kleiner kommunikation
max@kleiner.com

1 Einsatz von Interfaces

Zu was dienen eigentlich Interfaces in ObjectPascal? Dieses Leistungsmerkmal haben die Borländer zur Unterstützung von COM erschaffen (seit Delphi 4), es wurde aber unabhängig von COM implementiert, d.h. die Mechanismen funktionieren sogar in Kylix und lassen sich unter Windows losgelöst von COM gewinnbringend einsetzen. Zeit also einen Blick hinter die Kulissen der Zwischengesichter zu werfen, sozusagen Face to Face.

1.1 Designfragen

Das Object Pascal-Schlüsselwort `interface` erlaubt das Erstellen und Verwenden von Schnittstellen in Anwendungen. Definieren wir mal:

Ein Interface ist eine Liste von Zeigern auf Methoden.

Schnittstellen erweitern das Modell der Einfachvererbung der VCL, indem einzelne Klassen mehr als eine Schnittstelle implementieren können. Gleichzeitig können mehrere Klassen mit unterschiedlichen Basisklassen gemeinsam auf ein Interface zugreifen. Wozu das gut sein soll ? Schnittstellen sind hilfreich, wenn beispielsweise Stream-Operationen, Transferoperationen oder Sicherheitsfunktionen für viele verschiedene Objekte erforderlich sind, die aber aus Typisierungsverträglichkeit nicht in den jeweiligen Basisklassen sind. Außerdem bilden Schnittstellen eine der Grundlagen der verteilten Objektmodelle COM und CORBA und sind auch bekannte Konstrukte anderer Sprachen, da ja ein registriertes Interface schon als Type-Library gilt.

Beim Design umfangreicher Komponenten oder z.B. eines Open Tool API, gelangt man schon mal an die Grenzen einer linearen Objekthierarchie und lernt den Einsatz verschiedener Basisklassen schätzen. So lassen sich universelle Methoden mehrfach verwenden, ohne immer von der gleichen Hierarchie abhängig zu sein oder ständig in die Bibliothek eingreifen zu müssen.

Ich hab mal das Borland Beispiel in ein Klassendiagramm umgesetzt (Abb. 1), um auf einen Blick die Stärken von Interfaces darzustellen. Es sind zwei Basisklassen vorhanden, die `TSquare` und `TCircle` unterstützen. Nun erscheint es vernünftig, dass `TCircle` zwar `paint` implementiert aber nicht `Rotate`, da es bei einem Kreis wahrlich nichts zu drehen gibt. Die beiden Methoden sind so universell, das man sie aus Designgründen, wie der besseren Granularität und Wiederverwendung, nicht einfach in eine Basisklasse wie `TPolygonObject` stecken will. Gut, solche Konstrukte fallen nicht einfach vom Himmel, sie entstehen langsam nach diversen Reviews oder Refactorings im Schweisse des Zwischengesichtes.

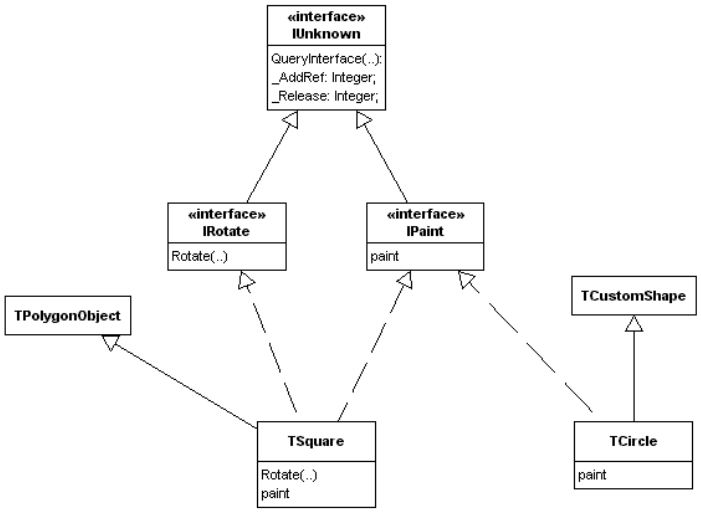

Abb.1: Das Borland Beispiel als Klassendiagramm

Wir sehen aber gleich praktisch, dass Interfaces auch ohne kompliziertes Design oder dem Umsetzen von standardisierten Patterns [MK00] ein paar tolle Merkmale aufweisen, die sich schon im Einzelfall lohnen können:

- N-Objekte sind zuweisungskompatibel, wenn sie dasselbe Interface unterstützen
- Automatische Speicherfreigabe durch Referenzzähler
- Bequemes Typecasting mit eingebauten Funktionen
- Interface Name wird durch eine GUID weltweit eindeutig

Eine Schnittstelle ähnelt einer Klasse, die nur abstrakte Methoden und eine genaue Definition ihrer Funktionalität enthält, jedoch besitzt sie selbst kein Grundverhalten. Solange die Schnittstelle von `IUnknown` erbt und nicht von `IDispatch` sind wir noch unabhängig von der COM-Jacke (Ich habe nichts von einer Zwangsjacke gesagt ;)).

```
unit income1;
IIncomeInt = interface (IUnknown)
    ['{DBB42A04-E60F-41EC-870A-314D68B6913C}'] //GUID
    function GetIncome(const aNetto: Currency): Currency; stdcall;
    function GetRate: Real;
    function queryDLLInterface(var queryList: TStringList): TStringList;
            stdcall;
    procedure SetRate(const aPercent, aYear: integer); stdcall;
    property Rate: Real read GetRate;
  end;
```

Eine mit einem Schnittstellentyp deklarierte Variable wie im folgenden `incomeIntRef` kann Instanzen jeder Klasse referenzieren, die von der Schnittstelle implementiert wird. Das Objekt muss einfach die Methoden der Schnittstelle unterstützen. Das typische an einer Schnittstellenvariablen in der Deklaration ist eben die Referenz auf die Schnittstelle und nicht auf die Klasse direkt zu setzen:

```
private
  incomeIntRef: IIncomeInt;
begin
   //in einer DLL+
```

```
incomeIntRef:= createIncome;
 //in einer DCU
incomeIntRef:= TIncomeRealIntf.create;
```

Das Schema sieht folgendermassen aus:

1. Deklaration als Member in einem Client: Variable von Schnittstelle
2. Definition im Konstruktoraufruf: Objekt an Variable zuweisen
3. Implementieren der Schnittstellenmethoden in der Klasse

Mit Hilfe solcher Variablen lassen sich Schnittstellenmethoden auch aufrufen, wenn zur Compilierzeit noch nicht bekannt ist, wo die Schnittstelle implementiert wird, jedoch Namen und Signaturen sind schon festgelegt. In der Teamarbeit ein unschätzbarer Vorteil. In unserem Beispiel bauen wir auf der Technik von DLL+ auf [DE01], d.h. die Implementation befindet sich in einer DLL, welche die Referenz des Objektes `TIncomeRealIntf` exportiert. Sonst bleibt sich vom Prinzip her alles gleich, d. h. Sie entscheiden ob Implementierung und Interface in der selben Datei sind, einzeln in einer separaten DCU oder eben ausgelagert in einer DLL. Ich empfehle aber, die Schnittstellen immer in eine eigene Datei zu legen, wie `unit income1` in Abb.2 es demonstriert.

Schnittstellen sind wie Klassen nur im äussersten Gültigkeitsbereich eines Programms oder einer Unit, nicht aber in einer Prozedur oder Funktion, zu deklarieren. Eine Schnittstellendeklaration ähnelt in weiten Teilen einer Klassendeklaration. Es gelten jedoch folgende Einschränkungen:

- Die Elementliste darf nur Methoden und Eigenschaften enthalten. Felder sind in Schnittstellen nicht erlaubt. Da für Schnittstellen keine Felder verfügbar sind, sind properties und die entsprechenden Methoden (get, set) einzusetzen
- Alle Elemente einer Schnittstelle sind als public deklariert. Sichtbarkeitsattribute und Speicherattribute sind nicht erlaubt. Es lässt sich aber ein Array-Property mit der Direktive default als Standardeigenschaft deklarieren.
- Schnittstellen haben keine Konstruktoren oder Destruktoren, lassen sich auch nicht instanzieren, ausgenommen durch die Klassen selbst, welche die Methoden implementieren. Bindings wie virtual, dynamic, abstract oder override sind nicht erlaubt. Da Schnittstellen keine eigenen Methoden implementieren, haben diese Bezeichnungen keine Bedeutung.

1.2 Baustelle

Genug von Theorie und Design, ich komme von der Schnittstelle zur Baustelle, d.h zur Implementierung. Unser Beispiel einer Zinseszinsberechnung in Abb. 2 besteht aus mindestens drei Units:

Die Library *income.dpr* beinhaltet als DLL die eigentlichen konkreten Methoden und die zu exportierende Referenz als Funktion.

Die Unit *income1.pas*, beinhaltet die Schnittstelle `IIncomeInt`, weitere Interfaces lassen sich hinzufügen.

Und eine Unit, die den Client als `TfrmIncome` mit der Referenz darstellt.

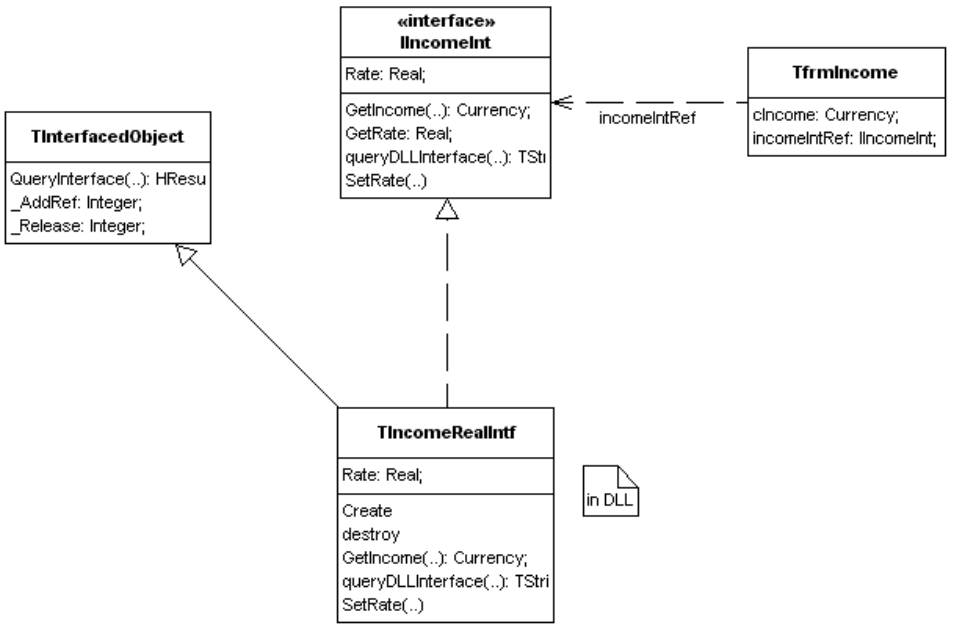

Abb. 2: Das Klassendiagramm mit dem Interface

Die Library *income.dpr*, die ja von der Schnittstelle, d.h. von der Klasse `IIncomeInt`, erbt, implementiert die Methoden nach Wunsch und Anforderung in einer DLL. Wie gesagt, die Implementation lässt sich auch in eine DCU oder ein Package packen. Die Diskussion DLL versus Packages sollte man ein andermal führen ;). Spezifisch für die Klasse ist nun, dass von der Schnittstelle einerseits und von einer Basisklasse andrerseits geerbt wird. Auf diese Basisklasse gehe ich nun näher ein.
Über allem steht in Delphi die Deklaration von `TInterfacedObject` aus der Unit System, da der Vorfahr ja `TObject` sein muss und zugleich IUnknown jeweils implementiert. `TInterfacedObject` implementiert also die Schnittstelle `IUnknown`. Daher deklariert und implementiert `TInterfacedObject` für uns jede der drei Methoden `QueryInterface`, `_AddRef` und `_Release` von `IUnknown` und eignet sich aus diesem Grund als Basis für jede weitere Klasse, die eine Schnittstelle implementiert.

```
TInterfacedObject = class(TObject, IUnknown)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

Das eigentliche Arbeitspferd ist die konkrete Klasse mit Implementierung in der DLL, die von der Basisklasse wie von der Schnittstelle erbt:

```
TIncomeRealIntf = class (TInterfacedObject, IIncomeInt)
    FRate: Real;
```

```
    function Power(X: Real; Y: Integer): Real;
  protected
    function GetRate: Real;
  public
    constructor Create;
    destructor destroy; override;
    function GetIncome(const aNetto: Currency): Currency; stdcall;
    .....
    property Rate: Real read GetRate;
  end;
```

Wenn Sie mit DLL+ arbeiten, muss noch die Referenz zum exportieren befähigt werden. Der eigentliche Export befindet sich in einer globalen Funktion, die als Rückgabewert die Referenz über den Constructor der Klasse zurückgibt. Dann benötigt man nur noch das Schlüsselwort `exports`.

```
function CreateIncome: IIncomeInt; stdcall;
begin
  result:= TIncomeRealIntf.Create;
end;
exports CreateIncome;
```

1.3 Speicher und Referenz

Da die Klasse `TInterfacedObject` die Methoden von `IUnknown` implementiert, führt sie automatisch die Referenzzählung und die Speicherverwaltung für Schnittstellenobjekte durch.

Eines der grundlegenden Konzepte beim Entwurf von Schnittstellen besteht in der Sicherstellung der Referenzverwaltung für die Objekte, welche die Schnittstellen implementieren. Die IUnknown-Methoden `_AddRef` und `_Release` ermöglichen die Implementierung dieser Funktionalität. Sie überwachen die Existenz eines Objekts mit Hilfe eines Referenzzählers. Dieser wird jedesmal erhöht, wenn eine Schnittstellenreferenz an einen Client übergeben wird. Sobald der Referenzzähler den Wert Null erreicht, wird das Objekt freigegeben.

Bei einem Objekt, das ausschliesslich durch Schnittstellen referenziert wird, ist eine manuelle Freigabe nicht mehr nötig. Seine Freigabe erfolgt automatisch, wenn der Referenzzähler den Wert Null erreicht. In unserem Beispiel erfolgt das Erzeugen der **lokalen** Schnittstellenreferenz (`incomeIntRef:= createIncome)` direkt auf Mausklick, wir erzwingen also mit bei jedem Durchlauf die Berechnung der Zinsen und stellen fest, das Objekt wird bei jeder neuen Berechnung wieder mit `destroy` aus dem Speicher entfernt. Warum, weil bei jedem neuen Konstruktoraufruf `IUnknown` automatisch das alte Objekt entfernt, obwohl kein `_Release` oder ähnliches im Spiel ist! Denn beim Verlassen der Prozedur wird der Zähler auf Null gesetzt. Auch beim Schliessen des Client erfolgt der gleiche Freigabemechanismus und auch hier wird `destroy` automatisch aufgerufen.

```
destructor TIncomeRealIntf.destroy;
begin
  messagebox(0,'automatic release','TIncomeRealIntf',mb_OK)
end;
```

Im Experiment lässt sich der Automatismus mit einem Expliziten Setzen von `AddRef` auch mal austricksen, d.h. wir treiben den Zähler künstlich in die Höhe und geben vor, dass mindestens eine Referenz noch im Spiel ist, wir verhindern also ständig die Freigabe. Hier hilft nur eine Release wieder, d.h. je nach Design und Dynamik sind die Automatismen eher hinderlich:

```
procedure TfrmIncome.BitBtnOKClick(Sender: TObject);
begin
```

```
  incomeIntRef:=createIncome;
   with incomeIntRef do begin
    if QueryInterface(IIncomeInt, incomeIntRef) = S_OK
    then begin
      //_addRef;to test
      SetRate(strToInt(edtZins.text),
                      strToInt(edtJahre.text));
.........
```

Wenn die Implementierung der Klasse der Schnittstellendefinition entspricht, ist die Schnittstelle vollständig polymorph: Zugriff und Verwendung der Schnittstelle ist also für alle Implementierungen dieser Schnittstelle identisch. Wir könnten also noch eine zweite Klasse, z.B. `TIcomeReal2Intf` Implementieren, die eine andere Art der Berechnung durchführt. Dann einfach Referenz im Client ändern und es steht eine andere Implementierung derselben Methoden im Einsatz.

Mit Schnittstellen eröffnet sich eine neue Möglichkeit zur Trennung von Einsatz und Implementierung einer Klasse. Zwei Klassen können also dieselbe Schnittstelle verwenden, auch wenn sie keine Nachkommen derselben Basisklasse sind.

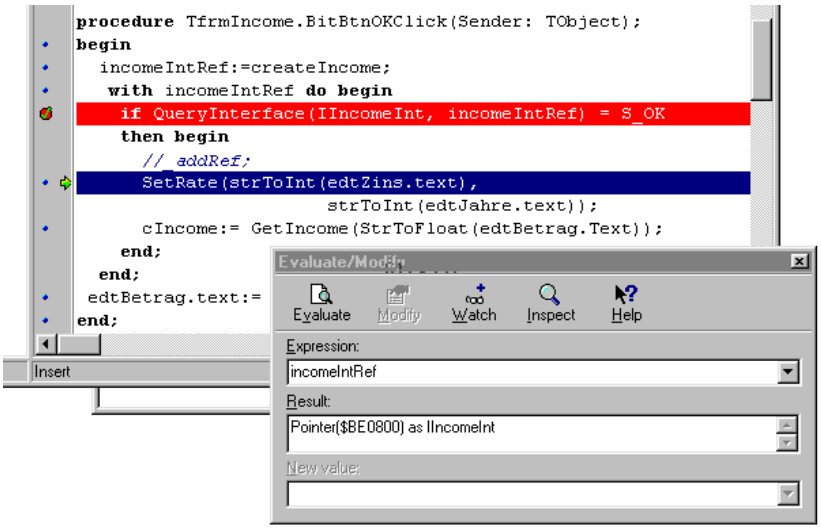

Abb. 3: Sicheres Typecasting mit QueryInterface

Wenn Klassen Schnittstellen implementieren, können sie den Operator `as` für die dynamische Bindung an die Schnittstelle verwenden. Spätestens hier wird eine IID als GUID benötigt, welche sich direkt im Code-Editor mit SHIFT+CTRL+G fabrizieren lässt.

Eine GUID (Schnittstellen-ID) ist ein binärer 16-Byte-Wert, der eine Schnittstelle weltweit eindeutig bezeichnet (Wahrscheinlichkeitsrechnung). Wenn eine Schnittstelle eine GUID hat, können Sie über eine Schnittstellenabfrage Referenzen auf ihre Implementationen abrufen.

Mit Hilfe des Operators `as` können Schnittstellenumwandlungen durchgeführt werden. Dieser Vorgang wird auch als Schnittstellenabfrage bezeichnet und macht dann Sinn, wenn so mehr als 3 Interfaces im Spiel sind. In Abb. 3 ist ersichtlich, dass beim Aufruf von `QueryInterface` immer ein `as` im Spiel ist. Eine Schnittstellenabfrage generiert auf der Basis des (zur Laufzeit vorliegenden) Objekttyps aus einer Objektreferenz oder einer anderen Schnittstellenreferenz einen neuen Ausdruck vom Typ einer Schnittstelle. Die Syntax für eine Schnittstellenabfrage lautet:

Objekt as Schnittstelle

Objekt ist entweder ein Ausdruck vom Typ einer Schnittstelle oder Variante, oder er bezeichnet eine Instanz einer Klasse, die eine Schnittstelle implementiert. Also, angenommen wir haben zwei

Interfaces und rufen den Konstruktor für die erste Schnittstelle auf. Diese Instanz möchte wir nun auf die zweite Schnittstelle `IUserAccount` anwenden und erhalten eine Umwandlung unserer ersten Instanz, ohne erneuten Aufruf des Constructors.

Beispiel:

```
var incomeIntRef: IIncomeInt;
    userRef: IUserAccount;
begin
  incomeIntRef:= TIncome.create
  incomeIntRef.setRate();
  userRef:= incomeIntRef as IUserAccount
  userRef.setWAPAccount()
```

Im Gegensatz einer direkten Typumwandlung mit `as` (die bei Fehler eine Exception auslöst), lässt sich mit `QueryInterface` einfach und locker nachfragen, ob ein definiertes Interface vom Objekt unterstützt wird:

```
  if QueryInterface(IIncomeInt, incomeIntRef)
                      = S_OK then begin
```

Wenn Objekt den Wert `NIL` hat, liefert die Schnittstellenabfrage als Ergebnis `NIL`. Andernfalls wird die GUID der Schnittstelle an die Methode `QueryInterface` übergeben. Wenn `QueryInterface` keinen gültigen Vergleich anbringen kann, wird NIL gesetzt. Ist der Rückgabewert dagegen Null oder gültig (d.h. das Objekt unterstützt mit seiner Implementation die Schnittstelle) ergibt die Schnittstellenabfrage eine Referenz auf das Objekt. Das heisst aber nicht, dass `QueryInterface` in irgend einer Art Objekte instanziert, ein erstes Objekt muss in jedem Fall vor dem Aufruf von `QueryInterface` instanziert worden sein.

Tipps zu Interfaces:

- Die Standard-Aufrufkonvention für Schnittstellen ist `register`. Schnittstellen, die man von verschiedenen Modulen gemeinsam benutzt, sollten die Methoden mit `stdcall` deklarieren. Dies gilt insbesondere, wenn diese Module in verschiedenen Programmiersprachen erstellt wurden.
- Wenn Sie den Operator `as` oder `QueryInterface()` mit einer Schnittstelle einsetzen, muss diese eine zugeordnete IID besitzen.
- In jedem Interface ist unterstellt, dass die Basisklasse oder ein Vorfahr die Methoden von `IUnknown` zur Referenzzählung implementiert.
- Die einfachste Art, diese Methoden zu implementieren, besteht darin, die implementierende Klasse von `TInterfacedObject` in der Unit *System* abzuleiten.
- Wenn es sich beim Objekt um eine VCL-Komponente oder ein Steuerelement handelt, unterliegt dieses Objekt der Speicherverwaltung von `TComponent`. Mischen Sie also nicht VCL Referenzverwaltung mit der COM-Referenzzählung.
- Die Funktion `GUIDToString()` konvertiert eine GUID in einen String aus druckbaren Zeichen. Jede GUID wird in einen eindeutigen String umgewandelt.
- Im Unterschied zu einer **abstrakten Klasse** enthält eine Schnittstelle überhaupt keine implementierte Methode, alle Methoden sind abstrakt. Abstrakte Klassen lassen sich verwenden, wenn bereits ein bestimmtes Grundverhalten in den abgeleiteten Klassen vorhanden sein soll.

1.4 Delegieren wir mal

Eine Möglichkeit der Wiederverwendung von Quelltext in Schnittstellen besteht darin, ein Objekt in die Schnittstelle aufzunehmen bzw. die Schnittstelle als Objekt in eine andere einzufügen. Die VCL verwendet zu diesem Zweck Properties, die Objekttypen darstellen. Zur Unterstützung dieser Struktur für Schnittstellen stellt Object Pascal das Schlüsselwort `implements` bereit, mit dem die Implementierung einer Schnittstelle ganz oder teilweise einem untergeordneten Objekt übertragen werden kann. So erreicht man eine zentralisierte Instanzierung mit einem Objekt. Dies nennt man Delegation. Es gibt aber ein Delegieren vom Typ einer Schnittstelle (Beispiel 1) und ein Delegieren vom Typ einer Klasse (Beispiel 2).

```
Type //Beispiel 1
IMyInterface = interface
  procedure P1;
  procedure P2;
end;
TMyClass = class(TObject, IMyInterface)
  FMyInterface: IMyInterface;
  property MyInterface: IMyInterface
              read FMyInterface implements IMyInterface;
end;
```

Die folgende Aggregation ist eine andere Möglichkeit, Sourcecode über diese Mechanismen wiederzuverwenden. Bei der Aggregation enthält ein übergeordnetes, umgebendes Objekt ein untergeordnetes. Vorteil: Es lassen sich bestehende Klassen, die nicht von einem Interface abstammen, einbinden. Das untergeordnete Objekt implementiert Schnittstellen, die (nur) für dieses übergeordnete Objekt verfügbar sind. Hier ist also der Unterschied zwischen Delegation und Aggregation zu sehen. Bei der Aggregation sind nebst der Schnittstelle mindestens zwei Objekte im Spiel, so dass Objekt A an ein Objekt B delegiert. Die VCL enthält beispielsweise Klassen, welche die Aggregation unterstützen. Das Objekt gibt also vor, selbst Interfaces zu implementieren, die in der Tat von anderen Objekten stammen.

```
IMyInterface = interface //Beispiel 2
  procedure P1;
  procedure P2;
end;
TMyImplClass = class
  procedure P1;
  procedure P2;
end;
TMyClass = class(TInterfacedObject, IMyInterface)
  FMyImplClass: TMyImplClass;
  property MyImplClass: TMyImplClass
               read FMyImplClass implements IMyInterface;
```

Fazit: Schnittstellen bieten gegenüber abstrakten Klassen mehr verfügbare Merkmale an, sind aber auch restriktiver in der Handhabung. Ich erinnere daran, dass ein Interface selbst kein Grundverhalten implementieren darf. Wenn ein Grundverhalten nicht nötig ist, steht dem Einsatz von Zwischengesichtern eigentlich nichts mehr im Wege. Das komplette Beispiel läuft übrigens auch unter Kylix, wie Abb. 4 als Starter zeigt. Der Code der SO liegt der CD-ROM bei. Wobei DLL+ dann zu SO+ (für Shared Objects Plus) mutiert. Warum sich aber die SO-Grösse verdreifacht und welche anderen wundersamen Libraries sonst noch geladen werden, steht dann in einem anderen Kapitel.

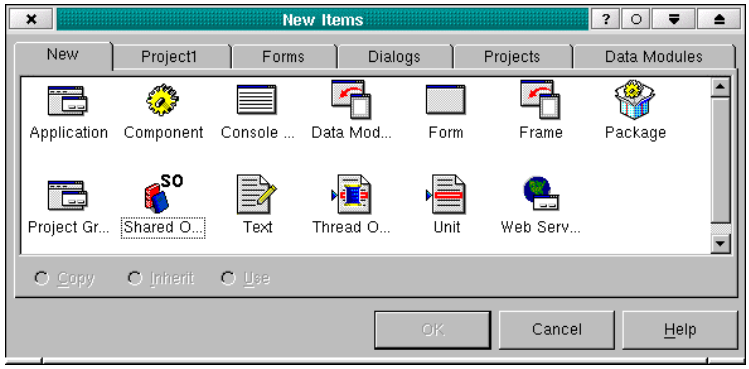

Abb. 4: Start einer SO als DLL+ in Kylix

Literatur & Links

[MK00], Max Kleiner: “UML mit Delphi“, Software & Support Verlag

[DE01], Das Plus beim Export, Der Entwickler 4.2001

http://www.delphi3000.com – Umfangreicher Delphipool mit Designkriterien

Max Kleiner

Beispiel auf der CD-ROM