" Taste – hier sollten nur „VK_"-Konstanten angegeben werden. Die folgenden Parameter bezeichnen (in dieser Reihenfolge), ob die [SHIFT], [CTRL], [ALT] und [<erweitert>] gedrückt werden sollen – wobei niemand weiß, was <erweitert> bedeutet. Der Rückgabewert gibt an, ob die Tastenkombination erfolgreich registriert wurde (`TRUE`) oder ob Fehler auftraten (`FALSE`).

Wenn die Tastenkombination gedrückt wird und das Fenster inaktiv ist, wird das Fenster aktiviert. Wenn das Fenster bereits aktiv ist, wird eine `WM_SYSCOMMAND`-Botschaft mit `wParam=SC_HOTKEY` gesendet.

Win16: Diese Art Tastenkombination funktioniert unter Win16 und Win32.

ACHTUNG: Von dieser Art Tastenkombination kann nur max. eine definiert werden – wenn bereits eine solche Tastenkombination definiert ist, wird diese überschrieben.

V Anhang

V.1 Verwandte Dokumente

[1] „Layoutkonventionen und Steuerelemente", Version 1.0 von Thomas Kullmann und Günther Reinecker
[2] „Borland 5.02 Hilfe"
[3] „Microsoft Visual C++®; Version 6.0; Quellen der Microsoft Foundation Class"

V.2 Index

V.3 Tabellen

V.4 Abbildungen