ISA Dialog Manager

OLE-Schnittstelle

A.06.03.b

Dieses Handbuch beschreibt die OLE-Schnittstelle des ISA Dialog Managers, die als Option des IDM für Microsoft Windows erhältlich ist. OLE (Object Linking and Embedding) ist eine Technik mit der Objekte miteinander kommunizieren und ineinander eingebettet werden können. Das Handbuch erläutert, wie OLE-Clients und -Server mit dem IDM implementiert werden.

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

1 Einleitung

In diesem Handbuch wird der Einsatz von OLE und Dialog Manager erläutert. Zunächst wird beschrieben, wie der Dialog Manager als OLE-Client und anschließend wie der Dialog Manager als OLE-Server eingesetzt werden kann. In beiden Kapiteln sind Beispiele enthalten, die das prinzipielle Arbeiten mit OLE verdeutlichen sollen.

Die COM-Technologie (Component Object Model) bietet Möglichkeiten, um Anwendungen miteinander kommunizieren zu lassen, inklusive der graphischen Integration. Diese Technologie ist besser bekannt unter dem Schlagwort OLE, was eine „Ausprägung“ von COM ist.

Mit Hilfe des Dialog Managers kann das Projektteam die Anwendungsmöglichkeiten von bestehenden Programmen erweitern, indem es in diese Programme neue, mit dem Dialog Manager definierte, Programmerweiterungen auf Basis von Standardschnittstellen (OLE) einfügt. Damit lassen sich die Schnittstellen und Features des Dialog Managers auch in bestehenden Programmen nutzen, ohne dass bisherige Investitionen an Wert verlieren.

1.1 Voraussetzungen

1.1.1 Entwickler

Die Benutzung der OLE-Funktionalität setzt Grundkenntnisse der COM-Technologie und Windows voraus. Es sollte der Umgang mit .reg-Dateien und mit .idl- bzw. .tlb-Dateien sowie Bedeutung von Interfaces in OLE bzw. COM bekannt sein. Ebenso sollten die Begriffe Methods oder Properties bezüglich des IDispatch-Interfaces bekannt sein.

1.1.2 System

Die OLE-Funktionalität ist nur unter Microsoft Windows funktionsfähig. Auf anderen Systemen sind Dialoge mit OLE-Funktionalität zwar "standalone" ablauffähig, die im Dialog beschriebene OLE-Serverfunktionalität hat dort aber keine Bedeutung.

1.1.3 Container/Client

Die Anwendung (im weiteren Sprachgebrauch der OLE-Client oder kurz Client) sollte über das IDispatch-Interface auf die Schnittstellen des OLE-Servers zugreifen, um die Methoden und Properties nutzen zu können. Des weiteren muss er einige Interfaces besitzen, die es dem Dialog Manager erlauben, eine Kommunikation mit dem Client aufzubauen.

64-Bit Datentypen

Ab Version A.05.02.f kann die **OLE-Schnittstelle** der **64-Bit-Version** des ISA Dialog Managers auch **64-Bit-Datentypen** verarbeiten. Ein 64-Bit-Datentyp wird dabei in den IDM-Datentyp *DT_pointer* umgewandelt, da der ISA Dialog Manager prinzipiell auf 32-Bit-Integerwerte ausgelegt ist.

Hinweis

DT_pointer wird von einem 32-Bit-IDM-OLE-**Server** in den OLE-Datentyp VT_UI4 umgewandelt, während ein IDM-OLE-**Client** VT_UI4 als *DT_integer* interpretiert.

1.1.4 Benutzer

Für den Endanwender oder Benutzer sind keine besonderen Voraussetzungen notwendig. Im Normalfall bleibt ihm der Aufbau der Anwendung als OLE-Client und -Server verborgen. Eine zeitliche Verzögerung durch den erhöhten Kommunikationsaufwand ist möglich, der aber im allgemeinen auf die Startphase bzw. auf das Starten des Servers beschränkt bleibt.

2 Das control-Objekt

Die Aufgaben des Control-Objektes sind abhängig davon, in welcher Art das Control Objekt benutzt wird. Dabei gibt es zwei prinzipielle Möglichkeiten:

» Wenn das Control-Objekt als OLE-Client eingesetzt wird, dann wird mit Hilfe dieses Objektes die Kommunikation mit dem Server durchgeführt. Zusätzlich definiert dieses Objekt dann den Bereich, in welchem der Server innerhalb des Clients erscheinen soll. Alle Aufrufe an den Server zum Setzen und Abfragen von Properties oder zum Aufruf von Methoden gehen dann über dieses Control-Objekt.

» Wird das Control-Objekt als OLE-Server eingesetzt, dient dieses Objekt zur Definition der Schnittstelle und zur Kommunikation mit dem Client. Der Dialog Manager kann nicht all seine Objekte, Methoden, Ressourcen, Attribute etc. als Schnittstelle anbieten. Dies würde den Rahmen der Interfaces in OLE sprengen, und es ist auch nicht sinnvoll, die Gesamtheit eines Dialogs dem fremden Programm darzubieten. Die Attribute, Methoden und Ereignisse des Control-Objektes beschreiben das Interface des OLE-Servers zu seinen Clients.

Definition

```
{ export | reexport } { model } control { <Bezeichner> }
{
  <Standardattribute>
  <Allgemeine Attribute>
  <Geometrieattribute>
  <Rasterattribute>
  <Hierarchieattribute>
  <Layoutattribute>
  <Objektspezifische Attribute>
}
```

Ereignisse

extevent

finish

help

key

paste

select

start

Kinder

canvas

checkbox

edittext

groupbox

image

listbox

menubox

menuitem

menusep

notebook

poptext

pushbutton

radiobutton

rectangle

scrollbar

spinbox

statictext

tablefield

treeview

window

Vater

dialog

groupbox

layoutbox

module

notepage

splitbox

toolbar

window

Menü

Popup-Menü

2.1 Attribute

acc_label

acc_text

accelerator

active

bgc

bordercolor

borderwidth

child[I]

childcount

class

connect

cursor

cut_pending

cut_pending_changed

dialog

external

external[I]

fgc

firstchild

firstrecord

firstsubcontrol

focus

font

function

groupbox

height

help

index

label

lastchild

lastrecord

lastsubcontrol

layoutbox

license_key

mapped

member[I]

membercount

menu

message[I]

mode

model

name

notepage

parent

picture

posraster

real_height

real_sensitive

real_visible

real_width

real_x

real_xraster

real_y

real_yraster

record[I]

recordcount

reffont

scope

sensitive

sizeraster

statushelp

subcontrol[I]

subcontrolcount

toolhelp

userdata

uuid

visible

width

window

xauto

xleft

xraster

xright

yauto

ybottom

yraster

ytop

3 Das subcontrol-Objekt

OLE-Objekte, die als Server im IDM mit Hilfe des IDM-Objektes ***control*** genutzt werden können, haben in der Regel eine recht komplexe Struktur und können aus einer Reihe weiterer Kindobjekte bestehen. Zum Beispiel hat das OLE-Control „Word.Application“ eine Kollektion „Documents“, wo die Dokumente verwaltet werden. Diese Dokumente wiederum haben „Words“, „Sentences“, „Ranges“ usw. als eigene Kinder. Um auf diese Unterobjekte aus der Regelsprache zugreifen zu können, sprich Methoden aufrufen oder Attribute abfragen, wurde das Objekt ***subcontrol*** eingeführt. Das ***subcontrol*** repräsentiert somit ein OLE-Kindobjekt, das direkt oder indirekt von einem OLE-Server verwaltet wird. Der Ausgangspunkt für den Zugriff auf ein solches Objekt ist deswegen immer ein ***control***.

Definition

```
{ export | reexport } subcontrol { <Bezeichner> }
{
  <Hierarchieattribute>
  <Objektspezifische Attribute>
}
```

Ereignisse

Keine

Kinder

subcontrol

Vater

control

subcontrol

Menü

Keins

3.1 Attribute

connect

dialog

external

external[I]

firstrecord

firstsubcontrol

groupbox

label

lastrecord

lastsubcontrol

layoutbox

model

module

notepage

parent

record[I]

recordcount

scope

subcontrol[I]

subcontrolcount

toolbar

userdata

window

3.2 Implizite Erzeugung

Die herausragende Besonderheit der Subcontrols ist, dass diese Objekte implizit erzeugt werden können, ohne dass man vorher das Objekt statisch in einem Dialog definiert oder mit **create()** dynamisch erzeugt hat. Allerdings sollten bei Abfrage von dynamischen OLE Eigenschaften, welche ein Subcontrol erzeugen, unbedingt die Hinweise in Kapitel „Dynamische OLE-Eigenschaften und Subcontrols“ beachtet werden.

Vermutlich ist die implizite Verwendung dieser Objekte die in der Praxis am häufigsten eingesetzte. So kann man beispielsweise bei der Verwendung von Word als OLE-Server wie folgt auf die einzelnen Dokumente zugreifen:

Beispiel

Sei ***Control***„Co“ irgendwo im Dialog definiert.

```
child control Co
{
  .mode mode_client;
  .name "Word.Application";
  .visible true;
```

```
  .connect true;
}
```

Dann könnte in einer Regel folgendes stehen:

```
rule OLETest ()
{
  variable object Doc;
  Doc := Co.Documents:Add();
  // Word besitzt eine Collection mit dem Namen Documents. Mit
  // Co.Documents wird auf dieses OLE-Objekt zugegriffen. Um das
  // Object im IDM ansprechen zu können, wird an dieser Stelle
  // ein Subcontrol mit dem Bezeichner "Documents" implizit
  // erzeugt, das als Kind von Control Co eingetragen wird.
  // Die :Add()-Methode ist ein Mitglied der Collection
  // "Documents". Deswegen kann sie nun auf das gerade eben
  // erzeugte Subcontrol angewandt werden. Dabei wird in Word
  // ein neues Document erzeugt, und als Rückgabewert ein Zeiger
  // auf das IDispatch-Interface dieses Dokuments
  // zurückgeliefert. Um ein entsprechendes Pendant auf der Seite
  // des IDM zu haben, wird auch hier ein Subcontrol als Kind
  // des ersten Subcontrols erzeugt, das selber mit dem
  // Word-Document verbunden ist. Dieses Objekt wird schliesslich
  // der Variablen "Doc" zugewiesen.

  // Nun kann das Subcontrol in Doc dazu benutzt werden, um
  // Methoden des OLE-Documents aufzurufen oder seine Properties
  // zu setzen/abzufragen.
  print Doc.FullName;
}
```

Generell gilt folgendes: Wird auf der Seite von OLE als Rückgabewert einer Methode oder Property ein Zeiger auf ein IDispatch-Interface zurückgeliefert oder wird dieser in einem Parameter an den IDM übergeben, so erzeugt der IDM an dieser Stelle ein Subcontrol, das mit dem IDispatch-Interface verbunden ist. D.h. das *.connect*-Attribut eines solchen Subcontrols ist bereits gleich *true*.

In der Regel wird der Bezeichner (Label) eines so erzeugten Objektes nicht gesetzt, so dass im Tracefile so etwas wie *„subcontrol Co.SUBCONTROL[2]“* erscheint. Eine Ausnahme bildet dabei der Zugriff auf die Properties, die OLE-Objekte zurückliefern. Da diese in der Regel Collections darstellen und dem IDM somit der Name bekannt ist, bekommt ein implizit erzeugtes Subcontrol in diesem Fall als Bezeichner den Namen der Property.

Beispiel

```
Doc1 := Co.Documents:Add();
Doc2 := Co.Documents:Item(1);  // Annahme: In Word gab es zuvor
                               // keine geoeffneten Dokumente.
```

In diesem Beispiel werden in der ersten Zeile zwei Subcontrols erzeugt: Eines mit dem Bezeichner „Documents“, ein anderes mit dem Bezeichner „SUBCONTROL“. Der Bezeichner des zweiten Subcontrols erklärt sich dadurch, dass der IDM beim Verarbeiten der Add()-Methode keine weiteren Informationen zur Verfügung hat. In der zweiten Zeile wird nur noch ein Subcontrol mit dem Bezeichner „SUBCONTROL[2]“ erzeugt. Das Erzeugen des Subcontrols für Documents entfällt, da der IDM an dieser Stelle erkennt, dass das Objekt bereits existiert. Es entsteht deswegen folgende Situation. Auf der Seite von OLE haben wir zwei Objekte: zum einem die Collection „Documents“ zum anderen das leere Dokument. Auf der Seite vom IDM haben wir drei Objekte: ein Subcontrol für die Collection „Documents“ und zwei Subcontrols in den Variablen Doc1 und Doc2, die mit dem leeren Dokument auf der OLE-Seite verbunden sind. Es bleibt also dem IDM-Programmierer überlassen, durch guten Programmierstil eine Flut von überflüssigen Subcontrols zu vermeiden.

Ein besorgter Leser mag sich jetzt vielleicht fragen, was denn nun mit all den Subcontrols geschehen muss, wenn diese nicht länger gebraucht werden. Sehen Sie dazu die Erläuterungen in Kapitel „Garbage Collection“.

3.3 Dynamische OLE-Eigenschaften und Subcontrols

Beim Aufruf einer dynamischen Eigenschaft eines OLE-Objekts kann es zum Anlegen eines neuen Subcontrols kommen, wenn diese Eigenschaft (z.B. .„ActiveSheet“ bei MICROSOFT EXCEL) ein Objekt zurückgibt. Dies ist erst einmal unproblematisch. Bei einem erneuten Aufruf dieser dynamischen Eigenschaft referenziert der IDM nun aber nicht das neue, jetzt gelieferte Objekt, sondern das bereits vorhanden und angelegte Subcontrol, welches der IDM quasi „cached“.

Sollte dieses Caching unerwünscht bzw. problematisch sein, kann es wie folgt umgangen werden:

- Nach dem Zugriff auf eine OLE-Eigenschaft, welche ein Subcontrol erzeugt, kann dieses Subcontrol explizit mit **:destroy()** zerstört werden, wenn es nicht mehr benötigt wird (vor dem nächsten Zugriff auf diese OLE-Eigenschaft!)
- Nach dem Zugriff auf eine OLE-Eigenschaft, welche ein Subcontrol erzeugt, kann das Attribut *.label* dieses Subcontrols auf "" (Leerstring) gesetzt werden. Hierdurch bleibt das Subcontrol-Objekt weiterhin verbunden und verfügbar. Es wird im Zuge der normalen Garbage Collection (siehe Kapitel „Garbage Collection“) gelöscht. Bei einem erneuten Zugriff auf die OLE-Eigenschaft wird ein neues Subcontrol-Objekt anlegen, da das alte nicht mehr gefunden wird (Die interne Suche erfolgt über *.label*).

Ein Setzen von *.connect* des betroffenen Subcontrols auf *false* ist hingegen keine Lösung, da das Subcontrol erhalten bleibt und lediglich alle weiteren Zugriffe fehl schlagen.

3.4 Garbage Collection

Alle implizit erzeugten Subcontrols müssen auch wieder gelöscht werden. Zu diesem Zweck merkt sich der IDM, von wie vielen Objekten ein Subcontrol gerade referenziert wird. Wird dabei festgestellt, dass ein Subcontrol von keiner Variablen (lokal, global, statisch) oder benutzerdefiniertem Attribut mehr verwendet wird, so wird das Subcontrol zerstört. Es gibt dabei zwei Strategien, die im

Folgenden vorgestellt werden. Wichtig: Folgende Erklärungen sind auf jeden Fall mit einem Änderungsvorbehalt versehen, da es nicht abzusehen ist, ob in Zukunft andere Strategien eingesetzt werden.

1. Wird ein implizit erzeugtes Subcontrol einer Variablen (oder Ähnlichem) zugewiesen, und wird diese Variable zu einem späteren Zeitpunkt wieder mit einem anderen Wert überschrieben, so kann das Subcontrol wieder freigegeben werden, vorausgesetzt, es hat keine Kinder.

   Beispiel

   ```
   Doc := Co.Documents:Add();  // 2 Subcontrols (A für Documents und B für
   Dokument, das
                               // von Add() erzeugt worden ist) werden implizit
   erzeugt.
   Doc := Co;                  // Das Subcontrol B wird nicht länger von
   Variable Doc
                               // benutzt und kann deswegen zerstoert werden.
   Nun kann
                               // auch Subcontrol A feststellen, dass es keine
   Kinder
                               // mehr besitzt und kann sich ebenfalls
   zerstoeren.
   ```

2. Bei ungünstigen Konstellationen kann es passieren, dass ein Subcontrol es nicht merkt, dass es eigentlich zerstört werden kann. Dies passiert z.B., wenn ein gerade erzeugtes Subcontrol nirgendwo zugewiesen wird. In diesem Fall fehlt das auslösende Moment, um das Objekt wegzuwerfen. Deswegen startet der IDM von Zeit zu Zeit Säuberungsaktionen. Dabei werden implizit erzeugte Subcontrols daraufhin getestet, ob diese nicht mehr gebraucht werden und im positiven Fall weggeworfen.

Wichtig

Da die Garbage-Collection vom IDM nur die Referenzen aus der Regelsprache berücksichtigen kann, ist Vorsicht geboten, wenn die ObjectIDs von Subcontrols aus der C-Schnittstelle (für COBOL und C++ gilt das gleiche) verwaltet werden. Wenn eine solche ID in C irgendwo zwischengespeichert wird, bekommt der IDM dieses nicht mit. Nach dem Aufruf der C-Funktion könnte der IDM feststellen, dass das Subcontrol nicht mehr gebraucht wird, und dieses entsorgen. Wird die Kontrolle wieder auf die C-Seite übergeben, kann dort nicht mehr davon ausgegangen werden, dass die zuvor gespeicherte ObjektID noch gültig ist. Es sollte deswegen vermieden werden, die Subcontrols in C zu speichern. Möchte man das trotzdem tun, muss sicher gestellt werden, dass das Objekt in der Regelsprache noch referenziert wird (z.B. in einer globalen oder statischen Variablen oder in einem benutzerdefinierten Attribut). Nur so kann garantiert werden, dass das Subcontrol nicht weggeworfen wird.

3.5 Beispiel

In dem folgenden kleinen Beispiel wird gezeigt, wie man Subcontrols verwenden kann:

```
dialog Word
```

```
window WnFenster
{
  .active false;
  .width 400;
  .height 100;
  .title "Word.Document.8";

  on close
  {
      if Co.connect then
        Co:Quit();
      endif
      exit();
  }

  child control Co
  {
    .visible true;
    .active false;
    .xauto 0;
    .xleft 20;
    .xright 20;
    .yauto 0;
    .ytop 20;
    .ybottom 40;
    .mode mode_client;
    .name "Word.Application";
    .connect true;
  }
  child pushbutton PbOpen
  {
    .xauto 1;
    .xleft 100;
    .width 76;
    .yauto -1;
    .height 27;
    .ybottom 10;
    .text "Open Doc";
    on select
    {
        variable object Doc:=null;

        // Setzt voraus, dass die Datei tatsaechlich existiert
        Doc := Co.Documents:Open("d:/tmp/altesdokument.doc");
```

```
      if Doc <> null then
          print "DocName: " + Doc.FullName;
          print "DocSaveStatus: " + Doc.Saved;
      endif
  }
}
child pushbutton PbAdd
{
  .xauto 1;
  .xleft 20;
  .width 76;
  .yauto -1;
  .height 27;
  .ybottom 10;
  .text "New Doc";
  on select
  {
      Co.Documents:Add("d:/tmp/neuesdokument.doc");
      // Setzt voraus, dass die Datei tatsaechlich existiert
  }
}
child pushbutton PbClose
{
  .xleft 180;
  .width 92;
  .yauto -1;
  .height 27;
  .ybottom 10;
  .text "Close Doc";
  on select
  {
      // Erstes (bzgl. der Documents-Liste) Document schliessen
      Co.Documents:Item(1):Close();
  }
}
child pushbutton PbQuit
{
  .xauto -1;
  .width 85;
  .xright 20;
  .yauto -1;
  .height 27;
  .ybottom 10;
  .text "Quit";
  on select
  {
```

```
        // Word beenden
        Co:Quit();
    }
  }
}
on dialog start
{
    Co.Visible := true;
}
```

4 Der Dialog Manager als OLE-Client

In diesem Kapitel wird beschrieben, wie man OLE-Objekte in den Dialog Manager einbinden kann.

Das Control-Objekt stellt im Client-Mode die Verbindung zu dem OLE-Server dar. In ihm wird definiert, welcher Server zu welcher Zeit in welcher Form erscheinen soll. Dazu werden die Attribute

- *.picture* für das dargestellte Bild im inaktiven Zustand
- *.name* für den Namen des externen Servers
- *.connect* für den Verbindungszustand
- *.visible* für die Sichtbarkeit
- *.active* für den Aktivierungszustand
- *.xleft*, *.ytop*, … für die Geometrie

gesetzt.

4.1 Einbindung des Servers

Die eigentliche Einbindung des Servers in den Dialog geschieht über das Control-Objekt. Hierzu müssen folgende Attribute mit den entsprechenden Werten versorgt sein:

- *.name* muss den Namen des zu verwendenden Servers oder dessen ProgID beinhalten
- *.xleft*, *.ytop*, … beinhaltet die Information, an welcher Stelle der Server erscheinen soll, falls das Control-Objekt nicht auf Top-Level definiert worden ist.
- *.visible* enthält die Information, ob das Control sichtbar sein soll
- *.sensitive* enthält die Information, ob das Control selektierbar sein soll
- *.active* enthält die Information, ob der Server aktiv und damit ansprechbar ist.
- *.connect* enthält die Information, ob eine Verbindung zum Server existiert.
- *.picture* enthält das Bild, das im inaktiven Zustand des Servers angezeigt werden soll.

Beispiel

```
tile Server_Bild "IDM_IMAGES:isaicon.gif";
window WnClient
{
  .title "Einbindung OCX in IDM-Dialog";
  child control Ctrl_Grid
  {
    .visible true;
    .active false;
    .xleft 11;
    .width 249;
    .ytop 48;
```

```
    .height 169;
    .picture Server_Bild;
    .mode mode_client;
    .name "MSGrid.Grid";
    .connect false;
    }
  }
```

4.2 Aktivierung des Servers

Die eigentliche Aktivierung des Servers erfolgt durch das Setzen der Attribute

- *.connect*
- *.active*

auf den Wert *true*. Wenn beide Attribute diesen Wert haben, existiert eine Verbindung zu dem OLE-Server und dessen Methoden können aufgerufen bzw. dessen Attribute können abgefragt und gesetzt werden.

Dabei sollte man beachten, dass man das Umsetzen des Attributs *.connect* auf *true* immer abfragen sollte, da u.U. der Server nicht gestartet werden kann.

Beispiel

```
rule void Aktivierung
{
setvalue(Ctrl_Grid, .connect, true, true);
!! Das muss hier abgeprueft werden, da der
!! Verbindungsaufbau
!! u.U. fehlschlagen kann
if (Ctrl_Grid.connect = true) then
    WnClient.title := "Client + Server";
    !! Zum Schluss machen wir den Client noch aktiv
    Ctrl_Grid.active := true;
else
    print "konnte nicht verbinden";
endif
}
```

4.3 Benutzung des Servers

Der Aufruf von Methoden am Server sowie das Abfragen und Setzen von Attributen (Properties) am Server geschieht ausschließlich über das Control-Objekt. Dazu wird eine Syntax benutzt, die den benutzerdefinierten Attributen und Methoden entspricht, mit dem Unterschied, dass diese Attribute und Methoden am Control-Objekt nicht definiert sein dürfen. Wenn also auf ein benutzerdefiniertes Attribut am Control-Objekt zugegriffen wird, das dort nicht definiert ist, wird es an den Server

weitergereicht. Genauso verhält es sich mit den Methoden. Wenn eine Methode am Control-Objekt aufgerufen wird, die dort nicht definiert ist, wird versucht, diese Methode am Server aufzurufen. Ist sie dort auch nicht definiert, kommt es zu einem Fehler, der in der Trace- oder Logdatei mitprotokolliert wird.

Beispiel

Die Attribute .Cols, .Rows und :GridLines sind Attribute des verwendeten OLE-Servers und dürfen daher nicht im Control-Objekt definiert sein.

```
child control Ctrl_Grid
{
    .visible true;
    .active false;
    .xleft 11;
    .width 249;
    .ytop 48;
    .height 169;
    .picture Server_Bild;
    .mode mode_client;
    .name "MSGrid.Grid";
    .connect false;

    rule void Aktivierung
    {
variable integer Cols := 0;
variable integer Rows := 0;

setvalue(Ctrl_Grid, .connect, true, true);
!! Das muss hier abgeprueft werden, da der
!! Verbindungsaufbau
!! u.U. fehlschlagen kann
if (Ctrl_Grid.connect = true) then
    WnClient.title := "Client + Server";
    Cols := atoi(EtCols.content);
    Rows := atoi(EtRows.content);
    !! Setzen der eingestellten Zeilen und Spaltenanzahl
    if (Cols <> 0) then
Ctrl_Grid.Cols := Cols;
    endif
    if (Rows <> 0) then
Ctrl_Grid.Rows := Rows;
    endif
    Ctrl_Grid.GridLines := CbGridLines.active;
    !! Zum Schluss machen wir den Client noch sichtbar
    Ctrl_Grid.active := true;
else
```

```
    print "konnte nicht verbinden";
endif
   }
```

4.4 Das Herausfinden der Schnittstellen

Wenn man einen OLE-Server benutzen möchte, muss man seine Schnittstellen kennen. Diese Schnittstellen bestehen aus Attributen (Properties) und Methoden. Um diese Schnittstellen kennen zu lernen, gibt es unterschiedliche Wege, die hier kurz vorgestellt werden sollen:

- Lesen des Handbuchs: Zu den verwendeten OLE-Servern muss man das entsprechende Handbuch suchen, in dem die Attribute und Methoden genau definiert sind. Dieses ist der sicherste Weg, scheitert jedoch in der Regel daran, dass solche Handbücher nicht verfügbar sind.
- Nachsehen mit Hilfe des Microsoft C++ Compilers: Ab der Version 4.0 beinhaltet der Microsoft C++ Compiler Hilfsmittel, die es einem erlauben, Schnittstellen von OLE-Objekten zu erfragen.

4.4.1 Microsoft C++ Compiler Version 4.x

Bei der Version 4 des Microsoft C-Compilers muss man wie folgt vorgehen, um Informationen über OLE-Objekte zu erlangen:

- Starten des OLE-COM Object Viewers über das Developer Studio (Menü "Tools")
- Auswahl des gesuchten Controls
- Auswahl des gesuchten Interfaces (in der Regel IDispatch....) mit einem Doppelklick. In der dargestellten Liste sind nur die fettgedruckten Interfaces wirklich verfügbar.
- Auswahl der gesuchten Information in der Combobox links oben im neuen Fenster
- Auswahl der gesuchten Methode in der linken oberen Listbox oder des gesuchten Attributs in der rechten oberen Listbox, um nähere Informationen (Datentyp, Parameter, ..) zu erfragen.

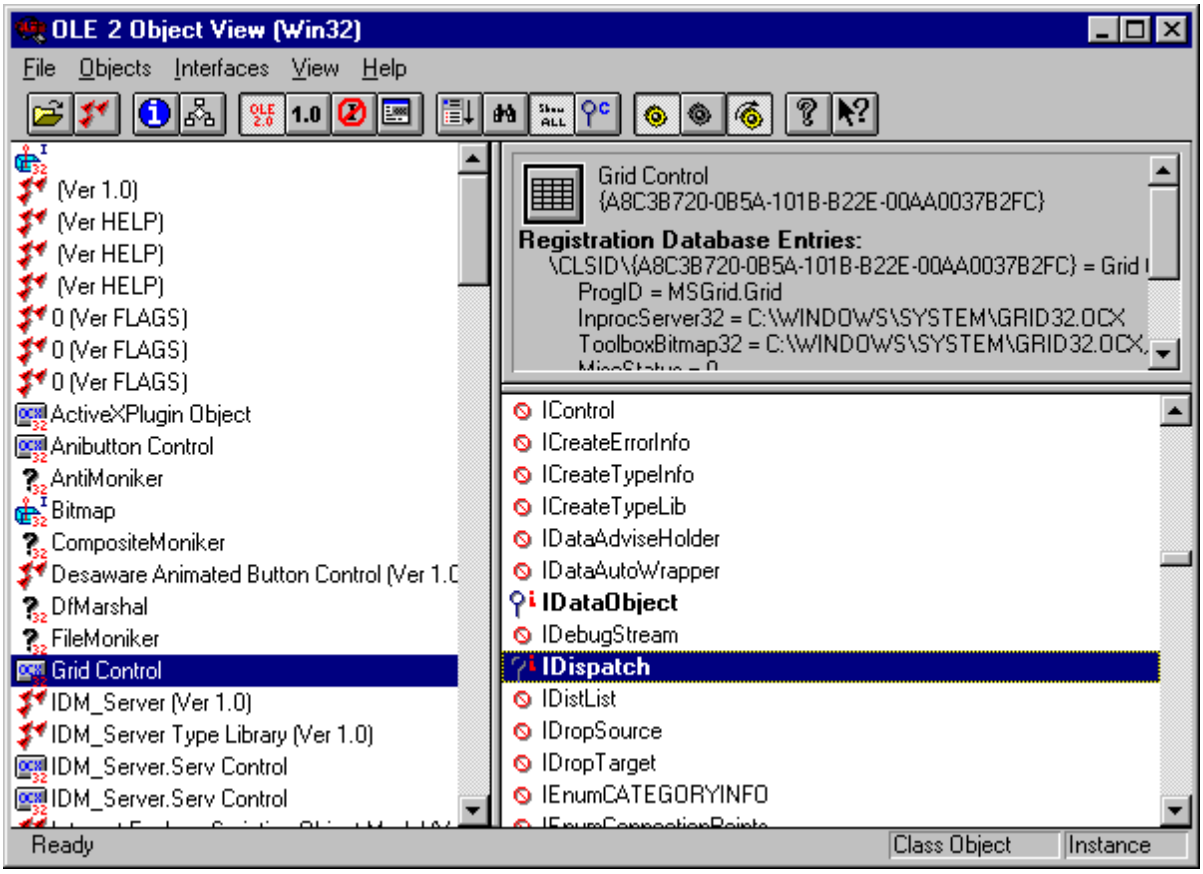

Abbildung 1: *OLE2 Object View des MSC Version 4.0 nach dem Start*

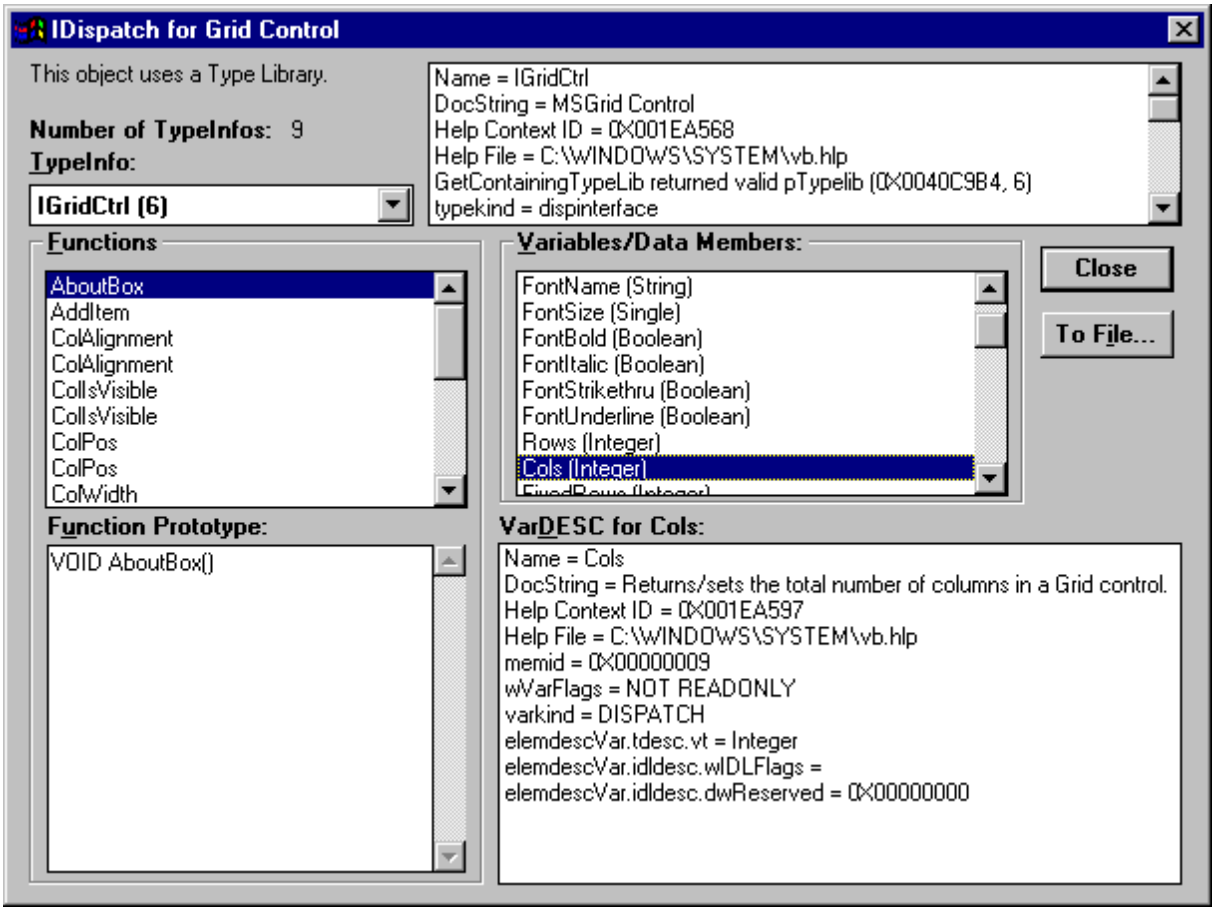

Abbildung 2: *Auswahl des gesuchten Attributs in MSC Version 4.0*

4.4.2 Microsoft C++ Compiler Version 5.0

Bei der Version 5.0 des Microsoft C-Compilers muss man wie folgt vorgehen, um Informationen über OLE-Objekte zu erlangen:

- Starten des OLE-COM Object Viewers
- Anwahl des Eintrags "Control" in der linken Liste
- Auswahl des gesuchten Controls
- Auswahl des Menüeintrags "View Type Information..." im Menü "Object" oder im Popup-Menü
- Auswahl des gesuchten Interfaces (in der Regel dispinterface ....)
- Auswahl des Eintrags "Methods", um Methoden zu betrachten, oder "Properties", wenn man Attribute erfragen möchte.
- Auswahl der gesuchten Methode oder des gesuchten Attributs, um nähere Informationen (Datentyp, Parameter, ..) zu erfragen.

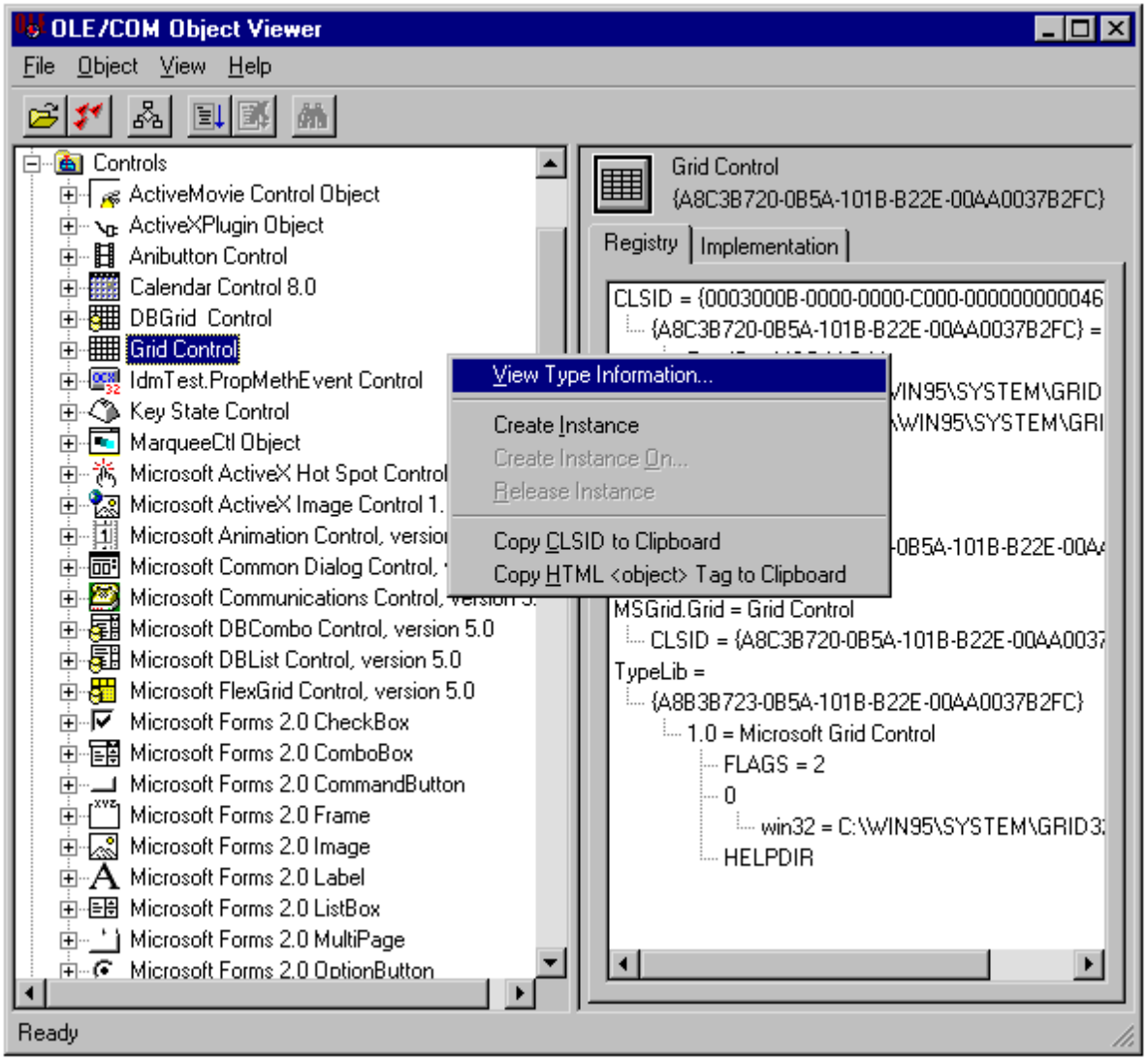

Abbildung 3: *Auswahl eines Controls in MSC Version 5.0*

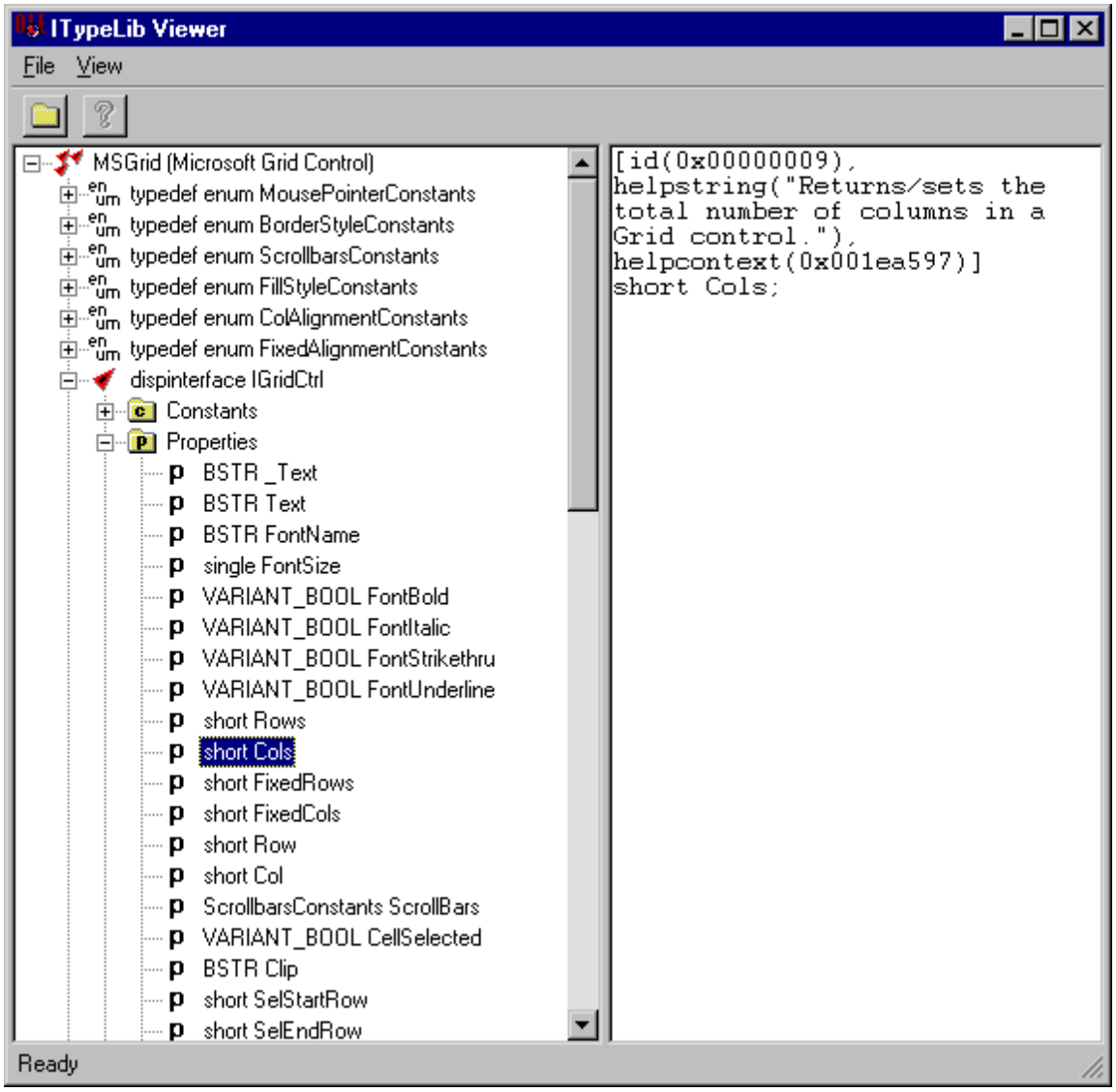

Abbildung 4: *Auswahl einer Property in MSC Version 5.0*

4.5 Verwendung des Grid-Controls

In diesem Beispiel wird gezeigt, wie man das Grid-Control in seinen Dialog einbauen und verwenden kann.

```
dialog Ocx
{
}
color CLRed rgb(255,0,0), grey(0);
font FnInv "10.MS Serif";
tile Server_Bild "IDM_IMAGES:isaicon.gif";
model pushbutton MpbVisible
{
  .xleft 458;
  .ytop 157;
```

```
  .text "Visible";
  boolean Visible := false;
}
window WnClient
{
  .active false;
  .xleft 87;
  .width 560;
  .ytop 132;
  .height 233;
  .iconic false;
  .title "Einbindung OCX in IDM-Dialog";
  on close
  {
      Ctrl_Grid.connect := false;
      exit();
  }
  child control Ctrl_Grid
  {
    .visible true;
    .active false;
    .xleft 11;
    .width 249;
    .ytop 48;
    .height 169;
    .picture Server_Bild;
    .mode mode_client;
    .name "MSGrid.Grid";
    .connect false;
    rule void Aktivierung
    {
variable integer Cols := 0;
variable integer Rows := 0;

setvalue(Ctrl_Grid, .connect, true, true);
!! Das muss hier abgeprueft werden, da der
!! Verbindungsaufbau
!! u.U. fehlschlagen kann
if (Ctrl_Grid.connect = true) then
    WnClient.title := "Client + Server";
    Cols := atoi(EtCols.content);
    Rows := atoi(EtRows.content);
    !! Setzen der eingestellten Zeilen und Spaltenanzahl
    if (Cols <> 0) then
Ctrl_Grid.Cols := Cols;
    endif
```

```
    if (Rows <> 0) then
Ctrl_Grid.Rows := Rows;
    endif
    Ctrl_Grid.GridLines := CbGridLines.active;
    !! Zum Schluss machen wir den Client noch sichtbar
    Ctrl_Grid.active := true;
else
    print "konnte nicht verbinden";
endif
    }
  }
  child pushbutton PbStart
  {
    .xleft 282;
    .width 90;
    .ytop 32;
    .height 50;
    .text "Start Server";
    on select
    {
Ctrl_Grid:Aktivierung();
    }
  }
  child rectangle
  {
    .xleft 265;
    .width 6;
    .ytop 4;
    .height 369;
  }
  child statictext
  {
    .sensitive false;
    .xleft 276;
    .width 204;
    .ytop 10;
    .text "Client";
  }
  child statictext
  {
    .sensitive false;
    .xleft 7;
    .width 206;
    .ytop 10;
    .height 18;
    .text "Server-Bereich";
```

```
  }
  child pushbutton PbStop
  {
    .xleft 413;
    .width 90;
    .ytop 30;
    .height 50;
    .text "Stop Server";
    on select
    {
Ctrl_Grid.connect := false;
    }
  }
  child checkbox CbGridLines
  {
    .xleft 286;
    .ytop 176;
    .text "Trennlinien";
    .state state_checked;
    on select
    {
Ctrl_Grid.GridLines := this.active;
    }
  }
  child statictext StCols
  {
    .sensitive false;
    .xleft 281;
    .ytop 139;
    .text "Spalten:";
  }
  child statictext StRows
  {
    .sensitive false;
    .xleft 282;
    .ytop 100;
    .text "Zeilen:";
  }
  child spinbox SpCols
  {
    .visible true;
    .xleft 345;
    .width 60;
    .ytop 133;
    .height 25;
    .curvalue 2;
```

```
    on scroll
    {
variable integer Cols := 0;

Cols := atoi(this.EtCols.content);
if (Cols <> 0) then
    Ctrl_Grid.Cols := Cols;
endif
    }
    child edittext EtCols
    {
      .active false;
      .xauto 0;
      .xleft -1;
      .xright 1;
      .yauto 0;
      .ytop 0;
      .ybottom 0;
      .maxchars 2;
      .content "2";
      .multiline false;
      .startsel 0;
      .endsel 1;
      on charinput
      {
  variable integer Cols := 0;

  Cols := atoi(this.content);
  if (Cols <> 0) then
      Ctrl_Grid.Cols := Cols;
  endif
      }
    }
  }
  child spinbox SpRows
  {
    .xleft 345;
    .width 60;
    .ytop 97;
    .curvalue 4;
    on scroll
    {
variable integer Rows := 0;

Rows := atoi(this.EtRows.content);
if (Rows <> 0) then
```

```
    Ctrl_Grid.Rows := Rows;
endif
    }
    child edittext EtRows
    {
      .active false;
      .xauto 0;
      .xleft -1;
      .xright 1;
      .yauto 0;
      .ytop -1;
      .ybottom 1;
      .maxchars 2;
      .content "4";
      .multiline false;
      .startsel 0;
      .endsel 1;
      on charinput
      {
  variable integer Rows := 0;

  Rows := atoi(this.content);
  if (Rows <> 0) then
      Ctrl_Grid.Rows := Rows;
  endif
      }
    }
  }
}
```

Wenn man diesen Dialog startet, erhält man folgendes Bild:

Abbildung 5: *Grid-Control-Beispiel nach dem Start*

Durch die Selektion des "Start Server" Pushbuttons wird der OLE-Server gestartet und aktiviert. Er erscheint dann anstelle des Bilds im Client.

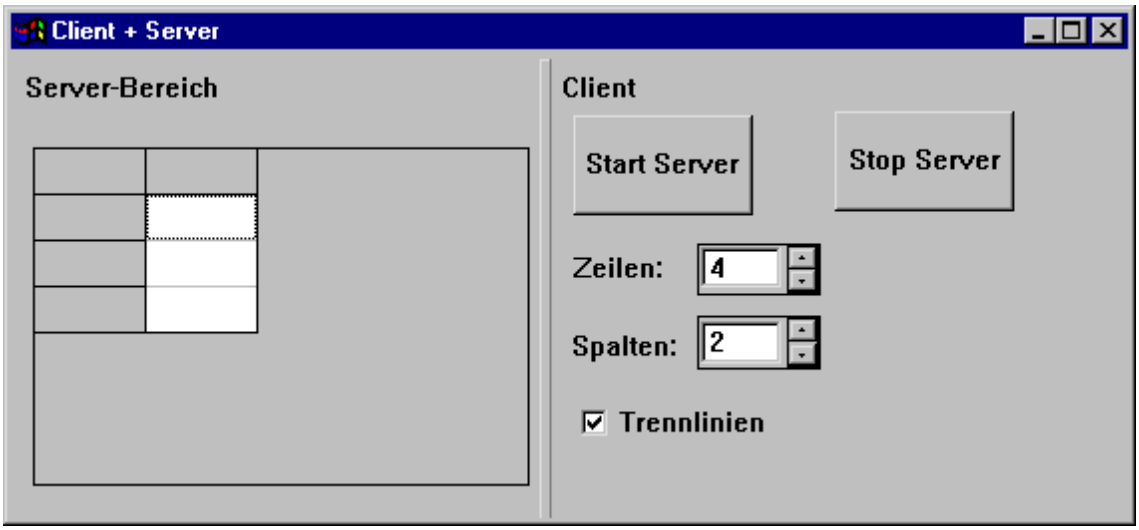

Abbildung 6: *Aktives Grid-Control*

Mit Hilfe der Spinboxes kann man jetzt Attribute des Servers verändern.

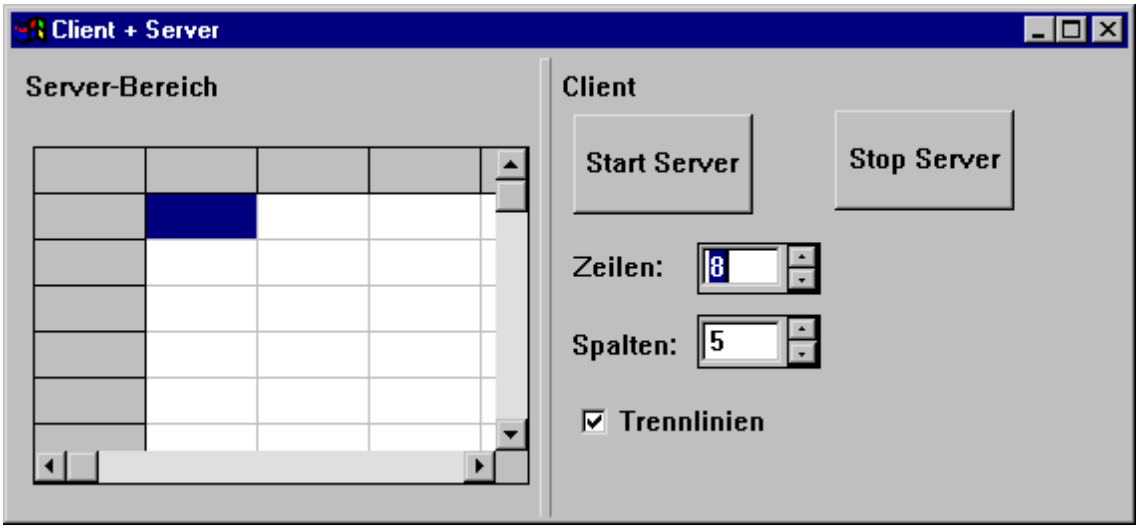

Abbildung 7: *Grid-Control mit veränderten Attributen*

4.6 Verwendung des Internet-Explorers

In diesem Beispiel wird gezeigt, wie der Internet-Explorer von einer Dialog Manager Anwendung aus gesteuert werden kann.

```
dialog Client2
{
}
model control CtTest
{
    .mode mode_client;
    .name "InternetExplorer.Application.1";
    .visible true;
    .active true;
    .connect false;
}
window Window1
{
  .title "Internet-Explorer Fernsteuerung";
  .width 400;
  .height 303;
  object Ctrl := null;

  child pushbutton Pb_Start
  {
    .xleft 37;
    .ytop 43;
    .text "Start";

on select {
```

```
if( Window1.Ctrl = null) then
!! Jetzt wird ein Control generiert, das den Server
!! aufnehmen soll.
   Window1.Ctrl := create(CtTest, this.dialog, true);
endif
!! Aufbau der Verbindung und internes
!! Sichtbarschalten des OLE Servers
!! Damit ist der Server ansprechbar aber noch nicht
!! wirklich sichtbar
Window1.Ctrl.connect := true;
Window1.Ctrl.visible := true;
if(Window1.Ctrl.connect <> true) then
print "ERROR";
print "Window1.Ctrl: " + Window1.Ctrl;
endif
}
  }
  child pushbutton Pb_Quit
  {
    .xleft 143;
    .ytop 41;
    .text "Quit";
on select {
!! Beenden des Internet-Explorers
!! durch Aufruf der Methode
Window1.Ctrl:Quit();
!! Zerstören des Control-Objektes
destroy(Window1.Ctrl);
}
  }
  child pushbutton Pb_visible
  {
    .xleft 34;
    .ytop 126;
    .text "Visible";

on select {
!! Sichtbarschalten des OLE-Servers
Window1.Ctrl.Visible := true;
}
  }
  child pushbutton Pb_navigate
  {
    .xleft 29;
    .ytop 193;
    .text "Navigate";
```

```
on select{
!! Fernsteuerung des Internet-Explorers
Window1.Ctrl:Navigate(Et1.content, nothing,
nothing, nothing, nothing);
}
  }
  child edittext Et1
  {
    .xleft 143;
    .width 175;
    .ytop 192;
  }
}
```

Nach dem Start der Anwendung erscheint folgendes Startfenster:

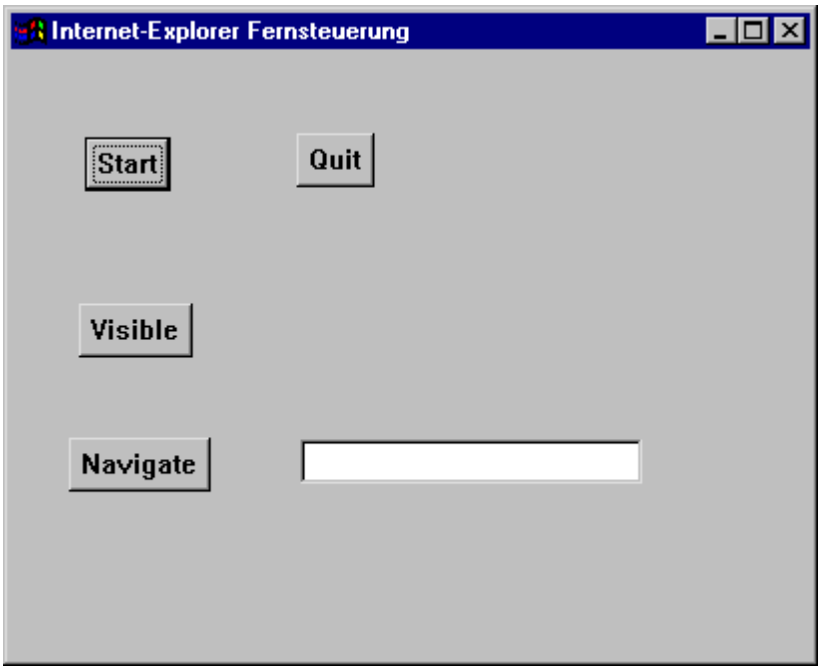

Abbildung 8: *Fenster zur Steuerung des Internet Explorers*

Nach der Selektion des "Start"-Pushbuttons und des "Visible"-Pushbuttons erscheint der Internet-Explorer auf dem Bildschirm.

Abbildung 9: *Gestarteter Internet-Explorer*

Nach der Eingabe einer Internet-Adresse, hier "www.sdr3.de" und der Selektion des "Navigate"- Pushbuttons zeigt der Internet Explorer die entsprechende Seite an.

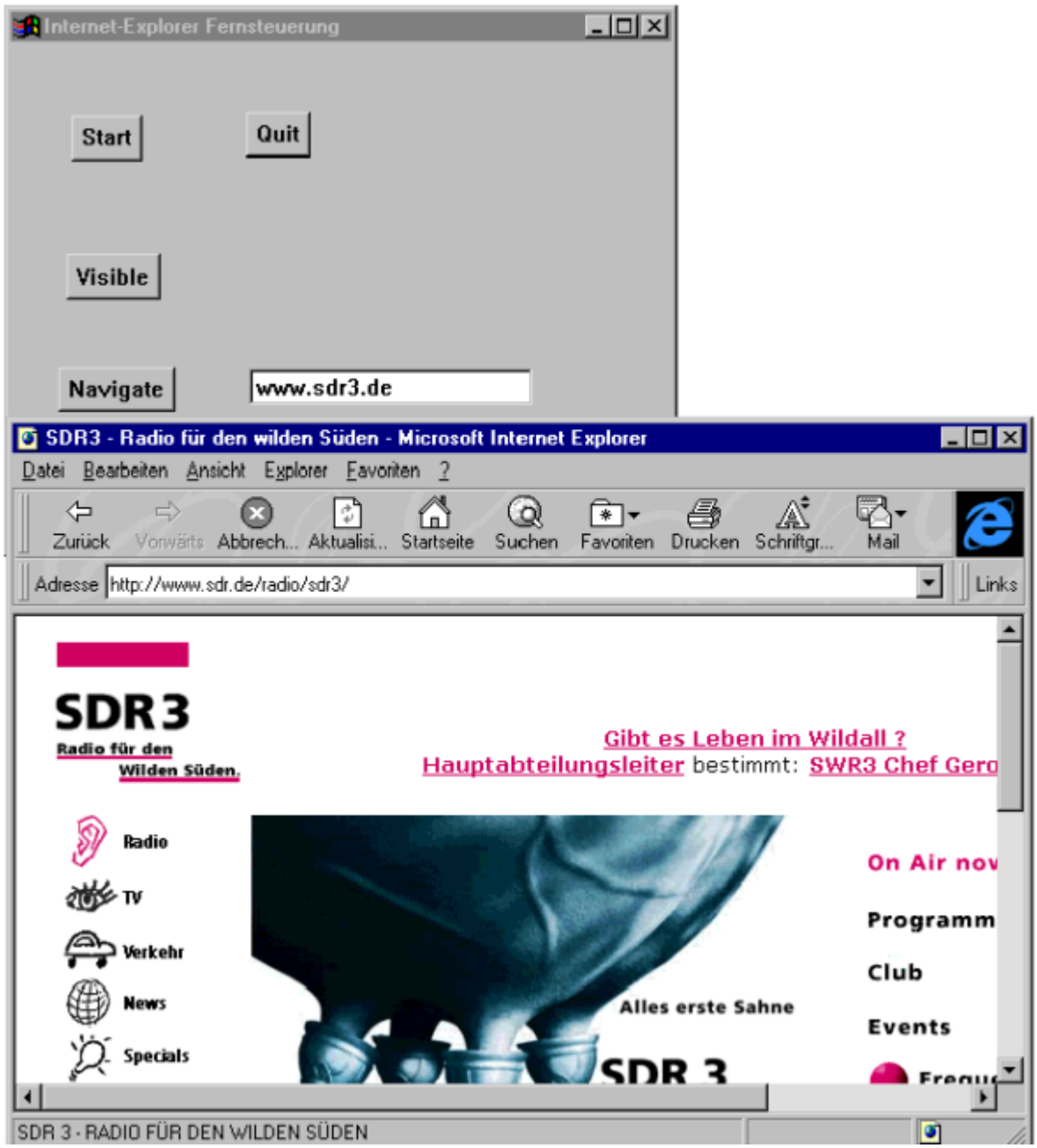

Abbildung 10: *Internet-Explorer nach Ausführung der Navigation*

5 Der Dialog Manager als OLE-Server

Der Dialog Manager kann nicht all seine Objekte, Methoden, Ressourcen, Attribute etc. als Schnittstelle anbieten. Dies würde den Rahmen der Interfaces in OLE sprengen und es ist auch nicht sinnvoll, die Gesamtheit eines Dialogs dem fremden Programm darzubieten. Der Dialog Manager stellt hier die Kommunikationsschnittstelle in Form eines Objektes zur Verfügung. Dieses Objekt gehört der Klasse control an. Die Attribute, Methoden und Ereignisse beschreiben das Interface dieses Controlobjektes.

5.1 Prinzipieller Aufbau des Controls als OLE-Server

Die Klasse Control beschreibt die Schnittstelle zu dem Client. Das Control kann mehrfach instanziiert werden, d.h. es kann mehrmals als verschiedene Instanzen in derselben Anwendung benutzt werden. Deshalb kann das Control-Objekt als Modell nach außen bekannt gegeben werden. Es ist aber auch möglich nur eine einzige Instanz zuzulassen, dann muss das Control nicht als Modell, sondern als normale Instanz deklariert werden. Die nach außen bekannten Control-Objekte dürfen nur im Dialog selbst und nicht etwa in nachzuladenden Modulen definiert werden, da beim Start der Anwendung die zur Verfügung gestellten Controls beim System sofort angemeldet werden müssen. Aus diesem Grund darf das Control-Objekt nicht nachträglich verändert werden, wie zum Beispiel neue Attribute hinzu zu generieren.

Das Control hat benutzerdefinierte Attribute und Methoden. Diese entsprechen den Properties und den Methoden im allgemeinen OLE-Sprachgebrauch. Das Control kann auch genau ein Kind besitzen, dieses ist dann das Objekt, welches im Client erscheinen soll. Ein Control kann auch neben diesem Kind Record-Objekte besitzen, diese sind nach außen nicht bekannt.

Damit ein Control-Objekt und damit sein Dialog als OLE-Server genutzt werden kann, muss dem Attribut .mode der Wert mode_server zugewiesen sein.

Die Attribute beschreiben die Properties. Es ist immer möglich diese zu setzen oder zu lesen. Erlaubt sind skalare, vektorielle und assoziative Attribute mit den Datentypen und Indextypen boolean, string und integer. Das Shadowing von anderen benutzerdefinierten oder vordefinierten Attributen ist ebenso zulässig.

Die Methoden beschreiben, wie der Name schon sagt, die Methoden der Schnittstelle. Diese können von der Anwendung benutzt werden. Bei den möglichen Datentypen bestehen noch Beschränkungen bei Parametern und Rückgabewerten: möglich sind boolean, string und integer.

Das Control verfügt über ein Attribut .picture. Dieses wird, falls der Client dazu in der Lage ist, in dem Client bei nicht aktivem Server dargestellt. Ist kein .picture vorhanden, dann wird nichts im Client angezeigt. Sollte der Client eine Aktivierung und Deaktivierung zulassen, dann wird das Kind des Controls im Client dargestellt, solange das Control InPlace-aktiv ist.

5.2 Eindeutigkeit der Schnittstelle

Damit ein Client einen OLE-Server ansprechen kann, benötigt dieser eine eindeutige Schnittstelle. Diese Schnittstelle ist in der Regel vom Programm und dessen Version abhängig, denn der Client muss immer in der Lage sein, das Programm und seine Schnittstelle eindeutig zu identifizieren. Die Eindeutigkeit dieser Schnittstelle wird beim OLE durch einen 128-Bit Code gewährleistet. Dieser 128 Bit-Code wird UUID genannt.

Diese UUIDs werden

- am Dialog angegeben und
- an jedem als OLE-Server benutzten Control.

Diese Kennung wird dann in der Registry und der Typelibrary benutzt, um die Anwendung und die jeweiligen Controls zu identifizieren. Diese Codes müssen per Hand (guidgen.exe) in das entsprechende Objekt und entsprechende Attribute eingefügt werden. Die Generierung dieser UUID geschieht über das Programm guidgen.exe, das mit dem Microsoft C++ Compiler installiert worden ist.

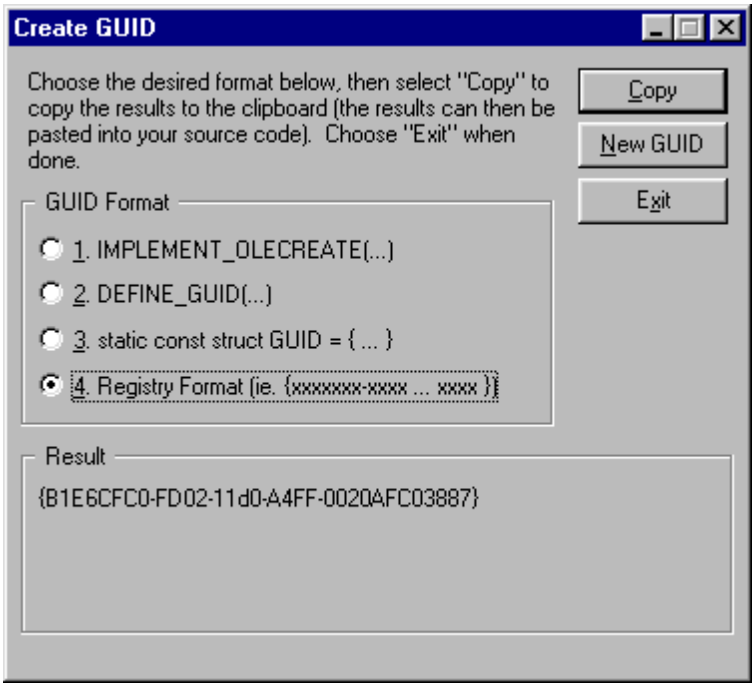

Abbildung 11: *Generierung der GUID*

Beispiel

```
dialog OLEServer
{
.uuid "A5142E00-F4B7-11d0-AA13-00608C63F57F";
}
message Event;

model control Mcontrol
```

```
{
.mode mode_server;
.uuid "A5142E01-F4B7-11d0-AA13-00608C63F57F";
string Property := "";
rule void Method( string Param)
{
}
child record PrivateData
{
}
child groupbox Childobject
{
}
}
```

5.3 Einfach- und Mehrfachbenutzung eines OLE-Servers

Je nachdem, wie man das Control-Objekt im Dialog Manager definiert, kann ein OLE-Server ein oder mehrere Clients gleichzeitig bedienen. Wird das Control-Objekt als Modell definiert, so können sich mehrere Clients auf diesen Server verbinden. Wird das Control-Objekt jedoch als Instanz definiert, so kann sich jeweils nur ein einziger Client auf den Server verbinden. Für den nächsten Client wird dann ein neuer Serverprozess gestartet.

Beide Methoden haben ihre Vor- und Nachteile, die hier kurz erläutert werden sollen:

- Wird für jeden Client ein eigener Serverprozess gestartet, so arbeiten u.U. mehrere Prozesse parallel nebeneinander, die alle im Hauptspeicher gehalten werden müssen. Daher ist in diesem Fall mit einem deutlich höheren Hauptspeicherbedarf zu rechnen. In diesem Fall muss aber bei der Programmierung nichts besonderes beachtet werden. Soll der Dialog auch als "Standalone"-Dialog ohne OLE-Client ablaufen können, so muss in der Dialog-Start-Regel das Attribut ".mode" des Control-Objekts so umgesetzt werden, dass keine Registrierung des OLE-Servers im System erfolgt. Dazu muss diesem Attribut der Wert "mode_none" zugewiesen werden. Dieses darf ausschließlich in der Dialog-Start-Regel erfolgen, danach ist der Server im System registriert und Clients können sich mit ihm verbinden.
- Wird ein Server so betrieben, dass er mehrere Clients bedienen kann, muss die gesamte Programmierung darauf abgestimmt werden, dass jederzeit ein Wechsel des aktiven Clients erfolgen kann. Jederzeit heißt in diesem Fall, jedes mal wenn der Dialog Manager in die Verarbeitung von Ereignissen eintritt, können neue OLE-Aufrufe zur Bearbeitung aktiviert werden. Daher darf in diesem Fall nicht mit globalen Variablen (weder in der Regelsprache noch in angebundenen Programmen) gearbeitet werden. Der Server darf sich in diesem Fall erst dann beenden, wenn alle seine Clients beendet worden sind. Aus diesem Grund zählt man hier in der Regel die aktiven Verbindungen zu Clients mit und beendet den Server, wenn diese Anzahl wieder 0 geworden ist.

Beispiel

```
model control MyControl
```

```
{
  integer Count := 0;

  on start
  {
    MyControl.Count := MyControl.Count + 1;
  }

  on finish
  {
    MyControl.Count := MyControl.Count - 1;
    if MyControl.Count=0 then
      exit();
    endif
  }
}
```

In diesem Beispiel kann der OLE-Server beliebig oft gleichzeitig benutzt werden. Jedes mal, wenn ein neuer Benutzer hinzukommt, wird der Count entsprechend erhöht, wenn ein Benutzer die Verbindung beendet, wird der Count erniedrigt. Wenn der Count wieder bei 0 angekommen ist, wird der Server beendet.

```
control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
```

Dieser OLE-Server kann nur einen Client gleichzeitig bedienen. Wenn ein zweiter Client die Dienste dieser Servers in Anspruch nehmen möchte, so wird ein neuer Serverprozess gestartet.

5.4 Attribute

Die benutzerdefinierten Attribute eines Controlobjektes entsprechen den sogenannten Properties aus der IDispatch-Namenskonvention. Eingebaute Attribute des Objektes sind nicht nach außen sichtbar. Sie können also nicht direkt von außen gelesen oder gesetzt werden, aber dies kann durch die explizite Bereitstellung über benutzerdefinierte Attribute oder Methoden geschehen. Nach außen sichtbaren Attribute dürfen nicht zur Laufzeit des Programms generiert werden, sondern müssen statisch definiert werden. Aus diesen Attributen wird die eigentliche Schnittstellen-Information generiert, die OLE-Clients benötigen, um den Server auszurufen.

Beispiel

```
control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
  .width 320;
```

```
  .height 200;
  integer I := 123;
  string S := "Dialog Manager";
  boolean B := true;
```

In diesem Beispiel können die Attribute "I", "S" und "B" vom Client abgefragt werden, nicht aber die Attribute ".heigth" und "width".

5.5 Methoden

Benutzerdefinierte Methoden des Control-Objekts können vom OLE-Client angesprochen werden, die eingebauten Methoden wie etwa :insert oder :delete stehen nicht automatisch dem Benutzer des Control-Objektes zur Verfügung.

Beispiel

```
control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  rule void M1
  {
  }
  rule integer M2 (integer I input)
  {
    return 17;
  }
```

In diesem Beispiel können die Methoden "M1" und "M2" vom Client aufgerufen werden.

5.6 Benachrichtigungen (Notifications)

Bei der Werteänderung eines benutzerdefinierten Attributes an einem Control-Objekt im Server-Modus wird eine PropertyChanged Notification an den Client gesendet. Ein nicht mit Dialog Manager programmierter Client muss dazu Notifications über das Standardinterface IPropertyNotifySink-Interface unterstützen.

Beispiel

```
control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
  .width 320;
  .height 200;
  integer I := 123;
  string S := "Dialog Manager";
```

```
  boolean B := true;
```

In diesem Beispiel erhält der Client Benachrichtigungen, wenn sich die Attribute "I", "S" und "B" ändern, nicht aber für die Attribute ".heigth" und "width".

5.7 Ereignisse

Ein OLE-Server kann Ereignisse an seinen Client versenden. Im Dialog Manager existiert hierfür die Ressource ***message*** und wird am Dialog oder Modul definiert. Der Entwickler muss bei dem Controlobjekt angeben, welche Messages versendet werden können. Das Messageobjekt steht auch anderen Objektklassen zur Verfügung und entspricht dem bisher bekannten extevent-Ereignis.

5.7.1 Die Ressource message

Ereignisse müssen vor der Benutzung inklusive Parameter definiert werden. Dazu dient die Message-Ressource. Sie ist wie folgt definiert:

```
<message> ::= {export | reexport} 'message' <label> { <messageSpec> };
<messageSpec> ::= '(' { <messageArg> [ ',' <messageArg> ] } ')'
<messageArg>  ::= <datatype> [<label>]
```

Damit die nach außen bekanntzugebenden Events auch zum Client weitergeleitet werden, sind diese am Control-Objekt zu definieren. Dazu dient das Feld-Attribut .message[]. Dieses Feld kann ähnlich wie .content[] behandelt werden, d.h. mit setvalue /getvalue angesprochen bzw. mittels .count[] nach der Größe gefragt werden.

Die Message-Ressource ist auch unabhängig von OLE-control nutzbar für die Definition von "benannten" Events. Das Event kann jetzt einfach mittels eines sendevent() an das Control-Objekt geschickt werden. Nur Events, die im .message[]-Feld definiert sind, werden an den Client geschickt, alle anderen werden wie externe Ereignisse im Dialog selber verarbeitet.

Einschränkungen

Die OLE-Option des IDM unterstützt nur die Datentypen integer, string und boolean. Zwar sind auch andere Typen an der allgemeinen Message-Ressource definierbar, aber es kann keine Typelibrary dafür generiert werden. Damit können diese Datentypen nicht zum Datenaustausch per OLE genutzt werden.

Beispiel

```
dialog D
message Msg(integer I, string S);
model control MC
{
  .message[1] Msg;
  :
  pushbutton Pb
  {
    on select
```

```
    {
      sendevent(this.control,Msg,1998,"Aktion ausloesen");
    }
  }
}
```

Anmerkungen und Einschränkungen

Der Client hat hierzu das Event-Interface auszuimplementieren und über die Standardprozedur für NotifySinks anzubinden. Der Client kann entweder dynamisch über die Type-Library des Servers den entsprechenden Sink generieren, oder statisch einen Sink für ein bekanntes Control verwenden. Werden unerwartete neue Events versendet, kann es zu Abstürzen kommen! Deshalb sollte man bei neuen Events dem Control eine neue UUID geben.

5.8 Zugriff auf Attribute beliebiger Objekte

Von einer fremden Anwendung aus kann nur auf die benutzerdefinierten Attribute des Controls selbst zugegriffen werden. Oftmals ist es aber sinnvoll auf bestimmte Attribute des Kindobjektes oder dessen Kinder zuzugreifen und diese zu manipulieren. Man könnte dies über Methoden programmieren oder über den shadow-Mechanismus des Dialog Managers. Hierbei muss bei der Modellbildung darauf geachtet werden, dass das Attribut der Instanz und nicht das des Modells "geshadowed" wird.

Beispiel

```
dialog D

model control Mcontrol
{
  .mode mode_server;
  string Visible shadows instance GrpBox.visible;
  string Content shadows instance Et.content;
  child groupbox GrpBox
  {
    child edittext Et
    {
    }
  }
}
```

Wird in diesem Beispiel das Control von außen benutzt, dann wird eine Instanz dieses Modells generiert. Beim Zugriff auf .Visible wird der zu setzende Wert auf die groupbox GrpBox umgeleitet und diese dadurch im Container sichtbar, aber nur wenn dieser InPlace-aktiv ist. Mit dem Attribut .Content kann der .content des Textfeldes Et gesetzt und auch gelesen werden.

Als Alternative kann man auch Methoden einsetzen.

Beispiel

```
dialog D
```

```
model control Mcontrol
{
  .mode mode_server;
  rule boolean GetVisible()
  {
    return this.GrpBox.visible;
  }
  rule void SetVisible( boolean Value )
  {
    this.GrpBox.visible := Value;
  }
  rule string GetContent()
  {
    return this.GrpBox.Et.content;
  }
  rule SetContent( string Value )
  {
    this.GrpBox.Et.content := Value;
  }
  child groupbox GrpBox
  {
    child edittext Et
    {
    }
  }
}
```

Mit diesem Beispiel erreicht man den gleichen Effekt. Hier kann man auch erkennen, warum shadows instance benutzt werden muss. In den Methoden wird über this die Instanz des Controls angesprochen und auch dessen Kinder. Würde man this weglassen, hätte dies den gleichen Effekt wie das Schlüsselwort instance am shadows wegzulassen, es würden die Objekte des Modells referenziert und nicht die der aktuellen Instanz.

5.9 Generierung der Schnittstelleninformation

In diesem Kapitel werden die Schritte vorgestellt, die notwendig sind um aus einem Dialog mit einem als *ole_server* definierten ***Control*** einen OLE-Server zu bauen.

5.9.1 Generieren der idl- und reg-Dateien

Mit der Option **-writeole <Basisname>** des IDM-Simulationsprogramms **idm.exe** werden aus einem Dialogskript, das als OLE-Server definierte ***Controls*** enthält, die zur Registrierung notwendigen Dateien erzeugt. Es werden eine **idl**- und eine **reg**-Datei generiert.

Mit folgenden zusätzlichen Optionen kann die Generierung beeinflusst werden:

- **+localserver <Pfad des eigentlichen Executables>**
- **+helpdir <Verzeichnis der Hilfedateien>**
- **+typelib <Pfad der Typbibliothek>**
- **+deficon <Pfad des Standardsymbol>**
- **+proxy <Name des Proxy-Stubs>**
- **-userregistry** (ab IDM-Version A.06.01.g)
  Registrierung nur für den aktuellen Benutzer (unter HKEY_CURRENT_USER in der Windows-Registry)

Folgende Standardeinträge werden mit `idm.exe <Dialog> -writeole <Datei>` in die Registrierungsdatei **<Datei>.reg** aufgenommen, wenn sie nicht durch eine der oben genannten Optionen überschrieben wurden:

```
LocalServer32 = "<Pfad zu idm.exe> <Dialog> /Automation"
TypeLib = "<Datei>.tlb"
HelpDir = ""
DefIcon = ""
```

Der Name des Proxy-Stubs wird standardmäßig auf **<Basisname>.dll** gesetzt.

5.9.2 Registrierung des Servers

Ein Doppelklick auf die **reg**-Datei (sofern **reg**-Dateien im System korrekt verknüpft sind) oder der Aufruf von **regedit.exe** mit der generierten **reg**-Datei als Argument registriert das Control beim System.

Beispiel

```
server.reg server.idl: server.dlg
  $(IDM) server.dlg \
  -localserver "$(IDM) server.dlg -IDMtracefile server.log" \
    -writeole server
  regedit server.reg
```

In diesem Beispiel ist auch die Kommandozeile für den Server angegeben, sodass dieser immer ein Tracefile erzeugt. Anschließend wird der Server noch im System registriert.

5.9.3 Weiterverarbeitung der idl-Datei

Die mit Hilfe der Option **-writeole** erzeugte **idl**-Datei muss mit dem MIDL-Compiler weiterverarbeitet werden, der mit MICROSOFT VISUAL STUDIO ausgeliefert wird. Der MIDL-Compiler erzeugt aus der **idl**-Datei eine Proxy-DLL, die das „OLE-Marshalling“ des Interfaces übernimmt. Als Standardpfad wird dafür der Ausgabename von **-writeole** mit der Erweiterung „.dll“ genommen, dies kann aber mit der Option **+proxy** überschrieben werden.

Beispiel

```
server.tlb server_p.c server_i.c server.h dlldata.c: server.idl
  midl /ms_ext /app_config /c_ext /tlb server.tlb /Zp1 \
    /env win32 /Os server.idl
```

Die mit dem MIDL-Compiler generierten Dateien müssen dann mit dem C-Compilers übersetzt und zu einer DLL gelinkt werden.

5.10 Beispiel

In diesem Beispiel wird ein Control definiert, das eine Methode „Answer“ und Attribute „Visible“ und „Content“ bereitstellt. Diese Attribute sind als Verweise auf Attribute eines Kindes des Controls definiert. Auf diese Art und Weise kann der Client auch Attribute abfragen und ändern, die nicht direkt am Control definiert sind.

```
dialog ExampleControl
{
  .uuid "FC4D3263-F6DC-11d0-AA13-00608C63F57F";
}

model control MDeepThought
{
  .mode mode_server;
  .uuid "FC4D3264-F6DC-11d0-AA13-00608C63F57F";

  boolean Visible shadows instance GrpBox.visible;
  string  Content shadows instance EQuestion.content;

  rule integer Answer( string Question )
  {
    this.GrpBox.SAnswer.text := "42";
    return 42;
  }

  child groupbox GrpBox
  {
    child edittext   EQuestion {}
    child statictext SAnswer {}
  }
}
```

Dieses Control wird mit Hilfe der Option **-writeole** und anschließendem Aufruf des Programms **regedit.exe** registriert. Danach steht den Anwendungen ein Control zur Verfügung, das im Control eine Groupbox mit einem Eingabefeld und einem Text darstellen kann, aber nur solange es „InPlace“ aktiviert ist. Die Antwort ist trivialerweise immer 42.

5.11 Beispielhafte Integration eines Servers in Word 7.0

In Word 7.0 soll mit Hilfe des Visual Basic for Applications ein OLE-Server integriert werden, der mit Dialog Manager gebaut worden ist. Der Server soll nichts weiteres machen, als Methoden bereitstellen, die der Client aufrufen kann.

5.11.1 Der Server-Dialog

Der Server besteht aus einer Listbox, die sich innerhalb des Bereichs des Client darstellen kann. Dazu wird das Control-Objekt wie bisher definiert und die Listbox als dessen Kind.

```
dialog IDM_Server
{
  .uuid "523E0007-F149-11d0-91F6-00A02444C34E";
}
tile TiCtrl "IDM_IMAGES:client.bmp";
variable integer Refcount := 0;
model control Serv
{
  .mode mode_server;
  .uuid "523E0008-F149-11d0-91F6-00A02444C34E";
  .interfaceid "523E0009-F149-11d0-91F6-00A02444C34E";
  .picture TiCtrl;
  rule void Beep
  {
      this.Lb:Msg("Beep()");
      beep();
  }
  rule void PPrint (string S input)
  {
      this.Lb:Msg((("Print(" + S) + ")"));
  }
  rule void KKill
  {
      exit();
  }
  on start
  {
      Refcount := (Refcount + 1);
      this.Lb:Msg(("Refcount: " + itoa(Refcount)));
  }
  on finish
  {
      Refcount := (Refcount - 1);
      this.Lb:Msg(("Refcount: " + itoa(Refcount)));
      if (Refcount <= 0) then
  exit();
```

```
      endif
  }
  child listbox Lb
  {
    .width 300;
    .height 400;
    .firstchar 1;
    rule void Msg (string S input)
    {
this.content[(this.itemcount + 1)] := S;
updatescreen();
    }
  }
}
```

Dieser Server wird im System registriert, sodass das OLE-Simulationsprogramm "idmole" diesen Dialog lädt. Danach kann die Programmierung im Word 97 erfolgen.

5.11.2 Implementierung des Clients

Zur Implementierung des Client wird das Word 97 genommen. Es werden Pushbuttons zum Starten und Beenden des OLE-Servers angelegt. Zusätzlich werden zwei Pushbuttons angelegt, die Methoden im Server aufrufen sollen.

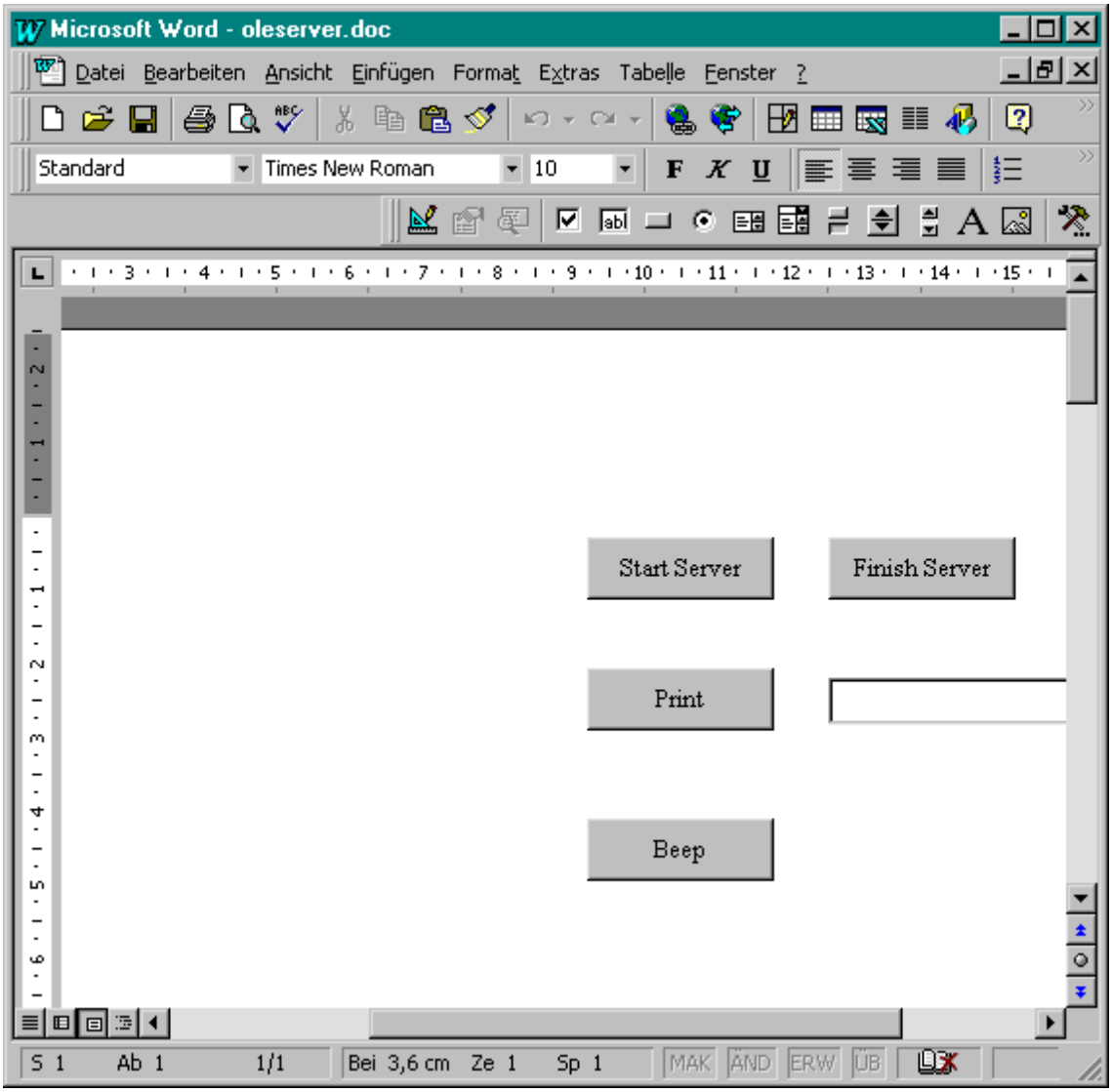

Abbildung 12: *Clientdefinition in Word 97*

Anschließend erfolgt die Programmierung in Visual Basic for Applications. Diese sieht dann wie folgt aus:

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "ThisDocument"
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Public OLE As Object

Private Sub EtPrint_Change()

End Sub
```

```
Private Sub PbEnde_Click()
OLE.OLEFormat.Object.KKill
End Sub

Private Sub PbPrint_Click()
OLE.OLEFormat.Object.PPrint EtPrint.Text
End Sub

Private Sub PbBeep_Click()
OLE.OLEFormat.Object.Beep
End Sub

Private Sub PbStart_Click()
Set OLE = ActiveDocument.Shapes.AddOLEObject(ClassType:="IDM_Server.Serv",
Width:=250, Height:=200)

End Sub
```

5.11.3 Arbeiten mit dem Server

Nach der Selektion des Start-Pushbuttons wird der Server aktiviert und stellt sich im Client dar. Durch die Selektion der Pushbutton "Beep" und "Print" können die entsprechenden Methoden im Server aufgerufen werden.

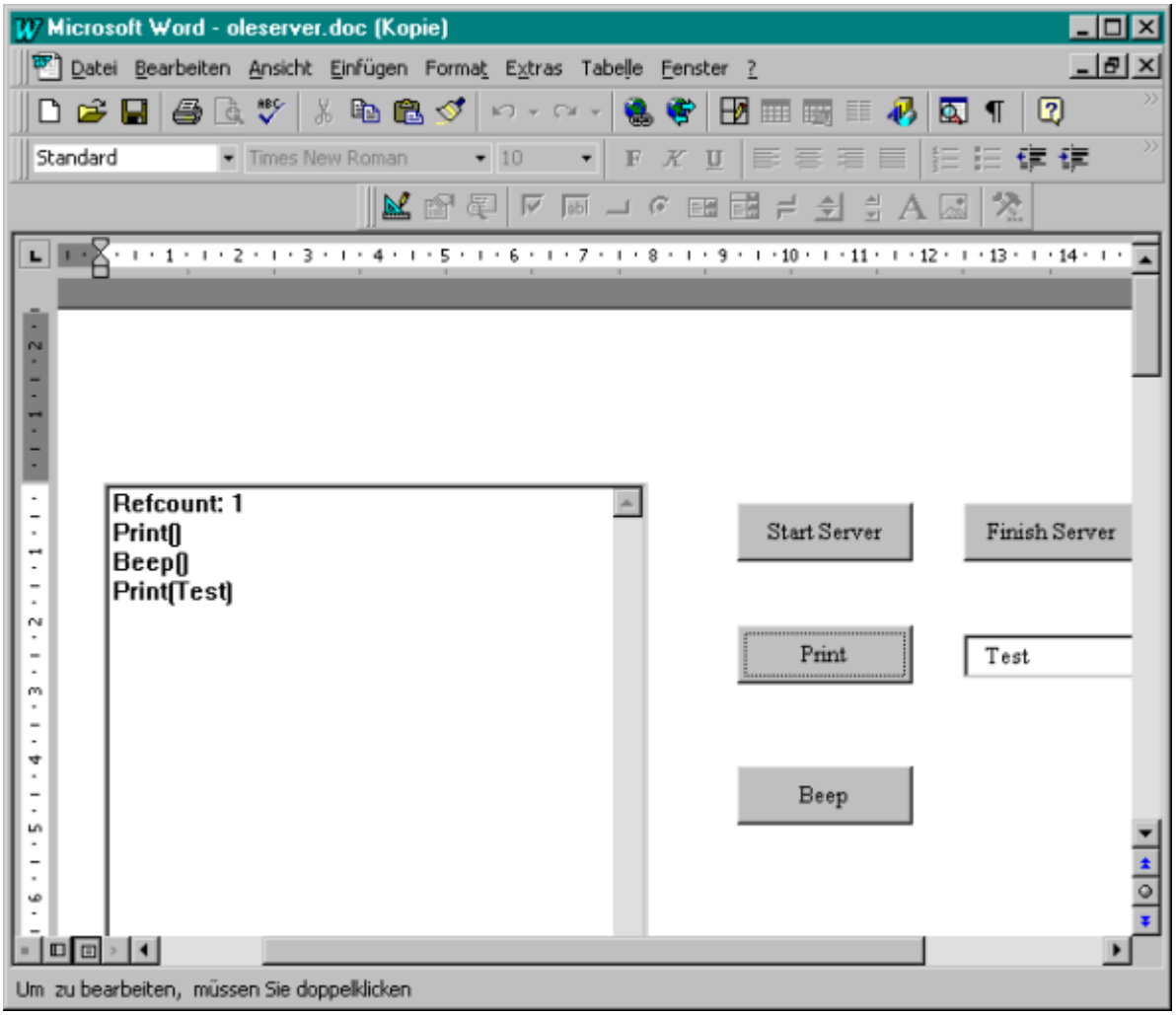

Abbildung 13: *Word 7.0 Client mit integriertem Server*

5.12 Zusammenfassung für die Bereitstellung eines OLE-Servers

Folgende Schritte müssen bei der Implementierung eines OLE-Servers durchgeführt werden:

1. Implementierung eines ***Control***-Objekts.
   - » Als Modell: Der Server kann mehrere Clients gleichzeitig bedienen.
   - » Als Instanz: Je Client wird ein eigener Serverprozess gestartet.
2. Vergabe einer eindeutigen UUID mit Hilfe des Programms **guidgen.exe**.
3. Generierung der **idl**- und **reg**-Dateien mit Hilfe der Option **-writeole**.
4. Registrierung des Servers im System mit Hilfe des Programms **regedit.exe**.
5. Generierung der notwendigen C- und TBL-Dateien mit Hilfe des MIDL-Compilers.
6. Übersetzen der generierten C-Dateien.
7. Linken einer DLL, die zum Aufruf des Servers vom Client aus benötigt wird.

Bei einer Installation des OLE-Servers auf einem anderen Rechner muss die Registrierung erneut durchgeführt und gegebenenfalls angepasst werden, sonst kann der OLE-Server nicht benutzt werden.

6 Implementierung eines Servers und eines Clients

In diesem Beispiel wird im Dialog Manager sowohl ein OLE-Client als auch OLE-Server implementiert und der Einsatz der unterschiedlichen Möglichkeiten zum Datenaustausch aufgezeigt.

6.1 Aufbau des Clients

Der Client besteht aus einem Notebook und einem Edittext. Über das Notebook wird der Server gesteuert, und in einer Listbox werden die Ergebnisse und Antworten des Servers sowie dessen Nachrichten mitprotokolliert.

Über die erste Seite des Notebooks wird der Server aktiviert und deaktiviert. Auf der zweiten Seite können Eigenschaften des Servers erfragt bzw. gesetzt werden. Auf der dritten Seite können Methoden des Servers aufgerufen werden.

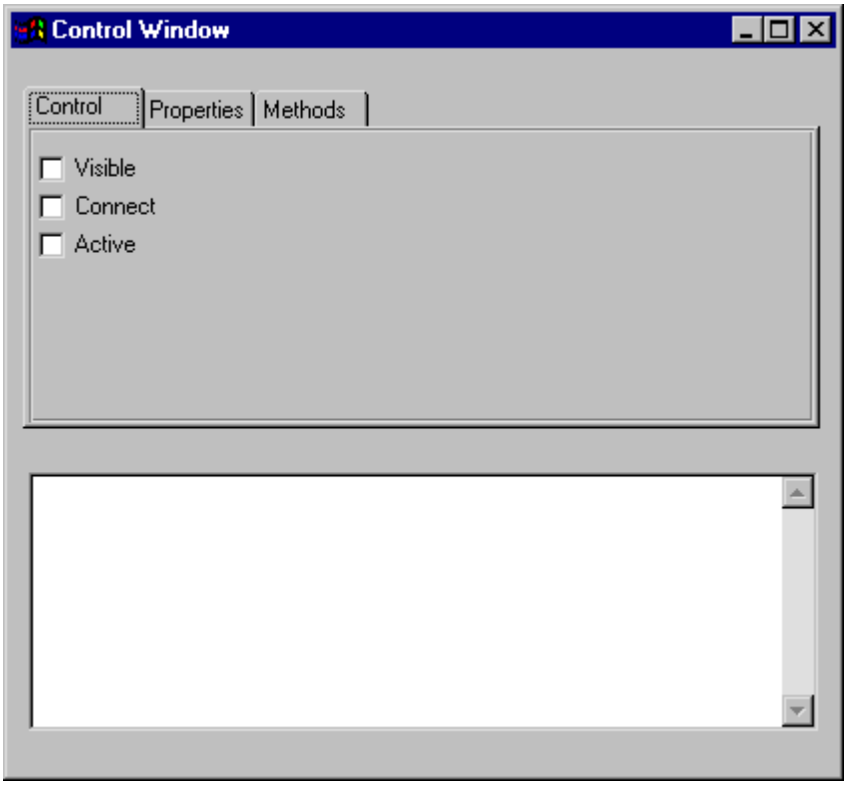

Abbildung 14: *Das Client-Fenster*

Das Control-Objekt ist dabei wie folgt definiert:

```
window WiControl
{
  .title "OLE Control";
  .width 60;
```

```
  .height 20;
  .xleft 450;
  .visible := false;

  control C
  {
    .mode mode_client;
    .name "IdmTest.PropMethEvent";
.width 300;
.height 300;
  }
}
```

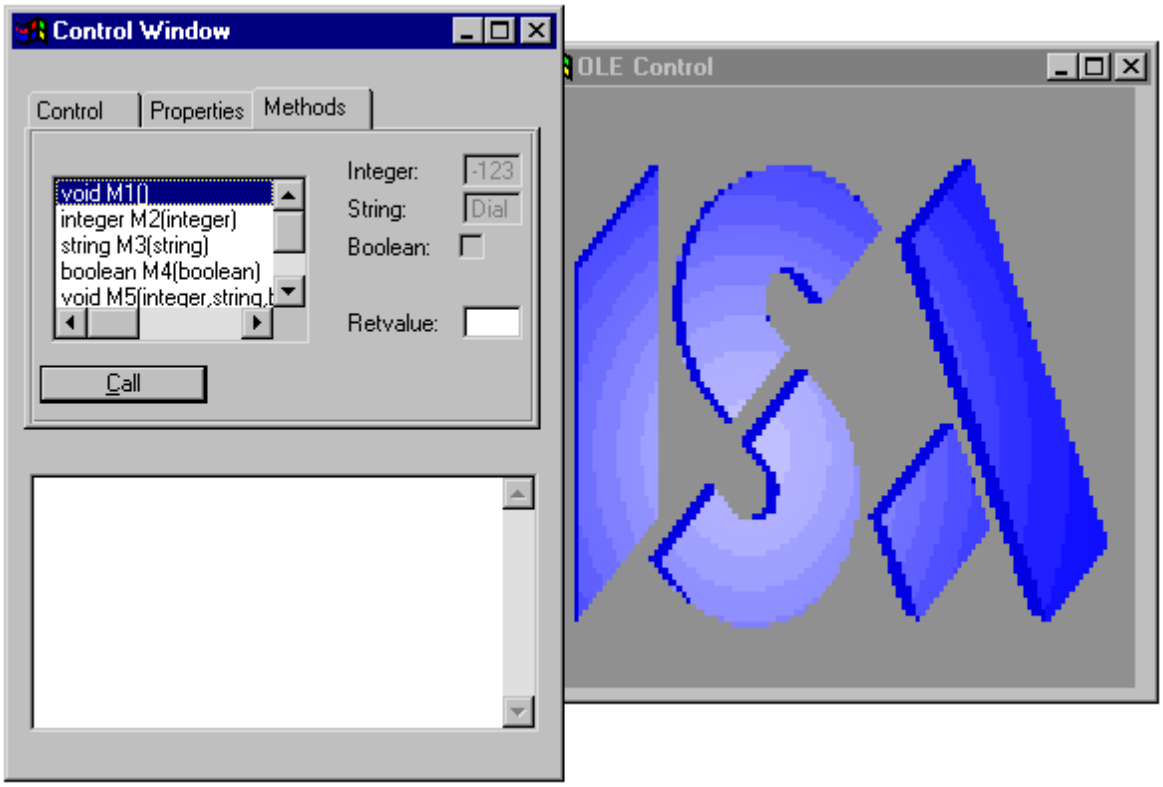

Abbildung 15: *Zustand der Anwendung nach Start des Servers*

6.1.1 Die Aktivierung des Servers

Durch die Selektion der "visible"-Checkbox wird der Server gestartet. Durch die Selektion der "connect"-Checkbox wird eine Verbindung zum Server aufgebaut und durch die Selektion der "active" Checkbox wird der Server aktiviert und steht für die Kommunikation zur Verfügung.

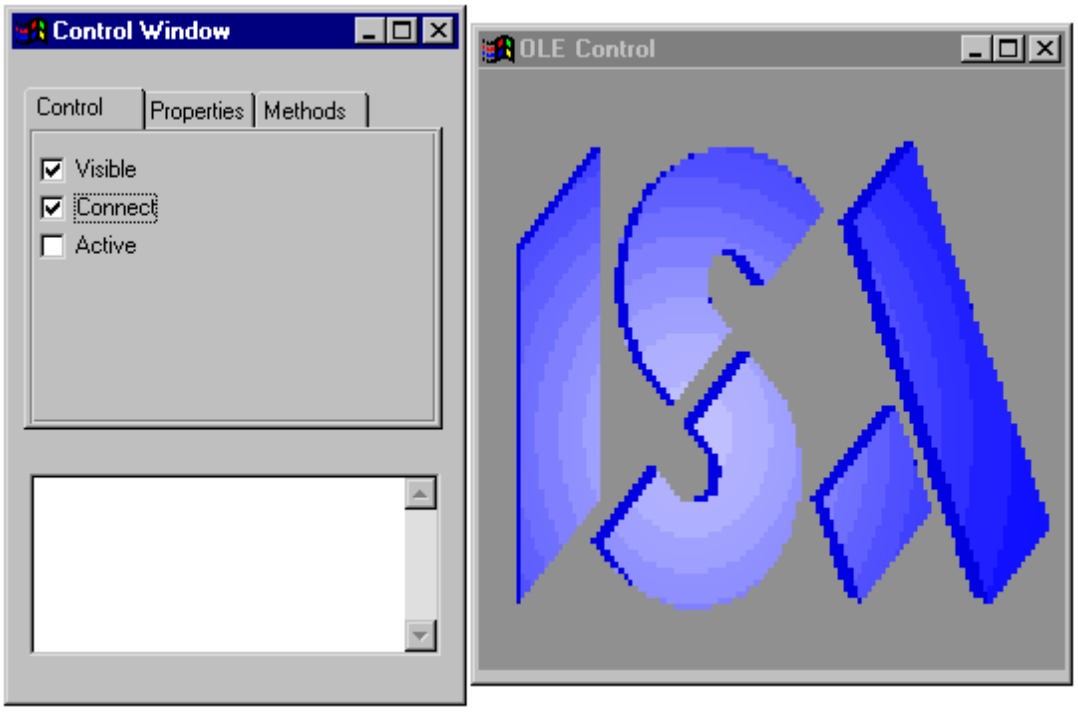

Abbildung 16: *Server nach dem Verbindungsaufbau*

Die notwendigen Regeln sehen dabei wie folgt aus:

Zuerst wird ein generelles Modell definiert, um die Regeln zu definieren:

```
model checkbox MCb
{
  boolean Value := false;

  on activate,deactivate
  {
    this.Value := this.active;
if this.Value <> this.active then
  Info("changing "+this.text+" FAILED");
endif
  }
}
```

Die eigentliche Definition der Objekte sieht dann wie folgt aus:

```
  child notepage NpControl
  {
     .active true;
     .title "Control";
     MCb CbVisible
{
   .text "Visible";
   .Value shadows WiControl.visible;
}
```

```
     MCb CbConnect
{
   .ytop 1;
   .text "Connect";
   .Value shadows WiControl.C.connect;
}
     MCb CbActive
{
   .ytop 2;
   .text "Active";
   .Value shadows WiControl.C.active;
}
  }
```

6.1.2 Abfragen und Setzen von Werten im Server

Zum Abfragen und Setzen von Werten im Server muss man im Client auf die zweite Seite des Notebooks umschalten. Hier hat man dann die Möglichkeit, sowohl Attribute abzufragen als auch zu setzen.

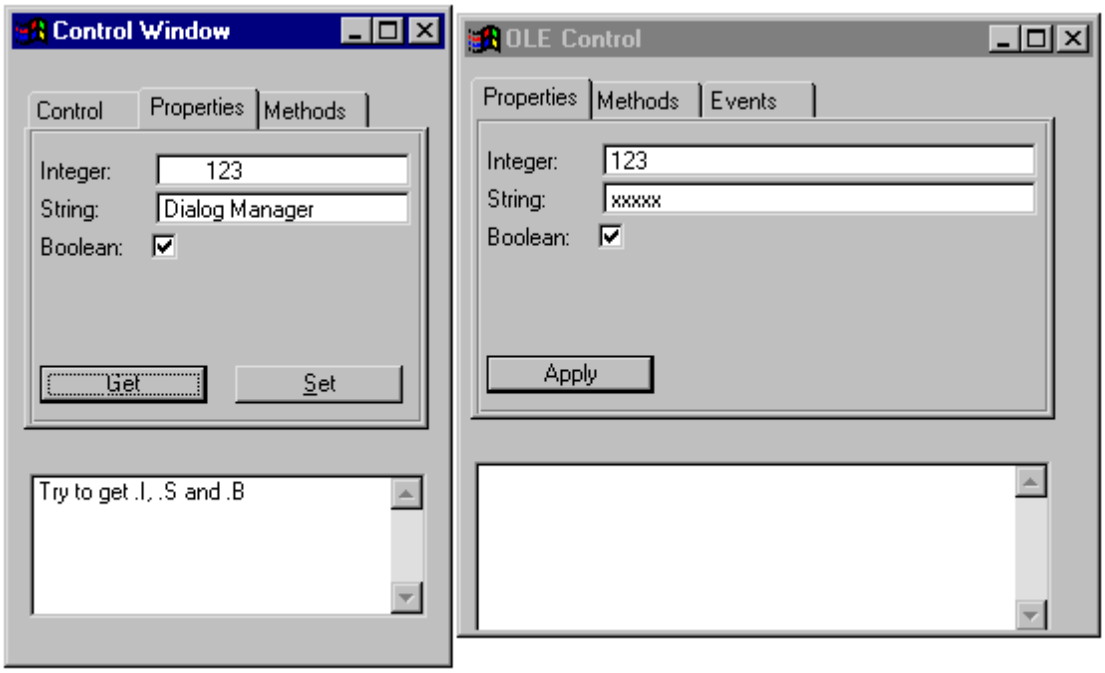

Abbildung 17: *Abfragen und Setzen vom Werten im Server*

Die dazu notwendigen Regeln im Client sehen wie folgt aus:

```
child pushbutton PbGet
{
  .yauto -1;
  .height 1;
```

```
  .text "&Get";
  .defbutton true;
  on select
  {
Info("Try to get .I, .S and .B");
    if fail(this.parent.Integer.Value := this.window.C.I) then
  Info("getvalue of .I FAILED");
endif
    if fail(this.parent.String.Value := this.window.C.S) then
  Info("getvalue of .S FAILED");
endif
    if fail(this.parent.Boolean.Value := this.window.C.B) then
  Info("getvalue of .B FAILED");
endif
  }
}
child pushbutton PbSet
{
  .yauto -1;
  .height 1;
  .text "&Set";
  .xleft 14;
  on select
  {
Info("Try to set .I, .S and .B");
    if fail(this.window.C.I := this.parent.Integer.Value) hen
  Info("setvalue of .I FAILED");
endif
    if fail(this.window.C.S := this.parent.String.Value) then
  Info("setvalue of .S FAILED");
endif
    if fail(this.window.C.B := this.parent.Boolean.Value) then
  Info("setvalue of .B FAILED");
endif
  }
}
```

Wie man hier erkennen kann, wird in den Regeln auf die Attribute "I", "S" und "B" des Control-Objektes zugegriffen, die dort aber nicht definiert sind. Damit werden diese Attribute an den Server übergeben und dort abgearbeitet.

6.1.3 Die Reaktion auf Ereignisse

Wenn der Server dem Client Nachrichten schicken kann, müssen diese beim Client abgefangen werden. Dann kann der Client auf diese Ereignisse reagieren. In der Regelsprache kommen diese

Ereignisse als "externe Ereignisse" an und müssen dementsprechend programmiert werden. Falls diese Ereignisse vom Server Parameter übergeben bekommen, müssen diese als Parameter der Regel definiert werden.

```
control C
{
on extevent "Msg1"
{
  Info("extevent Msg1()");
}
on extevent "Msg2" (integer I)
{
  Info("extevent Msg2("+I+")");
}
on extevent "Msg3" (string S)
{
  Info("extevent Msg3("+S+")");
}
on extevent "Msg4" (boolean B)
{
  Info("extevent Msg4("+B+")");
}
on extevent "Msg5" (integer I, string S, boolean B)
{
  Info("extevent Msg5("+I+","+S+","+B+")");
}
on extevent "Msg6" (integer P1, string P2, boolean P3,
integer P4, string P5, boolean P6)
{
  Info("extevent Msg6("+P1+","+P2+","+P3+","+P4+","
+P5+","+P6+")");
}
}
```

6.1.4 Die Reaktion auf Benachrichtigungen (Notifications)

Wenn der Server dem Client Benachrichtigungen (Notifications) schicken kann, können diese beim Client abgefangen werden. Dann kann der Client auf diese Änderungen von Werten reagieren. In der Regelsprache kommen diese Benachrichtigungen als "changed Ereignisse" an und müssen dementsprechend programmiert werden.

```
control C
{
on .I changed
{
  Info(".I changed");
if fail(NpProperties.Integer.Value := this.I) then
```

```
    Info("getvalue of .I FAILED");
  endif
}
on .S changed
{
  Info(".S changed");
  if fail(NpProperties.String.Value := this.S) then
    Info("getvalue of .S FAILED");
  endif
}
on .B changed
{
  Info(".B changed");
  if fail(NpProperties.Boolean.Value := this.B) then
    Info("getvalue of .B FAILED");
  endif
}
}
```

6.1.5 Aufruf von Methoden im Server

Damit vom Client Methoden des Servers aufgerufen werden können, muss im Client lediglich der Aufruf der Methode mit den entsprechenden Parametern implementiert werden. Die Methode selber darf nicht definiert werden.

Der Regelcode für den Aufruf von Methoden am Server sieht wie folgt aus:

```
child pushbutton PbCall
  {
    .yauto -1;
    .height 1;
    .text "&Call";
    .defbutton true;
    integer I shadows instance NpMethods.Integer.Value;
    boolean B shadows instance NpMethods.Boolean.Value;
    string S shadows instance NpMethods.String.Value;
    on select
    {
      case this.parent.LbMethods.activeitem
    in 1:
      Info("call :M1();");
      this.parent.Retval.Value := "";
      if fail(this.window.C:M1()) then
      Info("FAILED");
      endif
    in 2:
```

```
    Info("call :M2("+this.I+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
        ""+this.window.C:M2(this.I)) then
       Info("FAILED");
    endif

  in 3:
    Info("call :M3("+this.S+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
       ""+this.window.C:M3(this.S)) then
       Info("FAILED");
    endif

  in 4:
    Info("call :M4("+this.B+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
      ""+this.window.C:M4(this.B)) then
      Info("FAILED");
    endif

  in 5:
    Info("call :M5("+this.I+","+this.S+","+this.B+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
        ""+this.window.C:M5(this.I,this.S,this.B)) then
        Info("FAILED");
    endif
  in 6:
    Info("call :M6("+this.I+","+this.S+","+this.B+","+
     this.I+","+this.S+","+this.B+","+this.I+","+
      this.S+");");
       this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
      ""+this.window.C:M6(this.I,this.S,this.B,this.I,this.S,
      this.B,this.I,this.S)) then
      Info("FAILED");
    endif

  otherwise:
    Info("Error - unknown method");
      endcase
  }
}
```

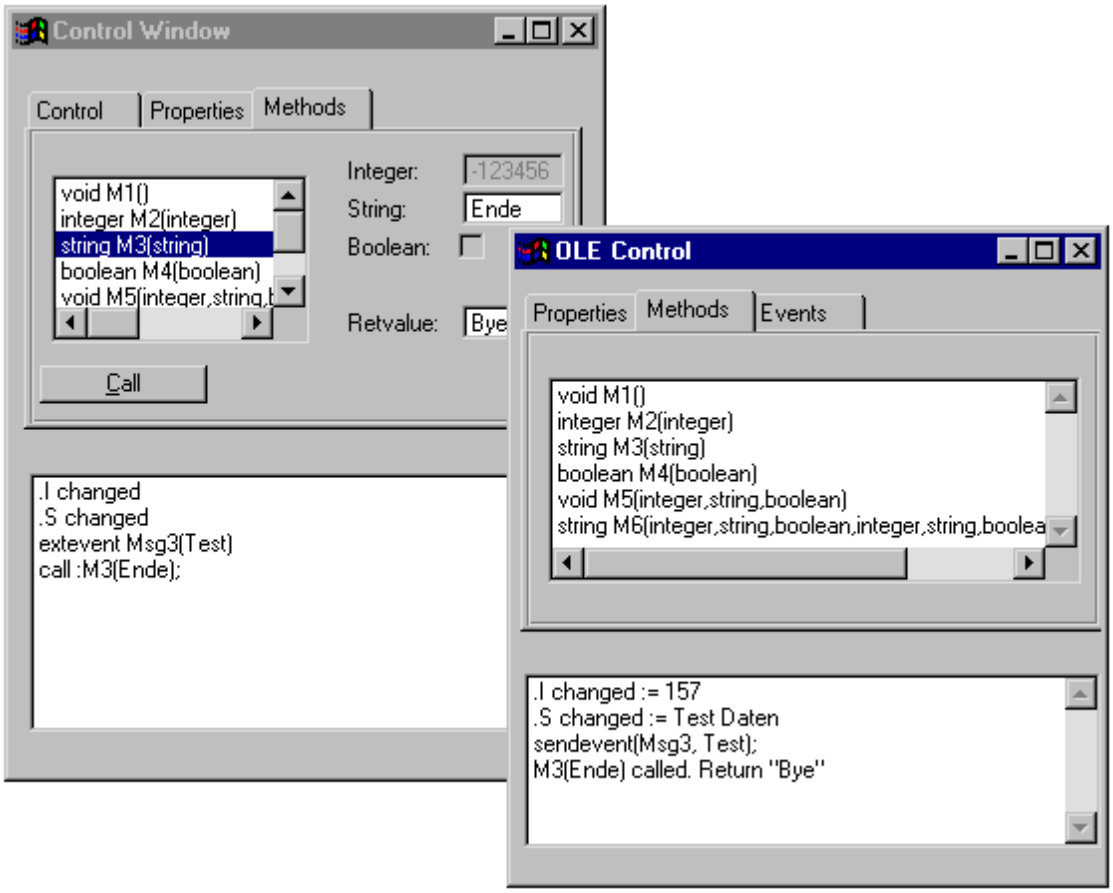

Abbildung 18: *Aufruf von Methoden im OLE-Server*

6.1.6 Der Client-Dialog

```
dialog Client
{
}
color ColWin
{
    0: rgb(47,175,207), grey(255), white;
    1: rgb(192,192,192), grey(255), white;
}
color ColInput
{
    0: rgb(111,159,175), grey(200), white;
    1: rgb(255,255,192), grey(200), white;
}
color ColBlack "BLACK", grey(0), white;
color ColWhite "WHITE", grey(0), white;
color ColRed "RED", grey(0), white;
color ColGreen "GREEN", grey(0), white;
```

```
color ColBlue "BLUE", grey(0), white;
color ColYellow "YELLOW", grey(0), white;
font FontNormal "8.Helv";
font FontBig "14.Helv";
font FontFixed "12.System VIO";

model groupbox MInteger
{
  .height 1;
  integer Value := -123456789;
  .xauto 0;
  on .Value changed
  {
    this.Et.content := itoa(this.Value);
  }
  child statictext
  {
    .text "Integer:";
  }
  child edittext Et
  {
    .xleft 8;
    .format "%-9d";
    .xauto 0;
    .content "-123456789";
    on charinput
    {
if fail(this.parent.Value := atoi(this.content)) then
    this.parent.Value := 0;
endif
    }
  }
}
model groupbox MString
{
  .xauto 0;
  .height 1;
  string Value shadows instance MString.Et.content;
  string Text shadows instance MString.St.text;
  child statictext St
  {
    .text "String:";
  }
  child edittext Et
  {
    .xleft 8;
```

```
.xauto 0;
    .content "Dialog Manager";
  }
}
model groupbox MBoolean
{
  .height 1;
  boolean Value shadows instance Cb.active;
  .width 80;
  .height 20;
  child statictext
  {
    .text "Boolean:";
  }
  child checkbox Cb
  {
    .xleft 8;
    .text "";
  }
}

window WiControl
{
  .title "OLE Control";
  .width 60;
  .height 20;
  .xleft 450;
  .visible := false;

  control C
  {
    .mode mode_client;
    .name "IdmTest.PropMethEvent";
     .width 300;
    .height 300;
    on .visible changed
    {
     CbVisible.Value := this.visible;
    }
    on .connect changed
    {
     CbConnect.Value := this.connect;
    }
    on .active changed
    {
     CbActive.Value := this.active;
```

```
}
on extevent "Msg1"
{
 Info("extevent Msg1()");
}
on extevent "Msg2" (integer I)
{
  Info("extevent Msg2("+I+")");
}
on extevent "Msg3" (string S)
{
  Info("extevent Msg3("+S+")");
}
on extevent "Msg4" (boolean B)
{
  Info("extevent Msg4("+B+")");
}
on extevent "Msg5" (integer I, string S, boolean B)
{
 Info("extevent Msg5("+I+","+S+","+B+")");
}
on extevent "Msg6" (integer P1, string P2, boolean P3,
   integer P4, string P5, boolean P6)
{
 Info("extevent Msg6 ("+P1+","+P2+","+P3+","+P4+"," +P5+","
    +P6+")");
}
on .I changed
{
  Info(".I changed");
  if fail(NpProperties.Integer.Value := this.I) then
    Info("getvalue of .I FAILED");
  endif
}
on .S changed
{
  Info(".S changed");
  if fail(NpProperties.String.Value := this.S) then
    Info("getvalue of .S FAILED");
  endif
}
on .B changed
{
  Info(".B changed");
  if fail(NpProperties.Boolean.Value := this.B) then
    Info("getvalue of .B FAILED");
```

```
      endif
    }
  }
}
rule void Info(string S)
{
  LbInfo.content[LbInfo.itemcount+1]  := S;
  LbInfo.topitem := LbInfo.itemcount;
}
model checkbox MCb
{
  boolean Value := false;

  on activate,deactivate
  {
    this.Value := this.active;
    if this.Value <> this.active then
      Info("changing "+this.text+" FAILED");
    endif
  }
}
window WiMain
{
  .width 60;
  .height 20;
  .title "Control Window";

  on close
  {
    exit();
  }
  object C shadows WiControl.C.self;

  child groupbox Gb
  {
    .xauto 0;
    .yauto 0;
    .borderwidth 0;
    child notebook Nb
    {
      .xauto 0;
      .yauto 1;
      .height 10;
    child notepage NpControl
    {
      .active true;
```

```
    .title "Control";
    MCb CbVisible
  {
    .text "Visible";
    .Value shadows WiControl.visible;
  }
    MCb CbConnect
  {
    .ytop 1;
    .text "Connect";
    .Value shadows WiControl.C.connect;
  }
    MCb CbActive
  {
    .ytop 2;
    .text "Active";
    .Value shadows WiControl.C.active;
  }
 }

   child notepage NpProperties
   {
     .title "Properties";
     child MInteger Integer
     {
     }
     child MString String
     {
       .ytop 1;
     }
     child MBoolean Boolean
     {
       .ytop 2;
     }
     child pushbutton PbGet
     {
       .yauto -1;
       .height 1;
       .text "&Get";
       .defbutton true;
       on select
       {
         Info("Try to get .I, .S and .B");
         if fail(this.parent.Integer.Value :=
            this.window.C.I) then
              Info("getvalue of .I FAILED");
```

```
        endif
        if fail(this.parent.String.Value :=
          this.window.C.S) then
            Info("getvalue of .S FAILED");
        endif
        if fail(this.parent.Boolean.Value :=
          this.window.C.B) then
            Info("getvalue of .B FAILED");
          endif
      }
    }
    child pushbutton PbSet
    {
      .yauto -1;
      .height 1;
      .text "&Set";
      .xleft 14;
      on select
        {
        Info("Try to set .I, .S and .B");
          if fail(this.window.C.I :=
             this.parent.Integer.Value) then
              Info("setvalue of .I FAILED");
          endif
          if fail(this.window.C.S :=
             this.parent.String.Value) then
             Info("setvalue of .S FAILED");
          endif
          if fail(this.window.C.B :=
             this.parent.Boolean.Value) then
            Info("setvalue of .B FAILED");
          endif
     }
   }
}
  child notepage NpMethods
  {
     .title "Methods";
      child listbox LbMethods
    {
      .xauto 1;
      .width 20;
      .yauto 0;
      .ybottom 1;
      .content[1] "void M1()";
      .content[2] "integer M2(integer)";
```

```
        .content[3] "string M3(string)";
        .content[4] "boolean M4(boolean)";
        .content[5] "void M5(integer,string,boolean)";
        .content[6] "string    M6(integer,string,boolean,
                   integer,string,boolean,integer,string)";
        .activeitem 1;
        .firstchar 1;
        on select
        {
       this.parent.Integer.sensitive :=
  (0 <> stringpos(this.content[this.activeitem], "integer"));
      this.parent.String.sensitive :=
  (0 <> stringpos(this.content[this.activeitem], "string"));
      this.parent.Boolean.sensitive :=
  (0 <> stringpos(this.content[this.activeitem], "boolean"));
        }
   }
child MInteger Integer
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .Et.active false;
}
child MString String
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .ytop 1;
}
child MBoolean Boolean
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .ytop 2;
}
child MString Retval
{
  .Text := "Retvalue:";
  .Value := "";
  .xleft 22;
  .yauto 1;
  .ytop 4;
}
```

```
child pushbutton PbCall
{
  .yauto -1;
  .height 1;
  .text "&Call";
  .defbutton true;
  integer I shadows instance NpMethods.Integer.Value;
  boolean B shadows instance NpMethods.Boolean.Value;
  string S shadows instance NpMethods.String.Value;
  on select
  {
    case this.parent.LbMethods.activeitem
  in 1:
    Info("call :M1();");
    this.parent.Retval.Value := "";
    if fail(this.window.C:M1()) then
    Info("FAILED");
    endif
  in 2:
    Info("call :M2("+this.I+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
        ""+this.window.C:M2(this.I)) then
       Info("FAILED");
    endif

  in 3:
    Info("call :M3("+this.S+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
       ""+this.window.C:M3(this.S)) then
       Info("FAILED");
    endif

  in 4:
    Info("call :M4("+this.B+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
      ""+this.window.C:M4(this.B)) then
      Info("FAILED");
    endif

  in 5:
    Info("call :M5("+this.I+","+this.S+","+this.B+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
```

```
          ""+this.window.C:M5(this.I,this.S,this.B)) then
          Info("FAILED");
      endif
    in 6:
      Info("call :M6("+this.I+","+this.S+","+this.B+","+
       this.I+","+this.S+","+this.B+","+this.I+","+
        this.S+");");
         this.parent.Retval.Value := "";
      if fail(this.parent.Retval.Value:=
        ""+this.window.C:M6(this.I,this.S,this.B,this.I,this.S,
        this.B,this.I,this.S)) then
        Info("FAILED");
      endif

    otherwise:
      Info("Error - unknown method");
        endcase
    }
  }
  }
  }
  }
  child listbox LbInfo
  {
    .xauto 0;
    .yauto 0;
    .ytop 10;
    .firstchar 1;
  }
}
```

6.2 Aufbau des Servers

Der Server besteht ebenfalls aus einem ***Notebook*** und einer ***Listbox***. Über das ***Notebook*** wird der Server gesteuert und in der ***Listbox*** werden die Ergebnisse und Anfragen des Clients protokolliert. Der Server wird „InPlace“ betrieben, das heißt, er nutzt die Fläche innerhalb eines Fensters des Clients für seine Darstellung.

Wenn eine Verbindung mit dem Server besteht, der Server aber noch nicht aktiviert worden ist, wird im Bereich des Clients ein Bild dargestellt. Nach der Aktivierung erscheinen dann die Objekte des Servers im Client.

Damit der Server ablauffähig ist, wurde dem ***Dialog*** und dem ***Control***-Objekt jeweils eine eindeutige UUID zugewiesen.

```
dialog IdmTest
{
  .uuid "499593d0-a159-11d1-a7e3-00a02444c34e";
}

tile TiPropMethEvent "IMD_IMAGES:isaicon.gif";

default control CONTROL
{
  integer Count := 0;

  on start
  {
    CONTROL.Count := CONTROL.Count + 1;
  }

  on finish
  {
    CONTROL.Count := CONTROL.Count - 1;
    if CONTROL.Count=0 then
      exit();
    endif
  }
}

control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
}
```

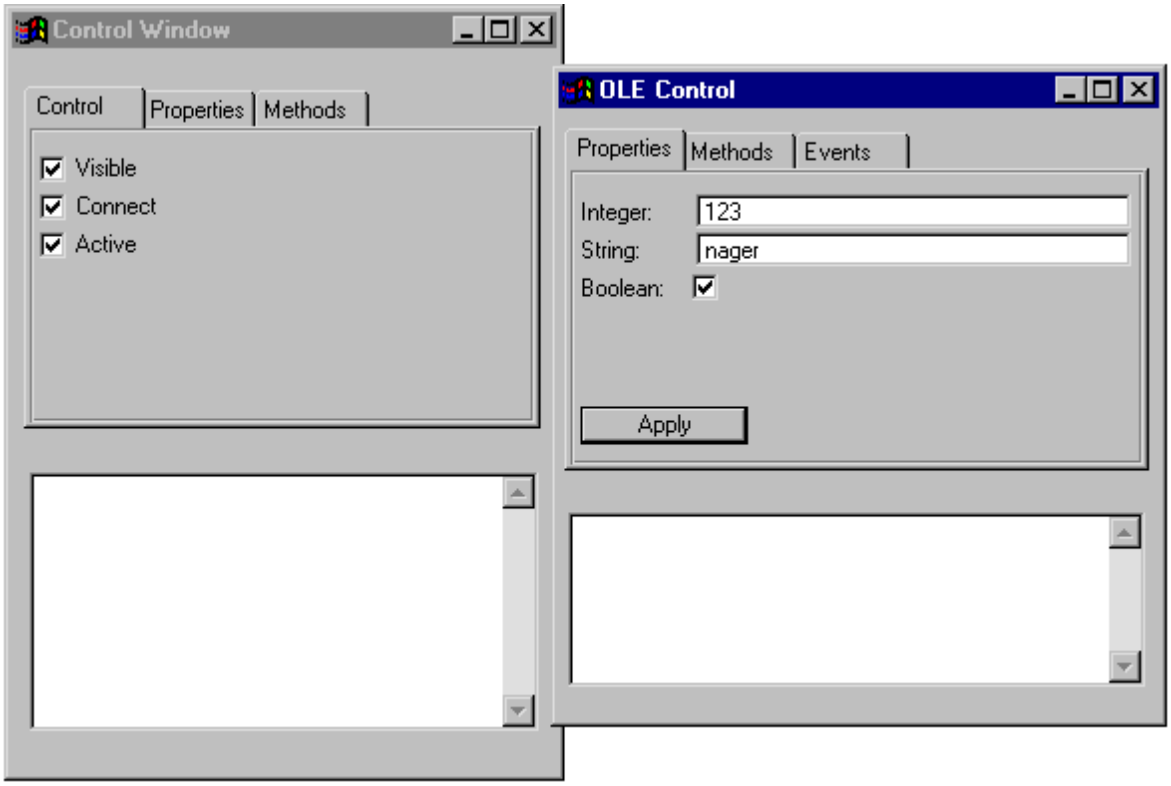

Abbildung 19: *Aktiver OLE-Server*

6.2.1 Bereitstellung des Servers

Anders als beim Client müssen beim Server einige Schritte durchgeführt werden, damit der Server als OLE-Server betrieben werden kann.

Zunächst wird mit Hilfe der Option **-writeole** des IDM-Simulationsprogramms die zur Registrierung des OLE-Servers notwendige **reg**-Datei sowie die zum Bereitstellen der Ablaufkomponenten notwendige **idl**-Datei erzeugt. Die Kommandozeile sieht dabei wie folgt aus:

```
$(THIS).reg $(THIS).idl: $(THIS).dlg
  $(IDM) $(THIS).dlg -localserver "$(IDM) $(THIS).dlg \
    -IDMenv MODLIB=$(THISDIR) -IDMerrfile $(THIS).log" \
    -writeole $(THIS)
  regedit $(THIS).reg
```

Mit dem Befehl `regedit` wird der Server gleich im System registriert.

Anschließend wird die erzeugte **idl**-Datei mit Hilfe des MIDL-Compilers übersetzt.

```
$(THIS).tlb $(THIS)_p.c $(THIS).h dlldata.c: $(THIS).idl
  midl /ms_ext /app_config /c_ext /tlb $(THIS).tlb /Zp1 \
    /env win32 /Os $(THIS).idl
```

Zum Schluss wird aus den Objektdateien eine DLL generiert.

```
$(OUTFILE) : $(OBJS) $(TARGET).res $(DEFFILE)
  echo ++++++++++
  echo Linking $@
  echo $(LINK)                  > $(TARGET).lrf
  echo $(ENTRY)                >> $(TARGET).lrf
  echo -def:$(THIS).def        >> $(TARGET).lrf
  echo -out:$(OUTFILE)         >> $(TARGET).lrf
  echo -machine:IX86           >> $(TARGET).lrf
  echo -subsystem:windows5.01  >> $(TARGET).lrf
  echo -align:0x1000           >> $(TARGET).lrf
  echo $(OBJS1)                >> $(TARGET).lrf
  echo $(OBJS2)                >> $(TARGET).lrf
  echo $(OBJS3)                >> $(TARGET).lrf
  echo $(OBJS4)                >> $(TARGET).lrf
  echo $(OBJS5)                >> $(TARGET).lrf
  echo $(OBJS6)                >> $(TARGET).lrf
  echo $(TARGET).res           >> $(TARGET).lrf
  echo $(LIBS)                 >> $(TARGET).lrf
  echo $(LIBS32)               >> $(TARGET).lrf
  link @$(TARGET).lrf
  del $(TARGET).lrf
```

Danach kann der Server von einem Client benutzt werden.

Das zugehörige Makefile sieht wie folgt aus:

```
THISDIR=h:\ole\clntserv
THIS=$(THISDIR)\server
IDM=idmole.exe

CL32    = -G3s
WX      =
LINKDLL = /DLL
DEFDLL  = -D_DLL
ENTRY   = -entry:LibMain32

DEFUNICODE = -DWIN32ANSI
LINKD32    = -debug:full $(LINKDLL) -debugtype:cv
LINKN32    = -debug:none $(LINKDLL)
DEFS32     = -DWIN32 $(DEFDLL) -D_X86_=1 $(DEFUNICODE)
TLBDEFS    = -DWIN32
#BOOKLIB   = inole.lib
LIBS32A    = msvcrt.lib kernel32.lib user32.lib gdi32.lib comdlg32.lib
advapi32.lib
LIBS32B    = ole32.lib oleaut32.lib uuid.lib
LIBS32     = $(LIBS32A) $(LIBS32B) $(BOOKLIB)

LIBS   = rpcrt4.lib
```

```
CONTIN =

CFLAGS = -c -Od -Z7 -Ze -W3 -nologo $(CL32) \
         -D_WIN32_WINNT=0x400
LINK   = $(LINKD32) /NOD
DEFS   = $(DEFS32) -DSTRICT -DDEBUG

.SUFFIXES: .h .obj .exe .dll .cpp .res .rc .tlb .odl

OUTFILE = $(THIS).dll
TARGET  = $(THIS)
DEFFILE = $(THIS).def

OBJS1 = $(THIS)_i.obj $(THIS)_p.obj
OBJS2 = dlldata.obj libmain.obj
OBJS3 = ""
OBJS4 = ""
OBJS5 = ""
OBJS6 = ""
OBJS  = $(OBJS1) $(OBJS2)

#####################################################################

.c.obj:
  echo ++++++++++
  echo Compiling $*.c
  cl $(CFLAGS) $(DEFS)  $*.c

.cpp.obj:
  echo ++++++++++
  echo Compiling $*.c
  cl $(CFLAGS) $(DEFS)  $*.cpp

.rc.res:
  echo ++++++++++
  echo Compiling Resources
  rc -r $(DEFS) $(DOC) -fo$@ $*.rc

#####################################################################

all: $(THIS).reg $(THIS).idl $(THIS).tlb $(THIS).dll

clean:
  del $(THIS).tlb
  del $(THIS).reg
  del $(THIS).dll
  del $(THIS).odl
```

```
  del $(THIS)_i.c
  del $(THIS)_p.c
  del $(THIS).h
  del dlldata.c
  del *.obj
  del *.exe
  del *.lrf

$(THIS).reg $(THIS).idl: $(THIS).dlg
  $(IDM) $(THIS).dlg -localserver "$(IDM) $(THIS).dlg \
    -IDMenv MODLIB=$(THISDIR) -IDMerrfile $(THIS).log" \
    -writeole $(THIS)
  regedit $(THIS).reg

$(THIS).tlb $(THIS)_p.c $(THIS).h dlldata.c: $(THIS).idl
  midl /ms_ext /app_config /c_ext /tlb $(THIS).tlb /Zp1 \
    /env win32 /Os $(THIS).idl

$(THIS)_p.obj: $(THIS)_p.c
$(THIS)_i.obj: $(THIS)_i.c
dlldata.obj: dlldata.c
libmain.obj: libmain.cpp

###########################################################################

$(OUTFILE) : $(OBJS) $(TARGET).res $(DEFFILE)
  echo ++++++++++
  echo Linking $@
  echo $(LINK)                  > $(TARGET).lrf
  echo $(ENTRY)                >> $(TARGET).lrf
  echo -def:$(THIS).def        >> $(TARGET).lrf
  echo -out:$(OUTFILE)         >> $(TARGET).lrf
  echo -machine:IX86           >> $(TARGET).lrf
  echo -subsystem:windows5.01  >> $(TARGET).lrf
  echo -align:0x1000           >> $(TARGET).lrf
  echo $(OBJS1)                >> $(TARGET).lrf
  echo $(OBJS2)                >> $(TARGET).lrf
  echo $(OBJS3)                >> $(TARGET).lrf
  echo $(OBJS4)                >> $(TARGET).lrf
  echo $(OBJS5)                >> $(TARGET).lrf
  echo $(OBJS6)                >> $(TARGET).lrf
  echo $(TARGET).res           >> $(TARGET).lrf
  echo $(LIBS)                 >> $(TARGET).lrf
  echo $(LIBS32)               >> $(TARGET).lrf
  link @$(TARGET).lrf
  del $(TARGET).lrf
```

6.2.2 Abfragen und Setzen von Attributen

Damit der Client Attribute am Server abfragen kann, muss im Server nichts programmiert werden. Es können automatisch alle am ***Control***-Objekt definierten Attribute vom Client erfragt und gesetzt werden.

Im Beispiel sind die Attribute „I“, „S“ und „B“ vom Client abfragbar.

```
control PropMethEvent
{
  integer I := 123;
  string  S := "Dialog Manager";
  boolean B := true;
}
```

6.2.3 Aufruf von Methoden

Damit der Client Methoden im Server aufrufen kann, müssen diese als ganz normale Methoden des ***Control***-Objektes definiert werden. Diese Methoden können dann auf alle im Dialog definierten Objekte und Attribute zugreifen.

In diesem Beispiel sehen die Methoden wie folgt aus:

```
control PropMethEvent
{
  .message[1] Msg1;
  .message[2] Msg2;
  .message[3] Msg3;
  .message[4] Msg4;
  .message[5] Msg5;
  .message[6] Msg6;

  rule void M1
  {
    Info("M1() called.");
    sendevent(this, Msg1);
  }

  rule integer M2 (integer I input)
  {
    Info("M2(" + I + ") called. Return -123456789");
    return -123456789;
  }

  rule string M3 (string S input)
  {
    Info("M3(" + S + ") called. Return \"Bye\"");
    return "Bye";
```

```
  }

  rule boolean M4 (boolean B input)
  {
    Info("M4(" + B + ") called. Return " + ( not B ));
    return ( not B );
  }

  rule void M5 (integer I input, string S input, boolean B input)
  {
    Info("M5(" + I + ", " + S + ", " + B + ") called.");
  }

  rule string M6 (integer P1 input, string P2 input,
    boolean P3 input, integer P4 input, string P5 input,
    boolean P6 input, integer P7 input, string P8 input)
  {
    Info("M6(" + P1 + ", " + P2 + ", " + P3 + ", " + P4 +
         ", " + P5 + ", " + P6 + ", " + P7 + ", " + P8 +
         ") called. Return \"Abracadabra!\"");
    return "Abracadabra!";
  }
}
```

6.2.4 Versenden von Ereignissen

Wenn der Server dem Client Ereignisse schicken soll, müssen diese im Server definiert und entsprechend programmiert werden. Die Definition solcher Ereignisse erfolgt mit Hilfe der Ressource ***Message***. Die Ereignisse werden dann mit der eingebauten Funktion **sendevent** an den Client geschickt.

Diese Ereignisse sind im Beispiel wie folgt definiert:

```
message Msg1;
message Msg2 (integer I);
message Msg3 (string S);
message Msg4 (boolean B);
message Msg5 (integer I, string S, boolean B);
message Msg6 (integer P1, string P2, boolean P3,
              integer P4, string P5, boolean P6);
```

Zusätzlich müssen die von einem Control verschickbaren Ereignisse am ***Control***-Objekt im Attribut *.message[integer]* definiert werden. Die in diesem Attribut definierten ***Messages*** werden dann an den Client weitergeleitet.

```
control PropMethEvent
{
```

```
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
  .message[1] Msg1;
  .message[2] Msg2;
  .message[3] Msg3;
  .message[4] Msg4;
  .message[5] Msg5;
  .message[6] Msg6;
}
```

Das Versenden der Ereignisse erfolgt nach Selektion des Pushbuttons „Send“ in folgender Regel:

```
child pushbutton PbSend
{
  .yauto     -1;
  .height    1;
  .text      "Send";
  .defbutton true;
  integer I shadows instance NpEvents.Integer.Value;
  boolean B shadows instance NpEvents.Boolean.Value;
  string  S shadows instance NpEvents.String.Value;

  on select
  {
    case this.parent.LbEvents.activeitem
    in 1:
      sendevent(this.control, Msg1);
      Info("sendevent(Msg1);");
    in 2:
      sendevent(this.control, Msg2, this.I);
      Info("sendevent(Msg2, " + this.I + ");");
    in 3:
      sendevent(this.control, Msg3, this.S);
      Info("sendevent(Msg3, " + this.S + ");");
    in 4:
      sendevent(this.control, Msg4, this.B);
      Info("sendevent(Msg4, " + this.B + ");");
    in 5:
      sendevent(this.control, Msg5, this.I, this.S, this.B);
      Info("sendevent(Msg5, " + this.I + ", " + this.S + ", " +
           this.B + ");");
    in 6:
      sendevent(this.control, Msg6, this.I, this.S, this.B,
                this.I, this.S, this.B);
      Info("sendevent(Msg6, " + this.I + ", " + this.S + ", " +
           this.B + ", " + this.I + ", " + this.S + ", " +
           this.B + ");");
```

```
    otherwise:
      Info("Error - unknown event");
    endcase
  }
}
```

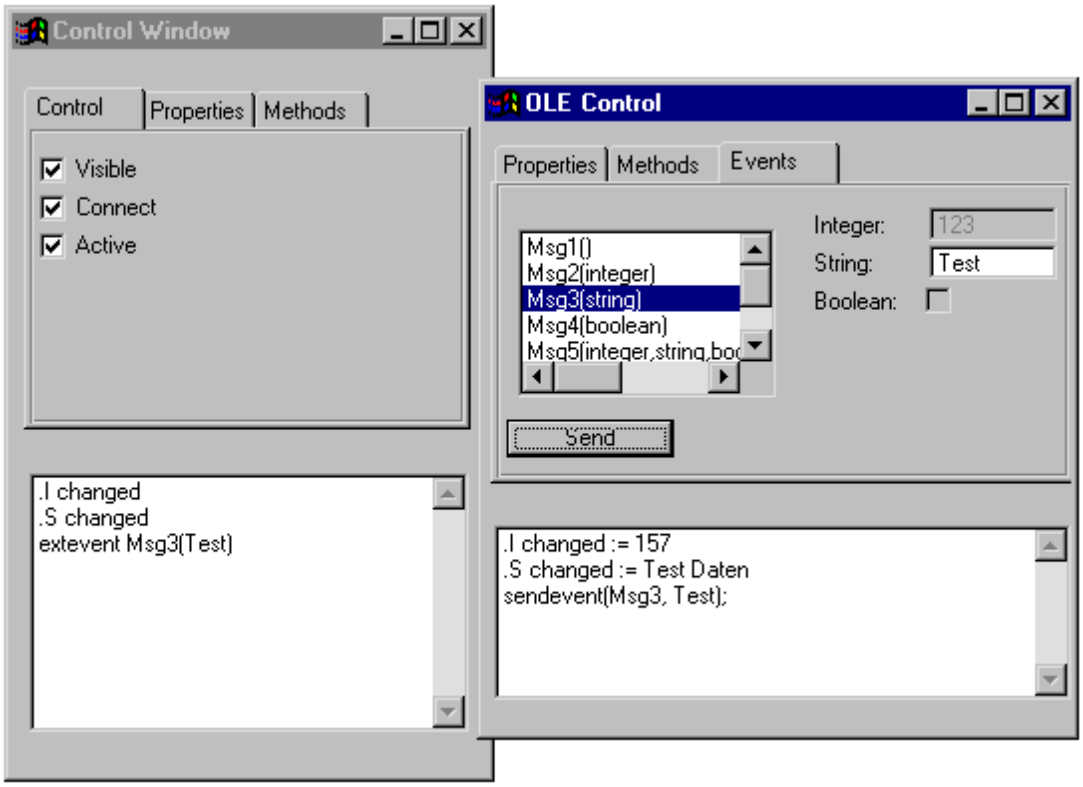

Abbildung 20: *Verschicken von Ereignissen*

6.2.5 Versenden von Benachrichtigungen (Notifications)

Um Benachrichtigungen über die Änderungen von Attributen an den Client zu verschicken, muss im Server nichts programmiert werden. Sobald sich ein Attribut an dem Control ändert, wird automatisch eine Benachrichtigung an den Client verschickt.

Der Regelcode für die Zuweisung innerhalb des Servers sieht wie folgt aus:

```
child pushbutton PbApply
{
  .yauto     -1;
  .height    1;
  .text      "Apply";
  .defbutton true;

  on select
  {
```

```
    this.control.I := this.parent.Integer.Value;
    this.control.S := this.parent.String.Value;
    this.control.B := this.parent.Boolean.Value;
  }
}
```

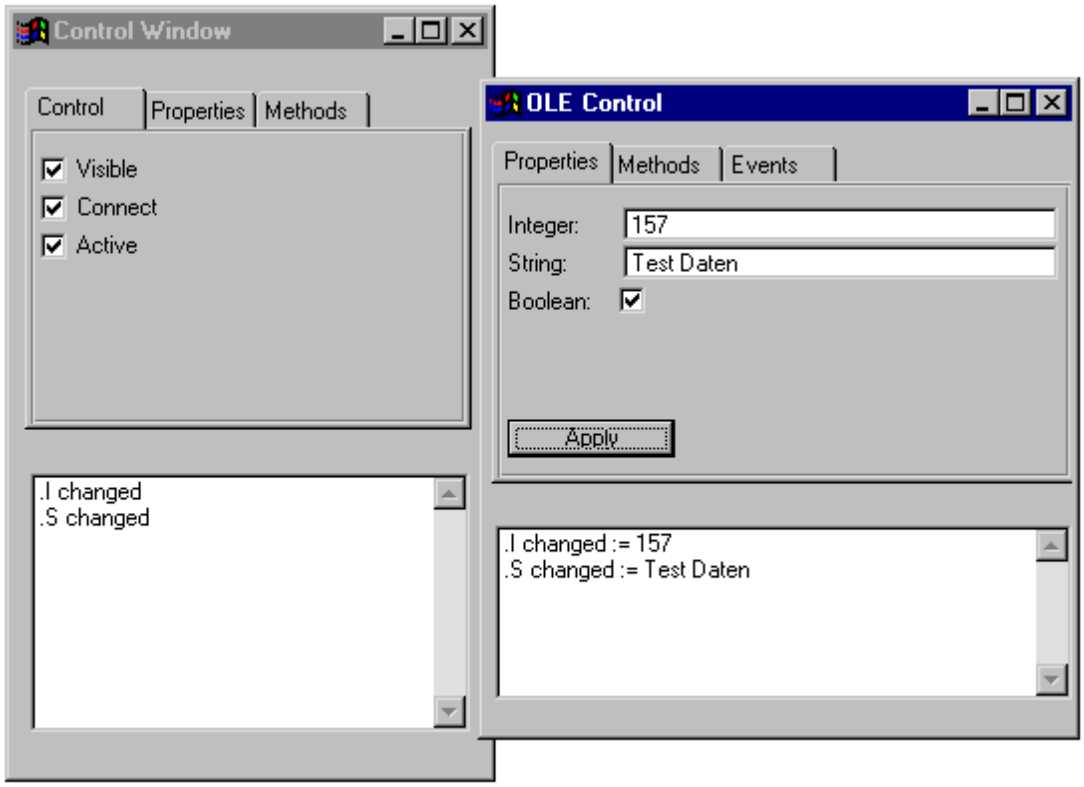

Abbildung 21: *Austausch von Benachrichtigungen nach Selektion des Apply-Pushbuttons*

6.2.6 Der Server Dialog

```
dialog IdmTest
{
  .uuid "499593d0-a159-11d1-a7e3-00a02444c34e";
}

tile TiPropMethEvent "IMD_IMAGES:isaicon.gif";

message Msg1;
message Msg2(integer I);
message Msg3(string S);
message Msg4(boolean B);
message Msg5(integer I, string S, boolean B);
message Msg6(integer P1, string P2, boolean P3,
             integer P4, string P5, boolean P6);
```

```
model groupbox MInteger
{
  .height 1;
  .xauto 0;
  integer Value := 123;

  child statictext { .text "Integer:"; }

  child edittext Et
  {
    .xleft 8;
    .xauto 0;
    .format "%-9d";
    .content "123";

    on charinput
    {
      if fail(this.parent.Value := atoi(this.content)) then
        this.parent.Value := 0;
      endif
    }
  }
  on .Value changed
  {
    this.Et.content := itoa(this.Value);
  }
}

model groupbox MString
{
  .height 1;
  .xauto 0;
  string Value shadows instance MString.Et.content;

  child statictext { .text "String:"; }

  child edittext Et
  {
    .xleft 8;
    .xauto 0;
    .content "Dialog Manager";
  }
}

model groupbox MBoolean
{
```

```
  .height 1;
  boolean Value shadows instance Cb.active;

  child statictext { .text "Boolean:"; }

  child checkbox Cb
  {
    .xleft 8;
    .text "";
  }
}

default control CONTROL
{
  integer Count := 0;

  on start
  {
    CONTROL.Count := CONTROL.Count + 1;
  }

  on finish
  {
    CONTROL.Count := CONTROL.Count - 1;
    if CONTROL.Count=0 then
      exit();
    endif
  }
}

model control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
  .message[1] Msg1;
  .message[2] Msg2;
  .message[3] Msg3;
  .message[4] Msg4;
  .message[5] Msg5;
  .message[6] Msg6;

  integer I := 123;
  string S  := "Dialog Manager";
  boolean B := true;

  rule void M1
```

```
{
  Info("M1() called.");
  sendevent(this, Msg1);
}

rule integer M2 (integer I input)
{
  Info("M2(" + I + ") called. Return -123456789");
  return -123456789;
}

rule string M3 (string S input)
{
  Info("M3(" + S + ") called. Return \"Bye\"");
  return "Bye";
}

rule boolean M4 (boolean B input)
{
  Info("M4(" + B + ") called. Return " + ( not B ));
  return ( not B );
}

rule void M5 (integer I input, string S input, boolean B input)
{
  Info("M5(" + I + ", " + S + ", " + B + ") called.");
}

rule string M6 (integer P1 input, string P2 input,
  boolean P3 input, integer P4 input, string P5 input,
  boolean P6 input, integer P7 input, string P8 input)
{
  Info("M6(" + P1 + ", " + P2 + ", " + P3 + ", " + P4 +
       ", " + P5 + ", " + P6 + ", " + P7 + ", " + P8 +
       ") called. Return \"Abracadabra!\"");
  return "Abracadabra!";
}

on .I changed
{
  Info(".I changed := " + this.I);
  this.Gb.Nb.NpProperties.Integer.Value := this.I;
}

on .S changed
{
  Info((".S changed := " + this.S));
```

```
  this.Gb.Nb.NpProperties.String.Value := this.S;
}

on .B changed
{
  Info((".B changed := " + this.B));
  this.Gb.Nb.NpProperties.Boolean.Value := this.B;
}

child groupbox Gb
{
  .xauto 0;
  .yauto 0;
  .borderwidth 0;

  child notebook Nb
  {
    .xauto 0;
    .yauto 1;
    .height 10;

    child notepage NpProperties
    {
      .active true;
      .title "Properties";

      child MInteger Integer { }

      child MString String { .ytop 1; }

      child MBoolean Boolean
      {
        .ytop 2;
        .Value := true;
      }

      child pushbutton PbApply
      {
        .yauto -1;
        .height 1;
        .text "Apply";
        .defbutton true;

        on select
        {
          this.control.I := this.parent.Integer.Value;
          this.control.S := this.parent.String.Value;
```

```
      this.control.B := this.parent.Boolean.Value;
    }
  }
}

child notepage NpMethods
{
  .active false;
  .title "Methods";

  child listbox LbMethods
  {
    .xauto 0;
    .yauto 0;
    .content[1] "void M1()";
    .content[2] "integer M2(integer)";
    .content[3] "string M3(string)";
    .content[4] "boolean M4(boolean)";
    .content[5] "void M5(integer, string, boolean)";
    .content[6] "string M6(integer, string, boolean,   ...\
                 integer, string, boolean, integer, string)";
    .firstchar 1;
  }
}

child notepage NpEvents
{
  .title "Events";

  child listbox LbEvents
  {
    .xauto 1;
    .width 20;
    .yauto 0;
    .ybottom 1;
    .content[1] "Msg1()";
    .content[2] "Msg2(integer)";
    .content[3] "Msg3(string)";
    .content[4] "Msg4(boolean)";
    .content[5] "Msg5(integer, string, boolean)";
    .content[6] "Msg6(integer, string, boolean,        ...\
                      integer, string, boolean)";
    .activeitem 1;
    .firstchar 1;

    on select
    {
```

```
    this.parent.Integer.sensitive :=
      (0 <> stringpos(this.content[this.activeitem],
                      "integer"));
    this.parent.String.sensitive :=
      (0 <> stringpos(this.content[this.activeitem],
                      "string"));
    this.parent.Boolean.sensitive :=
      (0 <> stringpos(this.content[this.activeitem],
                      "boolean"));
  }
}

child MInteger Integer
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .Et.active false;
}

child MString String
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .ytop 1;
}

child MBoolean Boolean
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .ytop 2;
}

child pushbutton PbSend
{
  .yauto     -1;
  .height    1;
  .text      "Send";
  .defbutton true;
  integer I shadows instance NpEvents.Integer.Value;
  boolean B shadows instance NpEvents.Boolean.Value;
  string  S shadows instance NpEvents.String.Value;

  on select
```

```
        {
          case this.parent.LbEvents.activeitem
          in 1:
            sendevent(this.control, Msg1);
            Info("sendevent(Msg1);");
          in 2:
            sendevent(this.control, Msg2, this.I);
            Info("sendevent(Msg2, " + this.I + ");");
          in 3:
            sendevent(this.control, Msg3, this.S);
            Info("sendevent(Msg3, " + this.S + ");");
          in 4:
            sendevent(this.control, Msg4, this.B);
            Info("sendevent(Msg4, " + this.B + ");");
          in 5:
            sendevent(this.control, Msg5, this.I, this.S,
                      this.B);
            Info("sendevent(Msg5, " + this.I + ", " + this.S +
                           ", " + this.B + ");");
          in 6:
            sendevent(this.control, Msg6, this.I, this.S,
                      this.B, this.I, this.S, this.B);
            Info("sendevent(Msg6, " + this.I + ", " + this.S +
                           ", " + this.B + ", " + this.I +
                           ", " + this.S + ", " + this.B +
                           ");");
          otherwise:
            Info("Error - unknown event");
          endcase
        }
      }
    }
  }

  child listbox LbInfo
  {
    .xauto 0;
    .yauto 0;
    .ytop 10;
    .firstchar 1;
  }
}

on extevent 1
{
  sendevent(this,Msg3,"Bye event");
  this:sendevent(Msg4, true);
```

```
  }
}

rule void Info (string S input)
{
  LbInfo.content[(LbInfo.itemcount + 1)] := S;
  LbInfo.topitem := LbInfo.itemcount;
}

on IdmTest extevent 1(object C)
{
  Info("on IdmTest extevent 1");
  sendevent(C, Msg2, 909);
}
```

7 Die Dialog Manager-Umgebung

Als Simulationsprogramm für OLE-Anwendungen kann das Programm **idmole** benutzt werden. Es hat die selben Optionen wie das Simulationsprogramm mit dem Unterschied, dass zusätzlich die in diesem Handbuch beschriebenen OLE-spezifischen Optionen verstanden werden.

Wenn eine Anwendung mit OLE-Fähigkeiten gebaut werden soll, müssen zusätzlich zu den üblichen Bibliotheken noch die **dmole.lib** und die **dmoleini.obj** dazu gelinkt werden.

Weitere Änderungen sind nicht notwendig.

7.1 Hinweise zur Nutzung des Microsoft Testcontainers

Wenn ein OLE-Control mit *.mode = mode_server* in den Microsoft „Testcontainer für ActiveX Controls“ eingefügt wird, dann stürzt der Testcontainer mit einer Zugriffsverletzung ab. Dies ist ein Problem des Testcontainers, welcher allerdings auch im Quellcode verfügbar ist.

Um diesen Quellcode zu erhalten, muss im Hilfeindex des Microsoft Visual Studios *„TSTCON“* eingeben und dann der Eintrag „TSTCON sample [MFC]“ (oder ähnlich) aktiviert werden. Jetzt kann der Quellcode installiert werden.

Innerhalb des Quellcodes ist im Modul **ExtCtl.cpp** in der Methode **CExtendedControl::QueryInterface** (ca. Zeile 180) folgender auskommentierter Code enthalten:

```
//    if (m_pControl == NULL)
//        hResult = E_UNEXPECTED;
//    else
```

Wenn hier nun die Kommentarzeichen „//“ entfernt und danach der Testcontainer neu compiliert werden, tritt das Problem nicht mehr auf.

Anmerkung

Der Testcontainer des Microsoft Visual Studios 2005 hat dieses Problem nicht mehr.

Index