ISA Dialog Manager

C++-Schnittstelle

A.06.03.c

In diesem Handbuch ist die Schnittstelle des ISA Dialog Managers für in C++ entwickelte Anwendungen dargestellt. Es beschreibt die Klassen zur Anbindung des IDM und deren Methoden. Die Funktionsweise von Code-Generator und Code-Mischer wird erklärt.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Deutschland

Darstellungskonventionen

DM wird in diesem Handbuch synonym zu "Dialog Manager" verwendet.

Die Bezeichnung UNIX schließt generell alle unterstützten UNIX-Derivate ein - außer in den explizit angegebenen Fällen.

Dort wo für geläufige englische Fachbegriffe keine gängigen deutschen Übersetzungen existieren, wird zur Vermeidung von Unklarheiten der englische Begriff verwendet.

< >	muss durch einen entsprechenden Wert ersetzt werden
color	Schlüsselwort ("keyword")
.bgc	Attribut
{ }	optional (0 oder einmal)
[ ]	optional (0 oder n-mal)
<A> | <B>	entweder <A> oder <B>

Beschreibungsmodus

Alle Schlüsselwörter sind fett und unterstrichen, z.B.

variable **integer** **function**

Indizierung von Attributen

Syntax für indizierte Attribute:

[I]

[I,J] bzw. [row,column]

Identifikatoren

Identifikatoren müssen mit einem Großbuchstaben oder einem "Unterstrich" ('_') beginnen. Die weiteren Zeichen können Groß-, Kleinbuchstaben, Zahlen oder Unterstriche sein.

Der Bindestrich ('-') ist für die Benennung von Identifikatoren als Zeichen ***nicht*** zugelassen!

Die maximale Länge eines Identifikators beträgt *31* Zeichen.

Beschreibung der zugelassenen Identifikatoren in Backus-Naur-Form

<Identifikator> ::= <erstes Zeichen>{<Zeichen>}

<erstes Zeichen> ::= _ | <Großbuchstabe>

<Zeichen> ::= _ | <Kleinbuchstabe> | <Großbuchstabe> | <Ziffer>

<Ziffer> ::= 1 | 2 | 3 | … 9 | 0

<Kleinbuchstabe> ::= a | b | c | … x | y | z

<Großbuchstabe> ::= A | B | C | … X | Y | Z

Inhalt

Darstellungskonventionen 3

Inhalt 5

1 Einführung 7

2 Prinzip, Umfang und Arbeitsweise 8

2.1 Prinzipien 8

2.2 Umfang der C++-Schnittstelle 10

2.3 Arbeitsweise 11

2.3.1 Record mit Funktionen anlegen 11

2.3.2 Anpassungen der Anwendung, Implementierung der Klasse 11

2.3.3 C++-Datei erzeugen 13

2.3.4 Kompilieren und Ausführen 13

3 Codegenerator 14

4 Code-Mischer 16

5 Anbindung 19

6 Klassenhierarchie 20

6.1 Klasse DM_Interface 20

6.1.1 getValues() 21

6.1.2 setValues() 21

6.1.3 resetValues() 22

6.1.4 freeValues() 23

6.2 Klasse DM_Object 24

6.2.1 DM_Object() (Konstruktor) 24

6.2.2 id() 24

6.2.3 id() 25

6.2.4 create() 26

6.2.5 destroy() 26

6.2.6 parsepath() 27

6.2.7 query() 27

6.2.8 start() 28

6.2.9 stop() 29

6.2.10 control() 29
6.2.11 getContent() 30
6.2.12 get() 31
6.2.13 get…() 32
6.2.14 getVector() 33
6.2.15 setContent() 34
6.2.16 set() 35
6.2.17 set…() 36
6.2.18 setVector() 38
6.2.19 reset() 39
6.2.20 call() 39
6.2.21 callMethod() 40
6.2.22 sendevent() 41
6.2.23 getToolkitData() 42
6.2.24 setToolkitData() 43
6.3 Klasse DM_Record 43
6.3.1 DM_Record() (Konstruktor) 43
6.3.2 id() 44
6.3.3 getRecInfo() 45
6.3.4 connect() 45
6.3.5 disconnect() 45
6.3.6 get() 46
6.3.7 put() 47
6.3.8 bind() 47
6.3.9 call() 48
7 Schnittstellen-Funktionen 50
7.1 DM_BindClass 50
7.2 DM_ConnectClass 51
8 Attribute 52
8.1 connect 52
8.2 shadow 52
Index 55

1 Einführung

In diesem Handbuch wird die Anwendungsschnittstelle (API) zwischen dem Dialog Manager (DM) und einer in C++ geschriebenen Anwendung beschrieben.

Gerichtet ist dieses Handbuch an Programmierer, die mit der DM-Entwicklungsumgebung, der C-Schnittstelle des DM und der Programmiersprache C++ vertraut sind. Voraussetzung sind außerdem gute Kenntnisse des verwendeten Betriebssystems, dessen Werkzeuge und der eingesetzten Compiler.

2 Prinzip, Umfang und Arbeitsweise

Die C++-Schnittstelle ist das Verbindungsglied zwischen einer Anwendung mit einer objekt-orientierten Modellierung und der objektorientierten Dialogschicht. Sie ist kein Ersatz der C-Schnitt-stelle sondern eine Ergänzung, die eine vereinfachte und objektorientierte Programmierung erlaubt.

2.1 Prinzipien

Jedes Record-Objekt eines Dialoges/Moduls, das Funktionen beinhaltet entspricht einer C++-Klasse. Der IDM selbst beinhaltet keinen C++ Code sondern stellt einen Generator bereit, der für diese Records C++-Klassencode generiert. Der Anwendungsprogrammierer programmiert dann nur noch die objektbasierten Funkionen als C++-Methoden aus. Der generierte Code definiert dabei für die Records jeweils eine Klasse mit gleichen Namen, abgeleitet von der vordefinierten DM_Record-Klasse oder, falls es sich um ein von einem Modell abgeleiteten Rekord handelt, entsprechend der Vererbungshierarchie. Des weiteren enthält der Code eine statische Funktion die das Bindeglied zum IDM ist und die notwendigen Aktionen zum Erzeugen und Löschen einer Instanz, der Transferierung von Strukturwerten und dem Aufruf von Methoden bewerkstelligt. Der Anwendungsprogrammierer muss diese Bindefunktion nur dem Record bekannt machen.

Dialog

```
modelrecord MArtikel
{
  integer Anzahl;
  function boolean Bestellen(string Anschrift);
}
```

C++

```
class MArtikel: DM_Record
{
  /* ... Code zur Anbindung an den IDM */
  DM_Integer Anzahl;
  DM_Boolean Bestellen(DM_String Anschrift);
};
```

Abbildung 1: *Methodendefinition im IDM und der Klassencode in C++*

Währrend der Ausführung eines solchen Dialoges mit angebundener C++-Klasse erzeugt dann der IDM automatisch die Instanzen aus der generierten C++-Klasse wenn der Record oder das Modell instanziert wird. Im Gegensatz zu C++ besteht im IDM keine Unterscheidung zwischen der Klas-sendeklaration und einer Klasseninstanz - der IDM kennt nur Objekte, ein Modell ist dabei genauso ein Objekt. Somit wird, wenn nicht explizit abgeschaltet, auch für ein Modell auf der Dialogseite eine Klasseninstanz auf der C++-Seite angelegt.

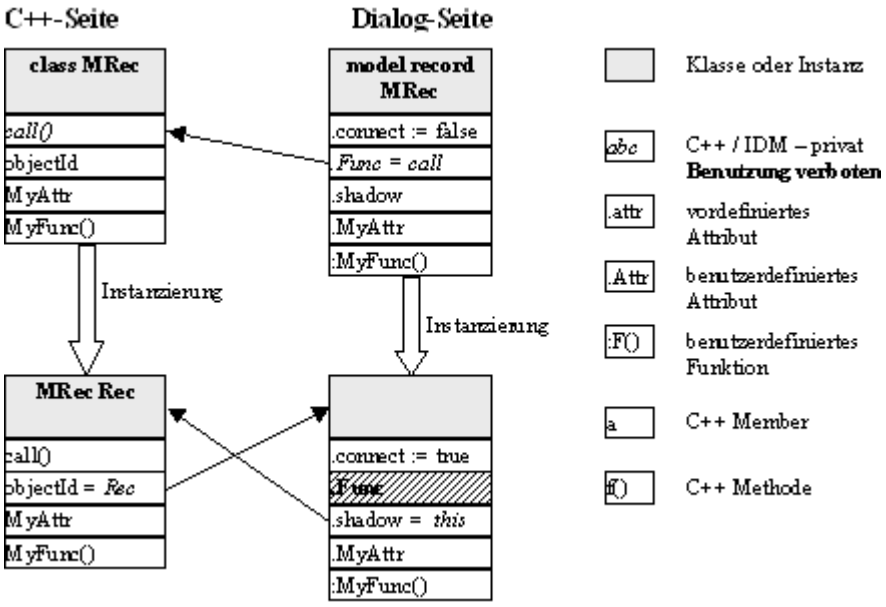

Abbildung 2: *Zusammenhang zwischen Dialog-Objekt und Klasseninstanz*

Das obige Bild macht den Zusammenhang zwischen einer Klasse, dem Modell auf Dialogseite und den Instanzen deutlich. Der IDM merkt sich also am Modell den Funktionspointer, die Instanz des Modells merkt sich den this-Pointer der C++-Klasseninstanz und die C++-Klasseninstanz merkt sich die ID des instanzierten Modells.

Ein Methodenaufruf einer Record-Funktion wird auf die C++-Seite weitergeleitet. Dabei wird vor dem eigentlichen Aufruf der Inhalt der Records an die C++-Instanz übertragen und nach dem Aufruf wieder zum IDM zurücktransferiert. Das Übertragen der eigentlichen Werte geschieht auf ähnliche Weise wie bei der C-Schnittstelle, wenn Funktionen mit Records benutzt werden. Eine Hilfsstruktur pro Klasse enthält alle notwendigen Abbildungsinformationen um vom C++-Strukturelement zum Attribut auf der Dialogseite zu gelangen und umgekehrt. Natürlich kann man während der Entwicklung auch Simulationsregeln für die Funktionen schreiben, um auch bei nichtangebundenem C++-Code noch weiterentwicklen zu können.

Den Vorgang der Instanzierung einer C++-Klasse, wird beim Starten des Dialoges (exakt beim Aufruf von DM_StartDialog() oder der Builtin-Regelfunktion start()) bzw. bei der Instanzierung eines Modells gemacht. Dieser Vorgang kann auch unterbunden und selbst gesteuert werden (Siehe dazu das .connect-Attribut).

Nochmal zurück zu dem vom IDM generierten C++-Code. Um die größtmögliche Flexibilität für den Anwendungsprogrammierer zu bewahren (z.B.: Verteilung auf mehrere Dateien, eigene Superklassen, eigene Members, usw.) wird der C++-Code nicht einfach nur generiert sondern der generierte Code kann mit bestehendem Code der Anwendung gemischt werden.

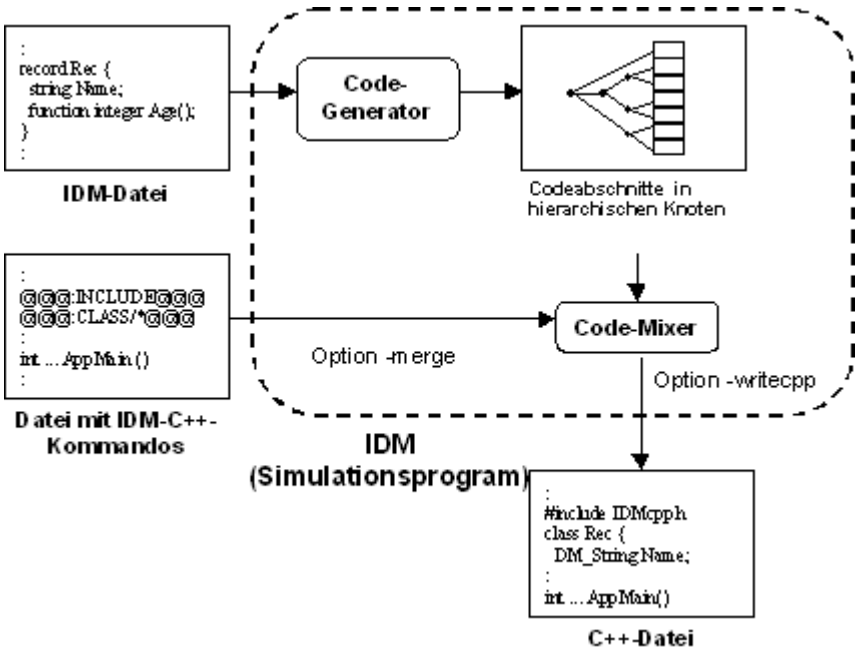

Abbildung 3: *Der Weg bis zur compilierbaren C++-Datei*

Die Anwendungsdatei enthält zu dessem Zweck noch zusätzliche „Kommandos“ in Form von @@@...@@@ die vom Code-Mischer interpretiert werden. Dieser fügt den von der Code-generierung erhaltenen und gegliederten Code dann anstelle des Kommandos ein.

Diese so erhaltene C++-Datei kann nun compiliert und gelinkt werden, um eine IDM-Anwendung mit Verwendung der C++-Schnittstelle zu erhalten.

Die C++-Schnittstelle unterstützt den C++-Programmierer außerdem im Umgang mit Attributen und Objekten durch die Klasse DM_Object. Diese Klasse kann sich die DM_ID eines Objektes merken und stellt Methoden zum einfachen Zugriff auf Attribute zur Verfügung. Merkmal der meisten von der C++-Schnittstelle definierten Methoden ist außerdem, dass sie Fehler nicht über Rückgabewerte sondern über auslösen einer Exception weitergeben.

2.2 Umfang der C++-Schnittstelle

1. Der IDM erlaubt die Definition von Funktionen als Kinder von Records. Diese sind dann aus der Regelsprache als Methoden aufrufbar und können auch durch Simulationsregeln simuliert werden. Die Objektklasse ***record*** ist um die Attribute *.connect* und *.shadow* erweitert.
2. Die Header-Datei IDMcpp.h wird im include-Verzeichnis zur Verfügung gestellt.
   - » Included automatisch IDMuser.h
   - » Definiert die Klassen DM_Interface, DM_Object, DM_Record
   - » Beinhaltet statische Methoden
   - » Objektbasierte Methoden zur Anwendung auf ein DM-Objekt
   - » Methodenstubs für die Schnittstelle zwischen C++-Klassen und DM-Objekten

3. Codegenerator und Mischer, gesteuert über die Optionen –writecpp und –merge des IDM-Programms.
4. Beispiel „bodysim“ im demo/cppif-Verzeichnis.

2.3 Arbeitsweise

Dieser Abschnitt soll einen schnellen Einstieg in die C++-Schnittstelle erlauben und dabei grundlegende Arbeitsweisen erklären.

Ein Lieferant für Sechskantschrauben will seinen Verkaufsprozess durch Direktbestellmöglichkeiten am Rechner optimieren. Das bisher in C++ geschriebenes Lagersystem bietet dabei die Klasse „Artikel“ und die nicht näher beschriebene Klasse „Bestellung“ an.

```
class Artikel
{
  int Menge;
  int Preis;
  char *Bezeichnung;
  int Hinzufuegen(int Menge);
  int Wegnehmen(int Menge);
}
```

2.3.1 Record mit Funktionen anlegen

Zu jedem Artikel soll zusätzlich zum Preis und Bezeichnung auch ein Bild auf dem Bildschirm erscheinen. Zum Datenaustausch zwischen Anwendung und Benutzeroberfläche definieren wir also den ***Record*** „MArtikel“ mit den notwendigen Erweiterungen:

```
model record MArtikel
{
  .connect false;  // Modell erhält nicht automatisch eine C++-Instanz
  string ABezeichnung;
  integer APreis;
  string Bilddatei;
  function boolean Verkaufe(int Anzahl, object BestellId);
}
```

2.3.2 Anpassungen der Anwendung, Implementierung der Klasse

Unser Anwendungsprogramm wird erweitert um die Anbindung der Klasse „MArtikel“ und die Datei wird z.B. als „direkt.dcc“ abgespeichert.

```
#include "lagerverwaltung.h"
@@@INCLUDE@@@
Bestellung LagerBestellung;
```

```
DM_Boolean inEuro = DM_FALSE;

@@@:CLASS/MArtikel!@@@, Artikel  // eigene Superklasse
{
  @@@MEMBER/*@@@     // benutzerdefinierte Attribute einmischen
  int MindestMenge;  // private Erweiterung der C++-Klasse
  string Lagerort;   // zur Koordination der Artikelnachbestellung

  MArtikel {
    // eigener Konstruktor
    // Werte für das GUI bereitstellen
    MindestMenge = Preis < 10 ? 20 : 5;
    APreis = inEuro ? Preis / 2 : Preis;
    ABezeichnung = Bezeichnung;
  }
  @@@METHOD/Verkaufe!@@@
  {
    DM_Object Bestellung;
    DM_Value argv[2];

    if (Menge < Anzahl)
      return DM_FALSE;  // nicht mehr genügend Artikel verfügbar
    if (Menge - Anzahl < MindestMenge) {
      // falls nötig, gleich die Mindestmenge nachbestellen
      LagerBestellung.Hinzufuegen(this, MindestMenge - Menge - Anzahl);
    }

    // Artikel in Bestellliste aufnehmen
    // indem wir eine Methode auf der Dialogseite aufrufen
    Bestellung.id(BestellId);
    argv[0].type = DT_string;
    argv[0].value.string =  Bezeichnung;
    argv[1].type = DT_integer;
    argv[1].value.integer =  Anzahl;
    Bestellung.callMethod(Bestellung.get(AT_label, ":ArtikelHinzufuegen"),
                                          2, argv);
    // Artikel aus dem Lager nehmen
    Wegnehmen(Anzahl);
    return DM_TRUE;
  }

   @@@METHOD/*@@@  // restliche Methoden
}
...
@@@:IMPL/*@@@  // notwendige IDM-Anbindung
...
int DML_c DM_CALLBACK AppMain __2((int, argc), (char **, argv))
```

```
{
  DM_Object dialog;
  DM_Object CbEuro;
  DM_Initialize (&argc, argv, 0);
  if (argc >= 2)
    dlgname = argv[1];

  dialog.id(DM_LoadDialog ("direkt.dlg", 0))
  if (!(dialog.id()))
  {
    DM_TraceMessage("could not load dialog", DMF_LogFile);
    return 1;
  }
  MArtikel::bind(dialog.id(), "MArtikel");  // Klasse anbinden
  CbEuro.id(dialog.parsepath((DM_ID)0, "WnOptionen.CbEuro"));
  inEuro = CbEuro.getBoolean(AT_active);
  dialog.start();
  DM_EventLoop(0);
  return 0;
}
```

2.3.3 C++-Datei erzeugen

Nun kann die eigentliche C++-Datei mittels des Kommandos `idm direkt.dlg –merge direkt.dcc –writecpp direkt.cc` erzeugt werden. Man muss nicht wie oben geschehen, Definition und Implementierung in eine Datei hinein generieren. Es ist durch den Codemischer möglich Definitions- und Implementierungsteil zu trennen oder auch einzelne Klassen in unterschiedlichen Modulen aufzuteilen.

2.3.4 Kompilieren und Ausführen

Zum Kompilieren der Anwendung kann ganz einfach ein Standard-Makefile aus dem Demo-Verzeichnis herangezogen werden.

Der C++-Compiler wie z.B MICROSOFT VISUAL C++ erkennt nun an der Endung „.cc“ automatisch das es sich um eine C++-Datei handelt und übersetzt sie entsprechend. Auf UNIX kann z.B. der g++ (mit Option **-fhandle-exceptions**, oder das in der Datei **MakeDefs** vordefinierte Symbol **CXX** statt **CC** nehmen) verwendet werden.

3 Codegenerator

Beim Aufruf des IDM mit der Option –writecpp in Verbindung mit der Option –merge wird der Codegenerator für den C++-Code aufgerufen. Dieser geht alle Records (unabhängig ob Default, Modell oder Instanz) durch und generiert C++-Code für jeden Record mit Funktionen.

Der Code für eine C++-Klasse besteht dabei aus einem Definitions- und einem Implementierungsteil. Die Attribute des Records erscheinen dabei als Member-Variablen (analog zur Trampolin-Generierung), Funktionen als C++-Methoden im Definitionsteil. Der Implementierungsteil der Klasse beinhaltet Code für die Anbindung der Klasse, einer Methode die Methodenaufrufe von der Regelsprache an die C++-Methode, Code der das Instantiieren und Freigeben vom IDM aus ermöglicht. Ebenfalls im Implementierungsteil befindet sich die für die Übertragung der Attributwerte notwendigen Record-Info-Strukturen.

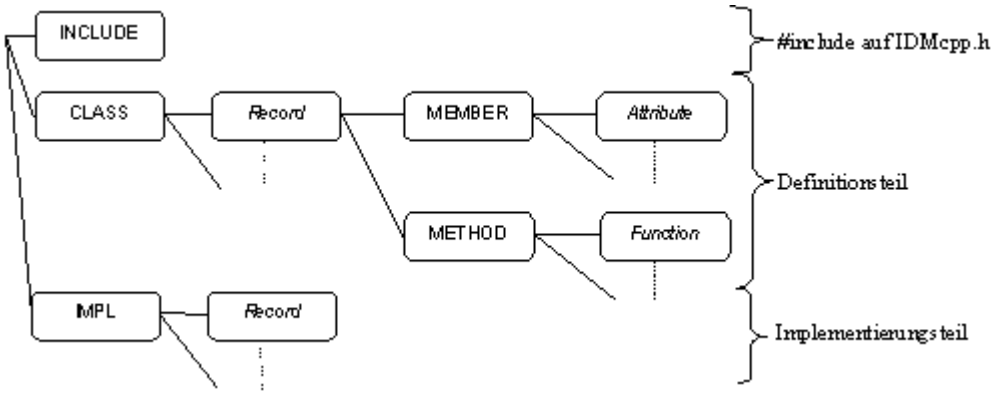

Abbildung 4: *Hierarchische Aufteilung des generierten C++-Codes*

Der generierte Code wird nicht in eine Datei geschrieben sondern nur intern gemerkt. Dabei wird der Code in Textabschnitte aufgeteilt, die mit einer Markierung versehen sind. Diese Markierungen sind wiederum hierarchisch gegliedert, können aber auf der gleichen Hierarchieebene auch mehrfach vorkommen. Solch ein markierter Textabschnitt ist also einfach nur ein Bruchstück aus dem gesamten generierten Code. Zusätzlich ist innerhalb eines Textabschnittes auch noch ein Unterbereich markiert. Ein Textbereich kann jeweils nur einmal herausgeholt werden.

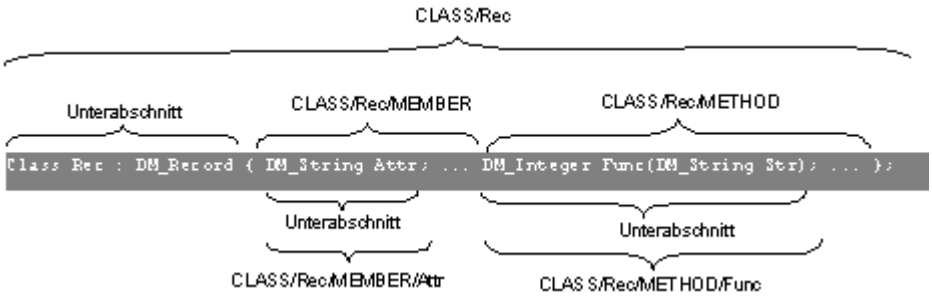

Abbildung 5: *Aufteilung des Definitionsteils in Hierarchieknoten*

Obige Abbildung verdeutlicht dieses Prinzip anhand des aus dem IDM-Record „Rec“ generierten Codes für die dazugehörende C++-Klasse. Der Record selbst besitzt Attribute und Funktionen.

Sinn und Zweck dieser Konstruktion ist folgendes: Der Anwendungsprogrammierer soll selbst bestimmen was aus dem generierten Code in welche Stelle der Datei kommen und was für Ergänzungen noch dazukommen. Dazu schreibt er in seine Ausgangsdatei quasi „Pfadangabe“ hinein, die gewünschte Textabschnitte aus des hierarchisch gegliederten generierten C++-Code herausnimmt. Durch die weitere Qualifizierung eines Unterbereiches können Anforderungen - wie Definition & Implementierung einer Klasse in derselben Datei, Methoden nicht deklarieren sondern gleich implementieren (z.B. ; am Ende weglassen und direkt den Code in { } hinzuschreiben) – realisiert werden.

Die folgende Tabelle gibt nun detailliert Aufschluss über die möglichen Pfade in der Hierarchie und deren Bedeutung. Kursiv gesetzte Wörter weisen dabei auf Symbolische Namen hin. Hier sind die Bezeichner aus der Dialogdatei gemeint.

Knotennamen	Inhalt und Zweck
INCLUDE	Enthält #include-Zeile für die IDMcpp.h-Datei.
CLASS	Umfasst alle Klassendefinitionen.
CLASS/*Record*	Klassendefinition einer speziellen Klasse inklusive Definitionen der Attribute und Methoden. Unterabschnitt: Enthält nur den Klassennamen um eigene Superklassen angeben zu können (ohne {}).
CLASS/*Record*/MEMBER	Enthält die Definition aller Membervariablen (Attribute) einer bestimmten Klasse.
CLASS/*Record*/MEMBER/*Attribute*	Enthält die Definition einer speziellen Membervariablen (Attribut). Unterabschnitt: Ohne ;-Abschluss.
CLASS/*Record*/METHOD	Enthält die Definition aller Methoden (Funktionen) einer bestimmten Klasse.
CLASS/*Record*/METHOD/*Function*	Enthält die Definition einer bestimmten Methode (Funktion). Unterabschnitt: Ohne ;-Abschluss.
IMPL	Enthält den Implementierungsteil aller Klassen (Anbindung an den IDM).
IMPL/*Record*	Enthält die Implementierungsteil einer bestimmten Klasse (Anbindung an den IDM).

4 Code-Mischer

Der Codemischer mischt Teile vom generierten Code in die Anwendungsdatei, sodass eine neue Datei inklusive des C++-Codes entsteht. Dieses Hineinmischen wird gesteuert über Kommandos die in „@@@<*Command*>@@@“ eingeschlossen sind, auch Platzhalter genannt. Der Anwendungsprogrammierer kann damit sehr genau bestimmen welches Codefragment wohin in die Datei eingefügt wird.

Das Kommando kann dabei folgendermaßen aussehen:

```
Command := [ ":" ] Label ( [ "/" Label ]* [ "!" ] [ "/" ] , [ "/" Label ]*
"/*" )
Label := ( "a", ..., "z", "A", ..., "Z" ) [ ( "a", ..., "z", "A", ..., "Z",
        "0", ..., "9", "_" ) ]*
```

Das Kommando dient einfach dazu einen Pfad innerhalb einer Hierarchie zu beschreiben, ähnlich dem Shell-Kommando „cd“. Der Inhalt des Knotens in der Hierarchie (siehe auch Kapitel „Codegenerator“) wird dann anstatt des Kommandos eingefügt. Der Knoten wird dann als schon gelesen markiert und kann nicht mehr weiter ausgelesen werden. Der Inhalt eines Knotens repräsentiert einen Textabschnitt im generierten Code. Oft ist aber auch nur ein Unterabschnitt relevant (z.B. um das „;“ wegzulassen), dann wird durch die Angabe von „!“ dieser markierte Unterbereich eingefügt.

„:“ am Anfang bedeutet von der Wurzel aus, nicht relativ. Damit wird ein Präfix-Pfad gesetzt, der für vor alle folgenden Pfadangaben ohne „:“ verwendet wird. Dabei wird der Pfad ohne den letzten Label als Präfix genommen, außer ein „/“ befindet sich am Ende.

Das „*“ am Ende eines Pfades ist eine Wildcard und sagt das alle noch nicht gelesenen Knoten genommen werden sollen.

Beispiele

:CLASS/A	Man erhält den vollen Textbereich von Knoten A.
:CLASS/*	Man erhält alle Textbereiche von noch nicht gelesenen Knoten unter CLASS
:CLASS/A!	Man erhält nur die markierten Textstellen des Knotens A.

Beispiel

Wir wollen einen Dialog an ein C++-Programm anbinden. Die Dialogdatei „bsp.dlg“ sieht wie folgt aus:

```
dialog D
record Rec
{
  string  A := "Def";
  integer B := 1000;
  function integer F(string S);
```

```
}
on dialog start
{
  print Rec:F("Hello");
  exit();
}
```

Die daraus generierte Klasse heißt „Rec“.

Nun schreiben wir den C++-Teil in eine eigene Datei und speichern Sie als „bsp.dcc“:

```
@@@:INCLUDE@@@
@@@:CLASS/*@@@
@@@:IMPL/*@@@
// nun die Methode F implementieren
DM_Integer Rec::F(DM_String S)
{
  return strlen(A) + strlen(S) + B;
}
...
  // in AppMain die Klasse mit allen Methoden anbinden
  Rec::bind(dialogID, "Rec");
...
```

Zum Mischen des Codes das Kommando `idm bsp.dlg +merge bsp.dcc +writecpp bsp.cc` verwenden. Nun nur noch kompilieren und ausführen!

Wir erweitern unser C++-Programm durch unsere eigene Superklasse und zusätzliche Members:

```
@@@:INCLUDE@@@

class MySuper
{
  int len;
  MySuper() { len = 3; }
};
@@@:CLASS/Rec!/@@@ , MySuper
{
  int MyMember;
  @@@MEMBER/A!@@@
  int MySecondMember;
  @@@MEMBER/*@@@
  @@@METHOD/F!@@@
  {
    return len+strlen(S)+strlen(A)+B;
  }
  @@@METHOD/*@@@
}
@@@:IMPL/*@@@
```

```
...
  // in AppMain die Klasse mit allen Methoden anbinden
  Rec::bind(dialogID, "Rec");
...
```

Die daraus resultierende zusammengemischte Datei „bsp.cc“ sieht etwa so aus:

```
#include "IDMcpp.h"  // eingemischter Code

class MySuper
{
  int len;
  MySuper() { len = 3; }
};
class Rec : DM_Record , MySuper  // eingemischter Code
{
  int MyMember;
  DM_String A;   // eingemischter Code
  int MySecondMember;
  DM_Integer B;  // eingemischter Code

  DM_Integer F(DM_String S)  // eingemischter Code
  {
    return len + strlen(S) + strlen(A) + B;
  }
  // eingemischter Code
}
...  // eingemischter Code
...
  // in AppMain die Klasse mit allen Methoden anbinden
  Rec::bind(dialogID, "Rec");
...
```

5 Anbindung

Zum Kompilieren und Linken kann der auf dem System übliche ANSI C++-Compiler verwendet werden. Beim Linken sind keine zusätzlichen Libraries für den IDM notwendig. Es ist aber zu beachten dass die Weitergabe von Exceptions angeschaltet sein sollte (beim GNU g++ geschieht dies über die Option **–fhandle-exceptions**).

Für UNIX-Architekturen wurde die MakeDefs-Datei um das Symbol CXX erweitert. Dies kann für den Aufruf des C++-Compilers verwendet werden. C++-Compiler erkennen außerdem meist an der Dateiendung ob es sich um eine C oder C++ Datei handelt. Gebräuchliche Endungen für C++ sind hier: **.cc** oder **.C**, **.CC** oder **.cxx**.

6 Klassenhierarchie

Die von der Headerdatei IDMcpp.h zur Verfügung gestellten Klassen lassen sich wie folgt gliedern:

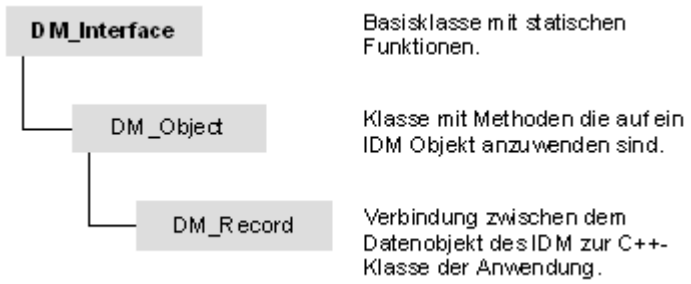

Abbildung 6: *Klassenhierarchie*

In den anschließenden Kapiteln wird nun näher auf die Klassen eingegangen und ihre Funktion erklärt. Die Funktionsdefinition die man in den Erklärungen der Klassenmethoden findet, weist im Gegensatz zur C-Schnittstelle noch Default-Werte auf. Parameter mit Default-Werten sind optional, müssen also nicht angegeben werden. Die Erklärungen der einzelnen Funktionen/Methoden fallen sehr kurz aus, da es sich um Funktionalität handelt, die ausführlich in der C-Schnittstelle beschrieben wird. Aus diesem Grund ist bei Detailfragen die Dokumentation der C-Schnittstelle zu konsultieren.

6.1 Klasse DM_Interface

Diese Klasse stellt einige wenige nicht objektbasierte Funktionen, die auch schon von der C-Schnittstelle bekannt sind, bereit.

Außerdem wird für die Ausnahmebehandlung eine *exception*-Klasse definiert

```
class DM_Exception {} ;
```

Diese Ausnahme wird von fast allen Funktionen benutzt, um einen Fehlerfall weiterzugeben.

Solch eine Ausnahmebehandlung kann, wie in dem folgendem Codefragment anhand der Abfrage auf das *.content*-Attribut einer ***Listbox***, codiert werden:

```
DM_String str;
DM_Value  idx;
DM_Object obj;
...
try {
  obj.id(listboxId);
  idx.type = DT_integer;
  idx.value.integer = 1234;
  str = obj.getString(AT_content, &idx);
}
```

```
catch (DM_Exception e) {
  DM_TraceMessage("Exception aufgetreten – getvalue fehlgeschlagen", 0);
}
...
  class DM_Exception {} ;
```

6.1.1 getValues()

Diese Funktion stellt die Funktion DM_GetMultiValue aus der C-Schnittstelle als statische Methode bereit und dient dazu in einem Aufruf gleich mehrere Attributwerte, auch von verschiedenen Objekten, zu erfragen.

```
void getValues
(
  DM_MultiValue *values,
  DM_UInt        count,
  DM_Options     options = 0
)
```

Parameter

-> DM_MultiValue *values

Liste von Attributen und Objekten deren Wert erfragt werden soll.

-> DM_UInt count

Länge des Feldes, das im Parameter *values* angegeben wurde.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_GetMultiValue

6.1.2 setValues()

Diese Funktion stellt die Funktion DM_SetMultiValue aus der C-Schnittstelle als statische Methode bereit und dient dazu in einem Aufruf gleich mehrere Attributwerte, auch von verschiedenen

Objekten, zu setzen.

```
void setValues
(
  DM_MultiValue *values,
  DM_UInt        count,
  DM_Options     options = 0
)
```

Parameter

-> DM_MultiValue *values

Liste von Attributen, Werten und Objekten, die gesetzt werden sollen.

-> DM_UInt count

Länge des Feldes, das im Parameter *values* angegeben wurde.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_SetMultiValue

6.1.3 resetValues()

Diese Funktion stellt die Funktion DM_ResetMultiValue aus der C-Schnittstelle als statische Methode bereit und dient dazu in einem Aufruf gleich mehrere Attributwerte, auch von verschiedenen Objekten, auf ihre geerbten Werte zurückzusetzen.

```
void resetValues
(
  DM_MultiValue *values,
  DM_UInt        count,
  DM_Options     options = 0
)
```

Parameter

-> DM_MultiValue *values

Liste von Attributen und Objekten, deren Wert zurückgesetzt werden soll.

-> DM_UInt count

Länge des Feldes, das im Parameter *values* angegeben wurde.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_ResetMultiValue

6.1.4 freeValues()

Diese Funktion stellt die Funktion DM_FreeVectorValue aus der C-Schnittstelle als statische Methode bereit und dient dazu einen vom **DM_GetVectorValue()**- bzw. **DM_Object::getVector()**-Aufruf allokierten Speicher freizugeben.

```
void freeValues
(
  DM_VectorValue *vector,
  DM_Options      options = 0
)
```

Parameter

-> DM_VectorValue *vector

Der vom DM allokierte Vektor.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_FreeVectorValue

6.2 Klasse DM_Object

Diese Klasse beherbergt alle allgemeingültigen objektbasierten Methoden auf IDM-Objekte. Dazu zählen die Zugriffsmethoden auf Attribute, Methoden und Funktionen, genauso wie die Handhabung von Objekten (Anlegen, Löschen).

Zu diesem Zweck merkt sich die Klasse die DM_ID des Dialog-Objektes in dem Member

```
DM_ID objectId
```

Eine neu angelegte Instanz dieser Klasse hat noch kein Objekt gesetzt. Dies kann schon während der Definition mittels Konstruktor oder während der Laufzeit über die Methode **DM_Object::id()** geschehen. Nun erst können die weiteren Klassenmethoden erfolgreich angewendet werden.

6.2.1 DM_Object() (Konstruktor)

Neben dem Default-Konstruktor erlaubt dieser die direkte Zuordnung der Klasseninstanz zu einem IDM-Objekt.

```
DM_Object
(
  DM_ID newId = (DM_ID)0
)
```

Parameter

-> DM_ID newId

Objekt-ID auf die dann die weiteren Klassenmethoden wirken.

Rückgabewert

Keiner.

Ausnahmen

Keine.

6.2.2 id()

Diese Funktion setzt die Zuordnung auf ein anderes IDM-Objekt um.

```
void id
(
  DM_ID newId
)
```

Parameter

-> DM_ID newId

Objekt-ID auf die dann die weiteren Klassenmethoden wirken.

Rückgabewert

Keiner.

Ausnahmen

Keine.

Siehe auch

Funktion DM_Object() (Konstruktor)

6.2.3 id()

Diese Funktion liefert die Zuordnung zu einem IDM-Objekt zurück.

```
DM_ID id
(
)
```

Parameter

Keine.

Rückgabewert

ID des zugeordneten IDM-Objektes, kann auch *(DM_ID)0* sein.

Ausnahmen

Keine.

Siehe auch

Funktion DM_Object() (Konstruktor)

DM_Object::id()

6.2.4 create()

Diese Funktion legt ein Kind-Objekt, abgeleitet von einem Modell oder Klasse, an. Das anzulegende Objekt kann ein Modell oder eine Instanz sein.

```
DM_ID create
(
  DM_ID      classOrModel,
  DM_Options options = 0
)
```

Parameter

-> DM_ID classOrModel

Objekt-ID des Modells oder eine Klasse aus *DM_Class**.

-> DM_Options options

Erlaubte Werte sind *0*, *DMF_CreateModel*, *DMF_CreateInvisible*, *DMF_InheritFromModel* oder eine Kombination aus diesen Werten.

Rückgabewert

Eine DM_ID *!= 0* wenn das Objekt angelegt werden konnte.

Ausnahmen

Keine.

Siehe auch

Funktion DM_CreateObject

6.2.5 destroy()

Diese Methode zerstört ein IDM-Objekt mit seinen Kindern und Regeln. Möglich ist auch die Zerstörung eines Modells mit allen seinen Instanzen.

```
void destroy
(
  DM_Options options = 0
)
```

Parameter

-> DM_Options options

Erlaubte Werte sind *0*, *DMF_ForceDestroy*.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_Destroy

6.2.6 parsepath()

Mit Hilfe diese Funktion kann die Dialog-ID zu einem Objekt erfragt werden, von dem nur der Identifikator bzw. der Pfad bekannt ist. Wenn eine Zugehörigkeit (gesetzt über **DM_Object::id()**) zu einem Objekt besteht, wird in dessen Modul gesucht, andernfalls in allen Modulen und Dialogen.

```
DM_ID parsepath
(
  DM_ID     root,
  DM_String path
)
```

Parameter

-> DM_ID root

Objekt-ID der Wurzel ab dem der Pfad gesucht werden soll. Eine *0* bedeutet hier im ganzen Modul suchen.

-> DM_String path

Identifikator oder Pfad.

Rückgabewert

Gesuchte Objekt-ID oder *0* wenn das Objekt nicht gefunden wurde.

Ausnahmen

Keine.

Siehe auch

Funktion DM_DialogPathToID

DM_Object::id

6.2.7 query()

Eine ***Messagebox*** oder Dialogbox wird sichtbar gemacht und gewartet bis der Benutzer diese schließt. Man erhält die Nummer des Buttons der zum Beenden verwendet wurde.

```
DM_Enum query
(
  DM_ID      parentId = 0,
  DM_Options options  = 0
)
```

Parameter

-> DM_ID parentId

Fenster, in dem die ***Messagebox*** erscheinen soll oder *0*.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Nummer des gedrückten Buttons. Mögliche Werte sind *MB_abort*, *MB_cancel*, *MB_ignore*, *MB_no*, *MB_retry*, *MB_yes*.

Ausnahmen

Keine.

Siehe auch

Funktion DM_QueryBox

6.2.8 start()

Diese Funktion startet die eigentliche Dialoganwendung und meldet dazu notwendige Ressourcen beim Fenstersystem an und führt die Startregel des Dialoges aus. Diese Funktion ist nur auf Dialoge anzuwenden.

```
void start
(
  DM_Options options = 0
)
```

Parameter

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_StartDialog

6.2.9 stop()

Diese Funktion beendet einen Dialog. Wenn es sich um den letzten handelt wird die Ereignisschleife verlassen und das Programm beendet.

```
void stop
(
  DM_Options options = 0
)
```

Parameter

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_StopDialog

6.2.10 control()

Diese Funktion erlaubt das Ändern von generellen Einstellungen des Dialog Managers oder der Ausführung von Aktionen.

```
void control
(
  DM_UInt    action,
  DM_Options options = 0
)
```

Parameter

-> DM_UInt action

Aktion die ausgeführt oder Einstellung die geändert werden soll.

-> DM_Options options

Option oder neuer Einstellungswert.

Die genaue Parameternutzung ist bei der Funktion DM_Control im Handbuch „C-Schnittstelle - Funktionen“ zu finden.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_Control

6.2.11 getContent()

Diese Funktion liefert den aktuellen Inhalt eines Objektes mit Textfeld (***Listbox***, ***Poptext*** oder ***Tablefield***) mit einem einzigen Aufruf zurück.

```
DM_Content * getContent
(
  DM_Value  *firstIdx,
  DM_Value  *lastIdx,
  DM_UInt   *count,
  DM_Options options = 0
)
```

Parameter

-> DM_Value *firstIdx
-> DM_Value *lastIdx

Start- und Endindex, wenn nur ein Teilbereich des Inhaltsvektors erfragt werden soll. Andernfalls kann hier *(DM_Value *)0* stehen um den normalen Start- bzw. Endpunkt zu bezeichnen.

<- DM_UInt count

Die tatsächliche Länge des gelieferten Vektors wird hier abgelegt.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Vektor mit dem Textinhalt des Objektes.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_GetContent, DM_SetContent

DM_Object::getContent()

6.2.12 get()

Diese Funktion erfragt den Wert eines Objektattributes und ist mit oder ohne Index anwendbar.

```
DM_Value get
(
  DM_Attribute attr,
  DM_Value    *index   = &DM_ValueVoid,
  DM_Options   options = 0
)
```

Parameter

-> DM_Attribute attr

Attribut dessen Wert erfragt werden soll.

-> DM_Value *index

Optional angebbarer Index-Wert für nicht-skalare Attribute.

-> DM_Options options

Erlaubte Werte sind *0*, *DMF_GetMasterString*, *DMF_GetLocalString*, *DMF_GetTextID*, *DMF_DontFreeLastStrings*.

Rückgabewert

Wert des Attributes.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_GetValue, DM_GetValueIndex

DM_Object::get...()

6.2.13 get…()

Diese Sammlung von Funktionen erfragt den Attributwert und stellt gleichzeitig den Rückgabetyp sicher. Bei Rückgabe eines falschen Typs wird eine Exception ausgelöst.

```
DM_Attribute getAttribute(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                          DM_Options options = 0);

DM_Boolean getBoolean(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                      DM_Options options = 0);

DM_Class getClass(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                  DM_Options options = 0);

DM_Enum getEnum(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                DM_Options options = 0);

DM_Event getEvent(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                  DM_Options options = 0);

DM_ID getId(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
            DM_Options options = 0);

DM_Integer getInteger(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                      DM_Options options = 0);

DM_Index getIndex(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                  DM_Options options = 0);

DM_Type getType(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                DM_Options options = 0);

DM_Method getMethod(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                    DM_Options options = 0);

DM_Pointer getPointer(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                      DM_Options options = 0);

DM_Scope getScope(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                  DM_Options options = 0);

DM_String getString(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                    DM_Options options = 0);
```

Parameter

-> DM_Attribute attr

Attribut dessen Wert erfragt werden soll.

-> DM_Value *index

Optional angebbarer Index-Wert für nicht-skalare Attribute.

-> DM_Options options

Erlaubte Werte sind *0*, *DMF_GetMasterString*, *DMF_GetLocalString*, *DMF_GetTextID*, *DMF_DontFreeLastStrings*.

Rückgabewert

Attributwert mit dem entsprechenden Rückgabetyp für die Funktion.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_GetValue, DM_GetValueIndex

DM_Object::get()

Handbücher „Attributreferenz“, „Objektreferenz“

6.2.14 getVector()

Diese Funktion erlaubt es alle oder einen kontinuierlichen Bereich von Werten eines vektoriellen Attributes in einem Schritt zu erfragen.

```
DM_VectorValue *getVector
(
  DM_Attribute attr,
  DM_Value    *firstIdx = &DM_ValueVoid,
  DM_Value    *lastIdx  = &DM_ValueVoid,
  DM_Options   options  = 0
)
```

Parameter

-> DM_Attribute attr

Vektorielles Attribut dessen Werte erfragt werden sollen.

-> DM_Value *firstIdx
-> DM_Value *lastIdx

Indexbereich (einschließend) aus dem die Attributwerte erfragt werden. Bei Nichtangabe werden die minimalen und maximalen Indizes des sichtbaren Feldes genommen.

-> DM_Options options

Erlaubte Werte sind *0*, *DMF_GetMasterString*, *DMF_GetLocalString*, *DMF_GetTextID*, *DMF_DontFreeLastStrings*.

Rückgabewert

Vektor mit den Attributwerten.

Ausnahmen

DM_Exception	Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_FreeVectorValue, DM_GetVectorValue

DM_Interface::freeValues()

6.2.15 setContent()

Diese Funktion setzt in einem Aufruf den Inhalt eines Objekts und erlaubt es für die Objekte ***Listbox***, ***Tablefield*** und ***Treeview*** effektiv den gesamten Inhalt oder auch nur einen Teilbereich zu setzen.

Der angegebene Indexbereich muss mit der Größe des *content*-Feldes übereinstimmen.

```
void setContent
(
  DM_Value   *firstIdx,
  DM_Value   *lastIdx,
  DM_Content *content,
  DM_UInt     count,
  DM_Options  options = 0
)
```

Parameter

-> DM_Value *firstIdx
-> DM_Value *lastIdx

Anfangs- und Endindex für den Indexbereich der gesetzt werden soll. Die Angabe eines *NULL*-Pointers bewirkt die Verwendung der sich aus dem Objektinhalt ergebende Minimal- und Maximalgrenze. Wird als *lastIdx* ein *NULL*-Pointer angegeben, so ergibt sich die neue Größe des Attributvektors aus der Größe des *content*-Feldes.

-> DM_Content *content

Vektor mit den Werten die gesetzt werden sollen.

-> DM_UInt count

Länge des *content*-Vektors.

-> DM_Options options

Erlaubte Werte sind *0*, *DMF_Inhibit*, *DMF_ShipEvent*, *DMF_UseUserData*, *DMF_OmitActive*, *DMF_OmitStrings*, *DMF_OmitSensitive*.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_GetContent, DM_SetContent

DM_Object::getContent()

6.2.16 set()

Diese Funktion setzt den Wert eines Attributes.

```
void set
(
  DM_Attribute attr,
  DM_Value    *index,
  DM_Value    *data,
  DM_Options   options = 0
)

void set
(
  DM_Attribute attr,
  DM_Value    *data,
  DM_Options   options = 0
)
```

Parameter

-> DM_Attribute attr

Attribut dessen Wert gesetzt werden soll.

-> DM_Value *index

Optionaler Index zum Setzen eines Einzelwertes in einem vektoriellen Attribut oder assoziativen Feld.

-> DM_Value *data

Wert den das Attribut annehmen soll.

-> DM_Options options

Erlaubt ist eine Kombination der Werte *0*, *DMF_Inhibit*, *DMF_ShipEvent*, *DMF_XlateString*.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception	Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_SetValue, DM_SetValueIndex

DM_Object::set…()

Handbücher „Attributreferenz“, „Objektreferenz“

6.2.17 set…()

Diese Sammlung von Funktionen setzt einen Attributwert. Der Wert wird dabei typabhängig übergeben, man spart sich somit den Umgang mit der **DM_Value**-Struktur.

```
void setAttribute(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                  DM_Attribute data, DM_Options options = 0);

void setBoolean(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                DM_Boolean data, DM_Options options = 0);

void setClass(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
              DM_Class data, DM_Options options = 0);

void setEnum(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
             DM_Enum data, DM_Options options = 0);

void setEvent(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
              DM_Event data, DM_Options options = 0);

void setId(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
           DM_ID data, DM_Options options = 0);
```

```
void setInteger(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                DM_Integer data, DM_Options options = 0);

void setIndex(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
              DM_Index data, DM_Options options = 0);

void setType(DM_Attribute attr , DM_Value *index = &DM_ValueVoid,
             DM_Type data, DM_Options options = 0);

void setMethod(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
               DM_Method data, DM_Options options = 0);

void setPointer(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
                DM_Pointer data, DM_Options options = 0);

void setScope(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
              DM_Scope data, DM_Options options = 0);

void setString(DM_Attribute attr, DM_Value *index = &DM_ValueVoid,
               DM_String data, DM_Options options = 0);
```

Parameter

-> DM_Attribute attr

Liste von Attributen und Objekten deren Wert erfragt werden soll.

-> DM_Value *index

Hier kann ein optionaler Index zum Zugriff auf Felder angegeben werden.

-> DM_... data

Wert den das Attribut annehmen soll.

-> DM_Options options

Erlaubt ist eine Kombination der Werte *0*, *DMF_Inhibit*, *DMF_ShipEvent*, *DMF_XlateString*.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_SetValue, DM_SetValueIndex

DM_Object::set()

Handbücher „Attributreferenz“, „Objektreferenz“

6.2.18 setVector()

Diese Funktion setzt bei einem vektoriellen Attribut gleichzeitig mehrere Werte.

```
void setVector
(
  DM_Attribute    attr,
  DM_Value       *firstIdx,
  DM_Value       *lastIdx,
  DM_VectorValue *values,
  DM_Options      options = 0
)
```

Parameter

-> DM_Attribute attr

Attribut dessen Werte gesetzt werden sollen.

-> DM_Value *firstIdx
-> DM_Value *lastIdx

Mit diesen Parametern wird der Indexbereich festgelegt für den die Attributwerte gesetzt werden. Dieser Bereich muss mit den Anzahl der Elemente im *values*-Vektor übereinstimmen. Bei Angabe eines *NULL*-Pointers als Startindex wird der erste Index der Attributes genommen. Bei Angabe eines *NULL*-Pointers als Endindex ergibt sich die Größe des vektoriellen Feldes nach dem Inhalt im *values*-Parameter.

-> DM_VectorValue *values

Feld von Werten die gesetzt werden sollen.

-> DM_Options options

Erlaubt ist eine Kombination der Werte *0*, *DMF_Inhibit*, *DMF_ShipEvent*, *DMF_XlateString*.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_GetVectorValue, DM_SetVectorValue

DM Object::getVector()

6.2.19 reset()

Diese Funktion setzt den Wert eines Attributes auf seinen geerbten Wert zurück.

```
void reset
(
  DM_Attribute attr,
  DM_Value    *index,
  DM_Options   options = 0
)
```

Parameter

-> DM_Attribute attr

Attribut dessen Wert zurückgesetzt werden soll.

-> DM_Value *index

Index-Parameter für das Zurücksetzen eines Einzelwertes in einem Feld.

-> DM_Options options

Erlaubt ist eine Kombination der Werte *0*, *DMF_Inhibit*, *DMF_ShipEvent*.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

DM_Object::get(), **DM_Object::set()**

6.2.20 call()

Diese Funktion ruft eine Regel oder Funktion auf.

```
DM_Value call
(
  DM_ID      rule    = 0,
  DM_UInt    argc    = 0,
  DM_Value  *argvec  = &DM_ValueVoid,
  DM_Options options = 0
)
```

Parameter

-> DM_ID rule

Objekt-ID der Regel oder Funktion die aufgerufen werden soll.

-> DM_UInt argc

Anzahl der Parameterwerte die im *argvec*-Feld mitgegeben werden. Höchster zulässiger Wert ist *16*.

-> DM_Value *argvec

Vektor der Parameter, die dem Aufruf mitgegeben werden sollen.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Rückgabewert der Regel oder Funktion.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_CallFunction, DM_CallMethod, DM_CallRule

DM_Object::callMethod()

6.2.21 callMethod()

Diese Funktion ruft eine Objekt-Methode auf.

```
DM_Value callMethod
(
  DM_Method  method,
  DM_UInt    argc,
  DM_Value  *argvec  = (DM_Value *)0,
  DM_Options options = 0
)
```

Parameter

-> DM_Method method

Methode die aufgerufen werden soll.

-> DM_UInt argc

Anzahl der Parameterwerte die im *argvec*-Feld mitgegeben werden. Höchster zulässiger Wert ist *16*.

-> DM_Value *argvec

Parameter die der Methode mitgegeben werden sollen.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Rückgabewert der Methode.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_CallMethod

DM_Object::call()

6.2.22 sendevent()

Diese Funktion sendet ein externes Ereignis an ein Objekt. Dieses Ereignis wird in eine Queue eingetragen und entsprechend der Ereignisverarbeitung aufgerufen. Erlaubt ist ebenfalls die Mitgabe von Parametern.

```
void sendevent
(
  DM_Value  *data,
  DM_UInt    argc    = 0,
  DM_Value  *argv    = &DM_ValueVoid,
  DM_Options options = 0
)
```

Parameter

-> DM_Value *data

Dieser Parameter definiert das externe Ereignis das ausgelöst werden soll.

-> DM_UInt argc

Anzahl der Parameterwerte die im *argv*-Feld mitgegeben werden. Höchster zulässiger Wert ist *16*.

-> DM_Value *argv

Parameter die der Ereignisregel mitgegeben werden. Diese müssen mit der Definition der Ereignisregel übereinstimmen.

-> DM_Options options

Erlaubt ist eine Kombination der Werte *0*, *DMF_DontTrace* und *DMF_Synchronous*.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktionen DM_QueueExtEvent, DM_SendEvent

Kapitel „Externe Ereignisse“ im Handbuch „Regelsprache“

6.2.23 getToolkitData()

Diese Funktion erfragt Fenstersystem-spezifische Daten zu einem Objekt.

```
FPTR getToolkitData
(
  DM_Attribute attr
)
```

Parameter

-> DM_Attribute attr

Fenstersystem-Attribut das abgefragt werden soll.

Rückgabewert

Je nach Art des abgefragten Attributes liefert die Funktion den Wert auf *FPTR* konvertiert.

Ausnahmen

Keine.

Siehe auch

Funktion DM_GetToolkitData

6.2.24 setToolkitData()

Diese Funktion bietet einen direkten Zugang zum Fenstersystem. Daten die nicht vom IDM, aber von Fenstersystem unterstützt werden, können hier gesetzt werden.

```
void setToolkitData
(
  DM_Attribute attr,
  FPTR         value,
  DM_Options   options = 0
)
```

Parameter

-> DM_Attribute attr

Fenstersystem-Attribut das gesetzt werden soll.

-> FPTR value

Wert der gesetzt werden soll.

-> DM_Options options

Unbenutzt, muss *0* sein.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Funktion DM_SetToolkitData

6.3 Klasse DM_Record

Diese Klasse umfasst alle Funktionen, die für die Anbindung eines Records mit Funktionen auf der Dialogseite an eine C++-Klasse bzw. deren Instanzen (Exemplare der Klasse) notwendig sind. Zum Teil werden die hier aufgeführten Funktionen erst durch den Codegenerator generiert, beispielsweise **DM_Record::call()**.

6.3.1 DM_Record() (Konstruktor)

Zusätzlich zum Default-Konstruktor erlaubt dieser die direkte Zuordnung der Klasseninstanz zu einem IDM-Objekt.

```
DM_Record
(
  DM_ID newId = (DM_ID)0
)
```

Parameter

-> DM_ID newId

Objekt-ID, auf die dann die weiteren Klassenmethoden wirken.

Rückgabewert

Keiner.

Ausnahmen

Keine.

Siehe auch

Klasse DM_Object, **DM_Record::id()**, **DM_Record::connect()**

6.3.2 id()

Diese Funktion setzt die Zuordnung auf ein anderes IDM-Objekt um.

```
void id
(
  DM_ID newId
)
```

Parameter

-> DM_ID newId

Objekt-ID, auf die dann die weiteren Klassenmethoden wirken.

Rückgabewert

Keiner.

Ausnahmen

Keine.

Siehe auch

DM_Object::id(), Klasse DM_Object

6.3.3 getRecInfo()

Diese Funktion ist nur dem IDM vorbehalten und sollte nicht vom Anwendungsprogrammierer aufgerufen werden.

6.3.4 connect()

Diese Funktion verbindet die Instanz einer C++-Klasse mit einem Record-Objekt auf der Dialog-Seite, sodass Methodenaufrufe von der Dialog-Seite möglich sind und ebenso das Zurücktransferieren der Werte, die auf der C++-Seite gesetzt wurden.

Vorher muss aber die Klasse mittels **DM_Record::bind()** angebunden worden und durch **DM_Record::id()** erst eine Zuordnung zum Record-Objekt gesetzt sein. Anschließend kann man Funktionen von der Dialog-Seite aufrufen oder auf der C++-Seite die Methoden **DM_Record::get()** und **DM_Record::put()** zum Holen und Setzen der Record-Strukturwerte verwenden.

Es ist zu beachten das die Struktur der C++-Klasse mit dem Record übereinstimmt, sonst kommt es bei den Methodenaufrufen und der Transferierung der Record-Strukturwerte zu Fehlern.

```
DM_Boolean connect
(
)
```

Parameter

Keine.

Rückgabewert

DM_TRUE	bei erfolgreicher Anbindung der C++-Instanz zu einem Dialog-Record
DM_FALSE	im Fehlerfall

Ausnahmen

Keine.

Siehe auch

Funktionen DM_BindClass, DM_ConnectClass

DM_Record::bind(), **DM_Record::disconnect()**, **DM_Record::get()**, **DM_Record::id()**, **DM_Record::put()**

Attribute connect, shadow

6.3.5 disconnect()

Diese Funktion trennt die Verbindung zwischen C++-Klasseninstanz und Record auf der Dialogseite. Danach ist es nicht mehr möglich von der Dialogseite Methoden aufzurufen. Abgekoppelt ist damit

auch das dynamische Löschen der Klasseninstanz, wenn der Record auf Dialogseite zerstört wird.

```
DM_Boolean disconnect
(
)
```

Parameter

Keine.

Rückgabewert

DM_TRUE	Trennen war erfolgreich
DM_FALSE	Fehlerfall

Ausnahmen

Keine.

Siehe auch

DM_Record::connect()

6.3.6 get()

Diese Funktion überträgt die Attributwerte des verbundenen Records auf der Dialogseite in die Struktur der Klasseninstanz auf der C++-Seite.

Beim Aufruf einer Record-Funktion von der Dialogseite aus erfolgt die Übertragung durch den IDM.

```
void get
(
)
```

Parameter

Keine.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception	Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

DM_Record::connect(), **DM_Record::put()**

6.3.7 put()

Diese Funktion überträgt die Strukturwerte der C++-Klasseninstanz in die Attribute des verbundenen Records auf der Dialogseite.

Innerhalb eines Aufrufs einer Record-Funktion von der Dialogseite erfolgt die Übertragung durch den IDM.

```
void put
(
)
```

Parameter

Keine.

Rückgabewert

Keiner.

Ausnahmen

DM_Exception Funktion konnte nicht fehlerlos ausgeführt werden.

Siehe auch

Record::connect(), **DM_Record::put()**

6.3.8 bind()

Diese Funktion bindet eine C++-Klasse an einen Record auf der Dialogseite. Dadurch werden alle notwendigen Informationen dem Dialog Manager bekannt gemacht, um Instanzen dieser C++-Klasse anzulegen, zu löschen oder Methoden aufzurufen.

Diese Funktion sollte vor dem Start des Dialoges (was im Normalfall mit **DM_StartDialog()** passiert) aufgerufen werden, um sicherzustellen das alle statischen Records mit C++-Instanzen verbunden werden.

```
DM_Boolean bind
(
  DM_ID parentId,
  DM_String path
)
```

Parameter

-> DM_ID parentId

Vater, z.B. Dialog oder Modul, der den Record mit Funktionen enthält.

-> DM_String path

Eindeutiger Pfad bzw. Bezeichner um den Record mit Funktionen zu identifizieren.

Rückgabewert

DM_TRUE	Anbindung war erfolgreich
	Fehler aufgetreten

Ausnahmen

Keine.

Beispiel

Für einen Dialog, der einen Record „Rec“ mit der Funktion „F“ definiert, kann das anbinden der Klasse wie folgt aussehen:

```
@@@:INCLUDE@@@
@@@:CLASS/Rec@@@
@@@:IMPL/Rec@@@

// nun die Methode F implementieren
DM_Integer Rec::F(DM_String S)
{
  return strlen(A) + strlen(S) + B;
}
...
int DML_c DM_CALLBACK AppMain(int argc, char **argv)
{
  DM_ID dialogID;
  DM_Initialize(&argc, argv);
  dialogID = DM_LoadDialog("dialog.dlg", 0);
  Rec::bind(dialogID, "Rec");
  DM_StartDialog(dialogID, 0);
  DM_EventLoop(0);
  return 0;
}
```

Siehe auch

Funktion DM_ConnectClass

DM_Record::connect()

6.3.9 call()

Dieser Funktion ist nur dem IDM vorbehalten und sollte nicht vom Anwendungsprogrammierer aufgerufen werden. Sie dient zum dynamischen Anlegen und Löschen von Klasseninstanzen und dem

Methodenaufruf von der Dialog Manager Seite aus.

7 Schnittstellen-Funktionen

7.1 DM_BindClass

Diese Funktion bindet den Funktionspointer für eine aus einem ***Record*** mit Funktionen generierte C++-Klasse an ein IDM Record-Objekt an.

```
DM_Boolean DML_default DM_EXPORT DM_BindClass
(
  DM_ID      objID,
  DM_String  path,
  DM_RecordFunc classFunc,
  DM_Options options
)
```

Parameter

-> IDM_ID objID

Hier muss entweder die ID des Records angegeben werden zu dem verbunden werden soll oder der Vater (z.B. das Modul oder der Dialog) von dem man mit zusätzlicher Angabe des Pfades den Record bestimmen kann.

-> IDM_String path

Ist dieser Parameter nicht *NULL* oder ein Leerstring, so ermittelt sich der zu verbindende Record über den hier vollständig angegebenen Pfad von der *objID* als Vater ausgehend.

-> IDM_RecordFunc classFunc

Adresse der generierten statischen C++-Anbindungsfunktion an den Dialog Manager.

-> DM_Options options

Dieser Parameter ist mit *0* zu belegen.

Rückgabewert

DM_TRUE	Anbindung war erfolgreich
DM_FALSE	Anbindung konnte nicht durchgeführt werden

Anmerkung

Diese Funktion muss nicht direkt vom IDM-Anwender aufgerufen werden. Es sollte nur die generierte Klassenmethode **DM_Record::bind()** verwendet werden.

7.2 DM_ConnectClass

Mit dieser Funktion wird der `this`-Pointer einer C++-Klasseninstanz an ein IDM-***Record***-Objekt, das über Funktionen verfügt, bekannt gemacht. Der IDM-Record weiß damit zu welcher Instanz auf der Anwendungsseite er gehört. Ebenfalls kann man mit dieser Funktion die Verbindung wieder aufheben.

```
DM_Boolean DML_default DM_EXPORT DM_ConnectClass
(
  DM_ID      objID,
  DM_Pointer thisPointer,
  DM_Options options
)
```

Parameter

-> IDM_ID objID

Hier muss die ID des Records angegeben werden der die Verbindung zur C++-Instanz erhalten soll.

-> IDM_Pointer thisPointer

`this`-Pointer der C++-Instanz oder *NULL* um die Verbindung zu trennen.

-> DM_Options options

Dieser Parameter ist mit *0* zu belegen.

Rückgabewert

DM_TRUE	Anbindung war erfolgreich
DM_FALSE	Anbindung konnte nicht durchgeführt werden

Anmerkung

Diese Funktion muss nicht direkt vom IDM-Anwender aufgerufen werden. Er kann dazu auch direkt die von ***DM_Record*** vererbten, generierten Klassenmethoden **connect()** und **disconnect()** für seine C++-Instanz aufrufen.

Siehe auch

DM_Record::connect(), **DM_Record::disconnect()**

8 Attribute

8.1 connect

Identifikator:

.connect

Klassifizierung:	Objektspezifisches Attribut der Klasse RECORD

Definition

Argumenttyp:	boolean
C-Definition:	AT_connect
C-Datentyp:	DT_boolean
COBOL-Definition:	AT-connect
COBOL-Datentyp:	DT-boolean
Zugriff:	set, get

„changed“, d.h. das Attribut kann zum Auslösen von Regeln verwendet werden.

Mit diesem Attribut kann man die Verbindung eines Records mit Funktionen zu einer C++-Klasseninstanz dynamisch steuern. Bei Änderung des Zustandes auf true wird eine C++-Klasseninstanz erzeugt und damit auch der this-Pointer der Instanz gemerkt. Setzt man den Attributwert auf false, so wird die Instanz auf der C++-zerstört.

Dieses Attribut ist nur statisch verfügbar und wird nicht weitervererbt.

Siehe auch

DM_Record::connect

Attribut shadow

8.2 shadow

Identifikator

.shadow

Klassifizierung:	Objektspezifisches Attribut der Klasse RECORD

Definition

Argumenttyp:	pointer
C-Definition:	AT_shadow
C-Datentyp:	DT_pointer
COBOL-Definition:	AT-shadow
COBOL-Datentyp:	DT-pointer
Zugriff:	set, get

„changed“, d.h. das Attribut kann zum Auslösen von Regeln verwendet werden.

Dieses Attribut enthält den this-Pointer der C++-Klasseninstanz von einem Record mit Funktionen der mit einer Klasseninstanz auf der C++-Seite verbunden ist. Keine Verbindung wird durch einen *NULL*-Pointer definiert. Es ist zu beachten, dass beim Setzen des Attributes mit einem falschen Pointer damit die Anwendung unter Umständen abstürzt.

Beim Setzen des Attributes auf einen korrekten this-Pointer wird automatisch auch das .connect-Attribut gesetzt. Beim Setzen des Attributes auf den *NULL*-Pointer wird im Gegensatz auch das .connect-Attribut zurückgesetzt, aber nicht die C++-Klasseninstanz zerstört.

Dieses Attribut ist nur statisch verfügbar und wird nicht weitervererbt.

Siehe auch

DM_Record::connect

Attribut connect

Index

A

AT-connect 52

AT-shadow 53

AT_connect 52

AT_shadow 53

B

bind() 47

C

call() 39, 48

callMethod() 40

Code-Mischer 10

Codegenerator 14

Codemischer 16

connect 52

connect() 45

constructor

DM_Object 24

control() 29

create() 26

D

destroy() 26

disconnect() 45

DM_BindClass 50

DM_ConnectClass 51

DM_Exception 20

DM_Interface 10, 20

freeValues() 23

getValues() 21

resetValues() 22

setValues8) 21

DM_Object 10, 24

call() 39

callMethod() 40

constructor 24

control() 29

create() 26

destroy() 26

get 32

get() 31

getContent() 30

getToolkitData() 42

getVector() 33

id() 24-25

Konstruktor 24

parsepath() 27

query() 27

reset() 39

sendevent() 41

set() 35

set...() 36

setContent() 34

setToolkitData() 43

start() 28

stop() 29

DM_Object() 24

DM_Record 10, 43
bind() 47
call() 48
connect() 45
disconnect() 45
get() 46
getRecInfo() 45
id() 44
Konstruktor 43
put() 47
DM_Record() 43

E

Erzeugen der C++-Datei 13

F

freeValues() 23

G

get() 31, 46
get...() 32
getContent() 30
getRecInfo() 45
getToolkitData() 42
getValues() 21
getVector() 33

I

id() 24-25, 44
Identifikator 3
IDMcpp.h 10, 20

K

Klasse
DM_Interface 20
DM_Object 24
DM_Record 43
Klassen 20
Kommando zum Mischen 16
Kompilieren 19
Kompilieren der C++-Datei 13
Konstruktor
DM_Object 24
DM_Record 43

L

Linken 19

M

merge 11

P

parsepath() 27
put() 47

Q

query() 27

R

reset() 39
resetValues() 22

S

sendevent() 41
set() 35-36
setContent() 34
setToolkitData() 43
setValues() 21
setVector()DM_Object
    setVector() 38
shadow 52
start() 28
stop() 29

W

writecpp 11, 14