1 Klassenbau in OP

Programmieren geht über Studieren oder eben doch modellieren. Leistungsfähige Tools für eine durchgängige Entwicklung mit Teamunterstützung sind mehr denn je gefragt. Kürzlich wurde UML 2.0 als Standardisierung vorgelegt. Man spricht von MDA (Model Driven Architecture) welche eine konkretere Generierung von Schnittstellen erlaubt. Vermehrt bieten die Tools, nebst der Codegenerierung, auch das Erstellen von Code aus Patterns an. Schwerpunkt dieser separaten Markt-Übersicht sollen Generatoren und das Austauschformat XMI, nebst der Anbindung an Delphi, sein. Hier nun will ich den Bau eines BusinessObjects zeigen.

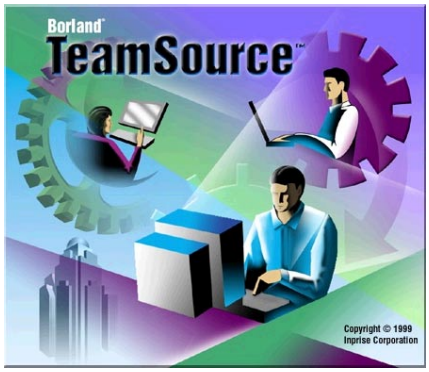
1.1 Codegenerierung in OP

Als reiner OP-Entwickler (ObjectPascal) wurde man in den letzten Jahren mit eingebauter Codegenerierung der Tools nicht gerade verwöhnt. Eigentlich gab es nur Delphi mit ModelMaker. Doch das Blatt hat sich gewendet. Mittlerweile gibt es für OP nebst Delphi und Kylix auch noch Free Pascal und Virtual Pascal, die mit leistungsfähigen Compilern die Sprache ObjectPascal teilweise unterstützen.

Auch mit dem Aufkommen der CLX sollten sich diese 4 Compiler eigentlich näherkommen (Leider haben sich unter der Haube der VCL, seit dem Aufkommen der CLX, auch einige Ungereimtheiten eingeschlichen). Zudem gibt es diverse Scriptsprachen (DelphiScript, PascalScript etc.), die mehr oder weniger eine Teilmenge von OP implementieren. Das haben auch die Tool-Hersteller gemerkt, die nun nebst Java und C++ vermehrt auch OP-Generierung anbieten.

Auch im Bereich der Anlagensteuerung und Automatisierung wird ObjectPascal vermehrt eingesetzt. Hier sind robuste Anwendungen auf einem Leitrechner mit Ethernet als Feldbus gefragt. Die meisten Tools warten also mit einer integrierten Code-Generierung für OP auf. Neu hinzugekommen ist objectiF, das endlich Delphi unterstützt (oder waren es meine hartnäckigen Mails ?). Viele Tools besitzen mittlerweile auch eine sprachengebundene Typisierung der Komponentenbibliothek an (VCL, MFC etc.), somit wird die Generierung direkter an den Zielcode gebunden.

Ist irgendwie auch naheliegend, da die Tool Hersteller die Libraries schon als eine Art „Code Patterns“ aufbereiten, welche sich dann bei Bedarf abrufen lassen, wie mit Streams oder Collections zu beweisen ist. In der VCL bspw. sind diverse Design Patterns wie Observer, Lock oder Composite implementiert worden.



Während den Workshops taucht manchmal die Frage auf, wo denn die Codegenerierung aufhört. Auch wenn es mittlerweile standardisierte Codesnippets wie getter und setter-Methoden, IDL-Generierung oder Listenklassen gibt, die sich mit Code-Templates auch erzeugen lassen, hört die eigentliche Generierung innerhalb der Methode auf.

Bei der Implementierung einer Methode bzw. einer Member Funktion in OP lassen sich die algorithmischen Konstrukte verwenden, die unter den bewährten Struktogrammen nach Nassi/Shneiderman, DIN 66001 oder Flowcharts auch eine grafische Darstellung anbieten. Warum also nicht die klassischen Struktogramme zur Detailspezifikation von Methoden einsetzen, wenn es die Komplexität erfordert ?

1.2 Das Klassenfinden

Wenn das Klassendiagramm ja so zentral ist, sollten doch viele Generatoren sich auf dieses Diagramm spezialisiert haben. Dies ist auch so, nur lässt sich auch aus Sequence- oder State Event-Diagrammen Code generieren, was viele Tools noch nicht anbieten. Sogar aus dem Package-Diagramm ist es auf Stufe Design Pattern möglich, Muster fast konstruktionsfertig zu erzeugen. Jedes Pattern beschreibt ja ein Problem, das immer wieder vorkommt und sich da-

durch wiederholt auflösen lässt. Dabei bezeichnet man mit dem Ausdruck Pattern sowohl das Ergebnis, das durch die Anwendung der Regel entsteht, als auch die Regel selbst. Denn es ist effektiv so: Man muss das Problem erkennen, bevor man die Lösung einsetzen kann.

So am Rande kann auch Delphi einiges an Code generieren, interessant ist auch die Code-Optimierung, die selten zur Sprache kommt.. Die Direktive $O steuert die Code-Optimierung. Der Compiler führt eine Anzahl von Optimierungen durch, indem er bspw. Variablen direkt in CPU-Registern platziert, doppelte Teilausdrücke eliminiert und schnelle Induktionsvariablen generiert. Die Optimierungen des OP-Compilers von Delphi führen zu keinerlei Änderungen am Verhalten des Programms, so die Delphi-Hilfe.

Bevor ich auf die einzelnen Tools in einem separaten Bericht eingehe, machen wir nun eine kleine Wanderung auf dem objektorientierten Weg, d.h. wir besuchen den "Analy-see" im "Digi-tal" und beobachten die Durchgängigkeit von UML anhand eines Exempels.

1.2.1 Vom Bau eines Business Objekts

Wir bauen uns nun ein Business-Objekt, das sozusagen als Referenz zu einzelnen Tools dienen soll. Einige Tipps zu Klassenbau und der visuellen Darstellung folgen. Mit diesem Business-Objekt will ich zeigen, wie sich mit UML auch bei kleinen Projekten ein durchgängiger Entwicklungszyklus lohnt.

Die Anforderungen an unser Objekt sind bescheiden: Das Teil soll uns ermöglichen, Lohnberechnungen und Änderungen durchzuführen und die Mutation der Daten als XML exportieren zu können. Vor allem sind wir an Lohnerhöhungen interessiert und dazu ist in der InterBase-Demo IBLocal mit den Tabellen EMPLOYEE und SALARY_HISTORY schon alles bestens vorbereitet. Sie benötigen also nur den Code und können direkt auf dem vorhandenen Datenmodell aufsetzen.

Als erstes nehmen wir die Geschäftsprozesse auf (Haupt- und Teilgeschäftsprozesse abgrenzen, Schnittstellen zwischen Prozessen festlegen, Geschäftsprozesse in einem Ist-Modell abbilden). Als Soll-Modell dient uns das zweite UML-Diagramm das Activity. Die Analyseergebnisse dienen als Ausgangspunkt für die Definition von Anforderungen an eine künftige Systemumgebung beim Nutzer und als Basis für eine nachfolgende Klassenmodellierung sowie deren Umsetzung mit Hilfe der weiteren Diagramme in der Praxis.

Z.B. kennen Sie das Problem, dass man eine Aktivität von einer anderen abhängig macht. Bei einer Abhängigkeit im Activity dient dazu der Synchronisationsbalken, der die einzelnen Bedingungen oder Trigger aufnehmen kann. Ein einfacher Synchronisationsbalken bedeutet, dass die ausgehenden Trigger erst in Aktion treten, wenn alle eingehenden vorhanden sind. Dies entspricht einer Und-Bedingung. In unserem Beispiel in Abb.1 wird die Aktivität <get Employee> oder <check Contract> erst möglich, wenn die Bedingung [valid] der Aktivität <login> erfüllt ist.

Die Aktivität <manage Salary> wiederum kann eintreten, wenn entweder der Angestellte auf der Lohnliste ist oder er einen Vertrag akzeptiert hat. Dies entspricht einer Oder-Bedingung, erkenntlich am direkten Einfliessen der zwei Pfeile (Trigger), in die Aktivität <manage Salary>. Der abschliessende Export in ein XML-File kann nur erfolgen, wenn die Mutationen gespeichert sind. Somit sollte jeder Trigger auch eine Bedingung enthalten, welche in UML als eckige Klammer, wie z.B. [saved], notiert wird.

So, fürs erste haben wir den Geschäftsprozess analysiert und die Anforderungen ungefähr abgesteckt. Im weiteren Verlauf gehen wir von der Aktivität <manage Salary> aus und spezifizieren daraus eine erste Klasse. Es ist aber nicht so, dass jede Aktivität eine Klasse für sich im Hintergrund hat, es können auch n-Aktivitäten in eine Klasse münden, d.h. sie teilen sich die Klasse. Hier sind eben schon erste Ansätze der „richtigen“ Klassengrösse auszumachen.

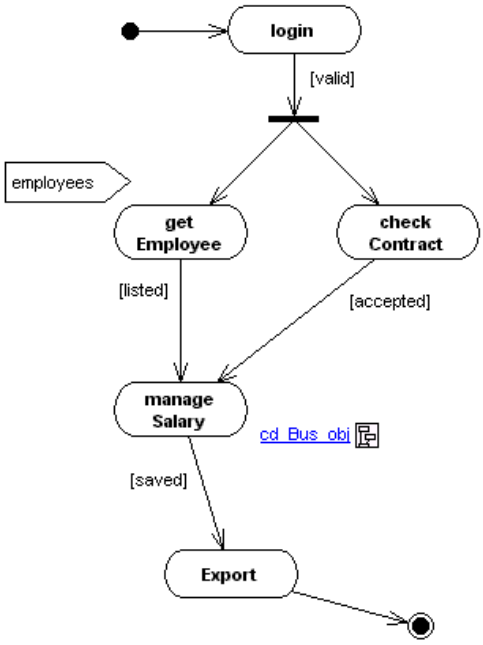


Abb.1: Das Activity mit den Bedingungen

Ein Business Objekt sollte eigentlich folgende Kriterien erfüllen:

- Die Klasse erbt von einem Daten-Provider
- Die Abfragen sind Teil der Klasse
- Logik und Berechnung erfolgt in der Klasse
- Die Methoden erscheinen als public-Services
- Das Objekt ist unabhängig von visuellen Komponenten

Ich komme zum Klassendiagramm. Lassen Sie uns das Beispiel in ObjectPascal kurz diskutieren (erben, Abfragelogik, Autonomie). Die Klasse `TBusinessObj` hat zwei öffentliche Methoden, die das Öffnen der SQL-Menge und das Ändern der Lohnsumme erlauben. Im weiteren ist die Berechnung der Lohnsumme, z.B. Angabe in Prozenten, eine „private“ Angelegenheit, welche mit der Funktion `calcSalary` erfüllt ist:

```
TBusinessObj = class (TDataModule1)
private
  function calcSalary(salary: double): Double;
  procedure changeGrade(amount: integer);
public
  constructor Create(aOwner :TComponent); override;
  destructor destroy; override;
  procedure changeSalary(amount: double);
  function getFullName: string;
  function getOldSalary: Double;
  function open QueryAll: Boolean;
  function open QuerySalary(qryID: integer): Boolean;
end;
```

Bei einer Änderung der Lohnsumme, in unserem Falle das InterBase-Feld SALARY, wird im gleichen Atemzug noch eine Berechnung anhand einer Business-Rule durchgeführt. Die Query selbst wurde vorgängig mit der Prozedur `open_QuerySalary`() geöffnet und dem Business Objekt vererbt, da das Objekt selbst von `TDataModule` erbt. Das Objekt ist auch für die Instanzierung des `TDataModule` verantwortlich, so dass im Constructor zur Laufzeit mit `inherited create(aOwner); database1.open`; die Datenbank geöffnet wird.

```
procedure TBusinessObj.changeSalary(amount: double);
begin
  with query1 do begin
    try
      edit;
       FieldByName('SALARY').asFloat:= calcSalary(amount);
      post;
    except on E: Exception
        do showmessage('Salary out of range' + E.message);
    end;
  end
```

`CalcSalary` alleine ist nicht für die ganze Business-Rule verantwortlich. Bei globalen Regeln, die jeden Client betreffen müssen, fährt man am besten mit einer Stored Procedure oder einer Check-Constraint. Im Falle der IBLocal wird die eingegebene Lohnsumme auf dem Server validiert, d.h. die Summe sollte sich in einem bestimmten Min.-Max.-Bereich befinden, der sich anhand des Job-Grade und dem COUNTRY ermitteln lässt:

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT INTEG_30
  CHECK ( salary >= (SELECT min_salary FROM job WHERE
       job.job_grade = employee.job_grade AND
       job.job_country = employee.job_country) AND
  salary <= (SELECT max_salary FROM job WHERE
       job.job_grade = employee.job_grade AND
       job.job_country = employee.job_country))
```

Bis zum jetzigen Zeitpunkt hat noch kein Export stattgefunden. Diesen wollen wir jetzt als XML-Format realisieren und hier sind Streams jederzeit willkommen. Während des Klassendesign starten wir meistens mit den public-Methoden, die dann mit Hilfe von CRC-Karten auch die privaten Funktionen hervorbringen.

Dieses iterative Spiel führt dann zu den nötigen Methoden. Um Klassen wiederverwenden zu können, sollten sie möglichst wenig mit anderen Klassen kommunizieren, denn je weniger eine Klasse von Ihrer Umgebung weiss, desto universeller ist sie einsetzbar (Autonomie).

Unsere Klasse `TDataToXML` hat nur ein property `pBuffer`, das die Grösse des Buffers aufnimmt. Zudem besteht eine Abhängigkeit zur Klasse `TFileStream`. Die entsprechenden getter und setter erzeugt ModelMaker automatisch. Andere Tools hatten beim Erzeugen dieser Klasse schon mehr Mühe, tja auf die richtige Konfiguration und Instrumentalisierung kommt es an:

```
Type
  TDataToXML = class (TObject)
  private
    FpBuffer: PChar;
    function getFieldStr(field: Tfield): string;
    procedure writeData(stm: TFileStream; fld:TField; s:string);
    procedure writeFileBegin(stm: TFileStream; dSet: TDataSet);
    procedure writeFileEnd(stm: TFileStream);
    procedure writeRowEnd(stm: TFileStream; isTitle: boolean);
    procedure writeRowStart(stm: TFileStream; isTitle: boolean);
  protected
    procedure SetpBuffer(abuffer: PChar); virtual;
    procedure writeString(stm: TFileStream; s:string); virtual;
  public
    constructor create;
    destructor destroy; override;
    procedure DatasettoXML(dSet: TDataset;
                           fileName: string); virtual;
```

```
    property pBuffer: PChar read FpBuffer write SetpBuffer;
  end;
```

Das Business-Objekt mit dem zugehörigen Export-Objekt ist soweit erstellt. Das Diagramm in Abb.2 ist nun fähig die Code-Rümpfe inklusive Parametrisierung zu erstellen. `TForm1` besitzt den Member `datEmployee` vom Typ `TBusinessObj`. Diese Beziehung wurde als <composition> modelliert, da `datEmployee` immer auch auf das Datenmodul angewiesen ist. Die <composition> ist stärker als <aggregation> und meint physik. Einbindung, in unserem Fall `datEmployee.dataSource1.dataSet` oder `cmxEmployee.items.add`, meistens in Komponenten oder Kollektionen zu finden und als schwarz gefüllte Raute notiert. Eine Komposition bedeutet auch eine starke Abhängigkeit von einem anderen Objekt. ModelMaker erkennt auch diese Beziehung. Folgender Code demonstriert diese Abhängigkeit und füllt zur Laufzeit eine Combobox mit den SQL-Daten als Auswahlmöglichkeit:

```
with datEmployee.dataSource1 do begin
  while not dataSet.EOF do begin
    cmxEmployee.items.add((dataSet.fieldValues['EMP NO']));
    dataSet.next;
  end;
end;
```

Es sind also alle 4 Relationen im Klassendiagramm enthalten: Die Vererbung ist klar mit dem Business-Objekt von `TDataModule1` ersichtlich. Das Datenmodul wiederum hat eine Aggregation zu `TQuery` als offene Raute notiert. Die Komposition erkennen wir als geschlossene Raute, da die Instanz `datEmployee` immer noch eine weitere Instanz im Aufruf benötigt. Als vierte Beziehung ist eine Assoziation im Diagramm ersichtlich. Die Assoziation <xmlExport> ist eine Beziehung zur Laufzeit, d.h. die Klasse kennt die Instanz nicht zur Designtime, typischerweise sind es lokale Instanzen die aufgerufen und wieder zerstört werden, wie folgender Code zeigt:

```
with TDatatoXML.create do begin
try
  dataSetToXML(datEmployee.query1, 'salaryXport.xml');
finally
  free;
end
```

Eine Assoziation erkennt man als einfache Linie. Der Vollständigkeit halber gibt es noch als gestrichelte Linie die Abhängigkeit <dependency> wobei die eher schwach definiert ist. Zusammenfassend lässt sich folgende Tabelle aufstellen:

Abb.2: Die Beziehungen zwischen Objekten

	Beziehung auf Stufe	Bekannt zur
Vererbung	Klasse	Designtime
Assoziation	Objekt	Runtime, Loadtime
Aggregation	Objekt	Designtime
Komposition	Mind. 3 Objekte	Designtime

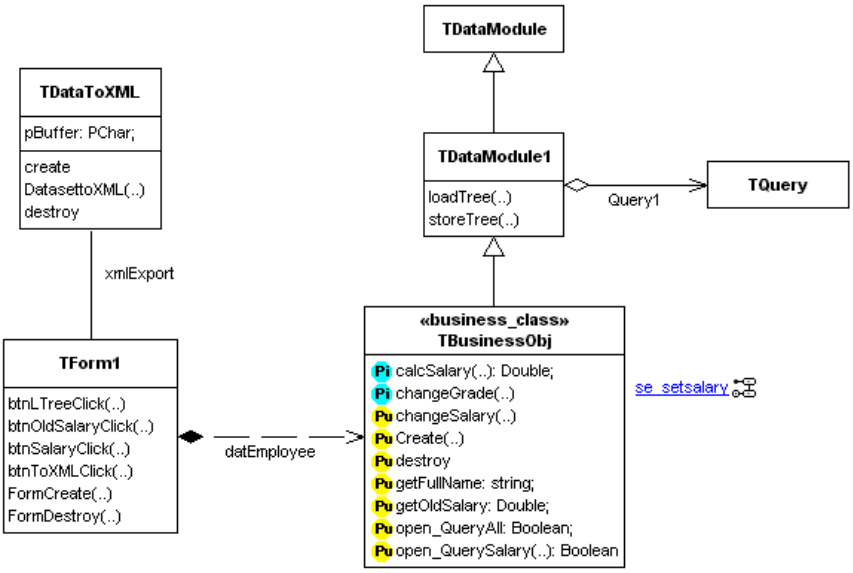


Abb. 3: Alle Relationen im CD auf einen Blick

Weiter geht es mit einer genaueren Definition des Business-Objekts mit Hilfe eines Zustandsdiagramms. Beim State-Event handelt es sich um eine detailliert Sicht der Klasse. Jedes Zustandsdiagramm ist also einer Klasse zugeordnet und beschreibt deren Verhalten genauer. Zwischen den Zuständen fliessen Ereignisse als Nachrichten oder kurzzeitige Aktionen werden ausgelöst. Ist eine Klasse mit eigenen Zustandsvariablen bestückt oder will man die möglichen Ereignisse mit ihren Übergängen genauer festhalten, dann sind State-Events erste Wahl:

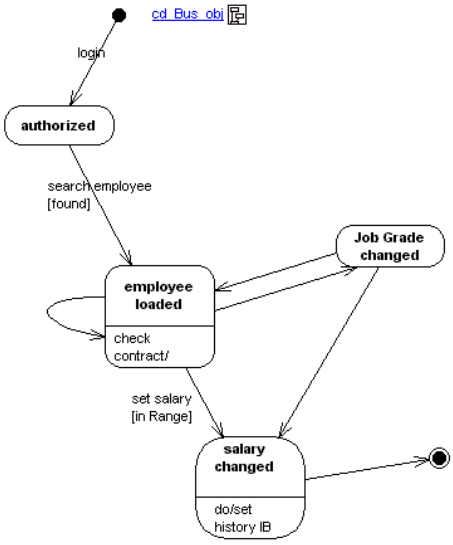


Abb. 4: Die erlaubten Zustandswechsel des Objekts

Interessant ist die Transition von <employee loaded> zu <salary changed> die nur erfolgt, wenn der Betrag gemäss der InterBase-Constraint im erlaubten Bereich liegt. Eine Ausnahmebehandlung wird nicht modelliert. Einerseits ist zu erwarten, dass die Verwendung von State-Events die Entstehung bestimmter Folgefehler reduziert; andererseits stellt sich die Frage, ob durch die spezifischen Mechanismen der Zustandsvariablen, wie das Abfragen des Zustandes bei Auftreten eines Ereignisses, bestimmte Probleme erst entstehen.

Wir jedenfalls sind jetzt im Zustand der abgeschlossenen Details, das nächste Diagramm nimmt schon Teile der Implementation vorweg und bewegt sich in Richtung Integration. Die Rede ist vom Sequenz-Diagramm. Das Bild beschreibt das Zusammenwirken verschiedener Objekte für einen bestimmten Anwendungsfall. Es definiert somit die

interne Sicht eines Use Case, beschreibt also, wie sich der Use Case innerhalb der Anwendung realisieren lässt. Der Nachrichtenaustausch zwischen den Instanzen ist hier in zeitlicher Reihenfolge sichtbar. Im Diagramm wird der Use Case <manage Salary> als Szenario realisiert. Zum jetzigen Zeitpunkt entstehen die Parameter mit den zugehörigen Typen, d.h. ein Objekt muss jetzt mit anderen Objekten auf verträgliche Art kommunizieren können:

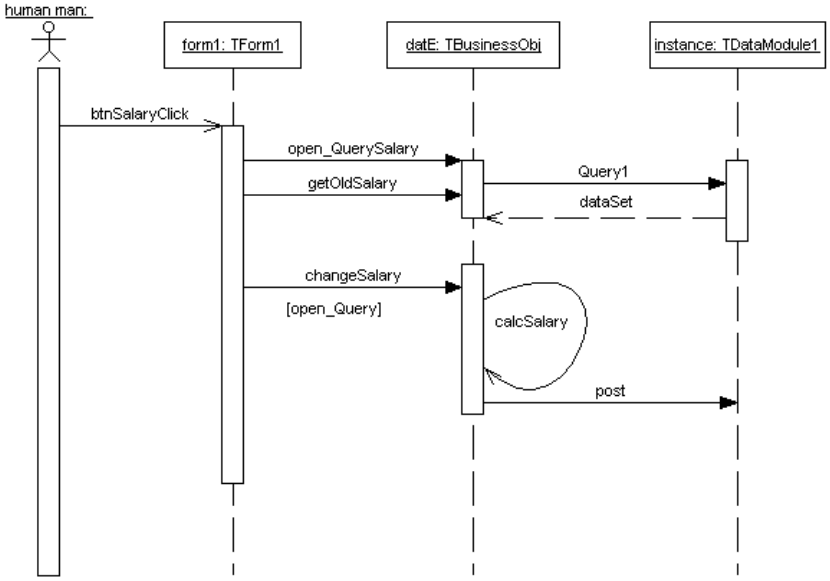


Abb. 5: Dynamischer Ablauf eines Szenarios

Wir sind am Ende unseres "geschäftlichen" Objektbaues angelangt, es würden noch Package-, Component und Deployment Diagramm folgen, die lassen sich aber aus der vorhandenen Projektstruktur und den Units leicht selbst erstellen. Bei einem Beispiel dieser „kleinen Grösse“ lohnen sich diese Diagramme wenig, dienen höchstens der vollständigen Dokumentation und weiteren Architekturüberlegungen. Auf der CD sind die restlichen Bilder enthalten.

Interessanter ist wohl sich am Schluss sich zu überlegen, wie denn die „richtigen“ Business-Objekte auf ihre Daten zugreifen. Richtige Business-Objekte haben oder können eine Persistenzfunktion gebrauchen, so dass auch der Zugriff und die Navigation auf eine relationale Datenbank objektorientiert, d.h. mit Methoden, erfolgen kann. Es ist mir also möglich, eine Personen-Tabelle wie ein Objekt zu behandeln und somit muss ich keine SQL-Statements absetzen:

```
 oPerson:=TPerson.Create;
try
   oPerson.Person ID:=30;
   oPerson.DBRead(nil);
  if oPerson.Found then
    S MBox(oPerson.LastName+' '+oPerson.FirstName);
finally
  oPerson.Free;
end;
```

Wie aber wenn wir es mit Relationen zwischen den Tabellen zu tun haben, auch hier besticht ein objektorientierter Zugriff. Die Beispiele stammen übrigens aus MetaBASE. Mit dem Framework MetaBASE ist es möglich, ein Datenmodell komplett objektorientiert zugänglich zu machen. Der Meta-Layer erlaubt ein einfaches Portieren und MetaGen speichert die aus dem Datenmodell generierten Objekte in ein sogenanntes Object Stream File. Später wird dieses File in der Entwicklungsschicht von Delphi oder von der Anwendung während der Laufzeit gelesen. Hier also ein relationaler Zugriff mit Methoden:

```
oPerson:=TPerson.Create;
oAddressList:=TAddressList.Create;
try
  oPerson.Person_ID:=30;
  oAddressList:=TAddressList.DBReadAllRelatedToObject(oPerson);
```

```
  for i:=0 to oAddressList.Count-1 do
    S_MBox(oPerson.LastName+' '+oAddressList.Adresses[i].Town);
finally
  oPerson.Free;
  oAddressList.Free;
end;
```

Mehr über MetaBASE erfahren sie unter: www.gs-soft.com

Die Klassenmethode `TAddressList.DBReadAllRelatedToObject(oPerson)` besitzt nun die Fähigkeit ein Adressliste anhand der übergebenen Person basierend auf dem relationalen Datenmodell zu erstellen. Eine Klassenmethode kann über eine Klassenreferenz oder eine Objektreferenz aufgerufen werden. Hier wird sie über die Klassenreferenz aufgerufen. Bei einer Objektreferenz erhält self als Wert die Klasse des betreffenden Objekts. Somit lässt sich die Adressliste an beliebige Instanzen von `TAddressList` zuweisen.

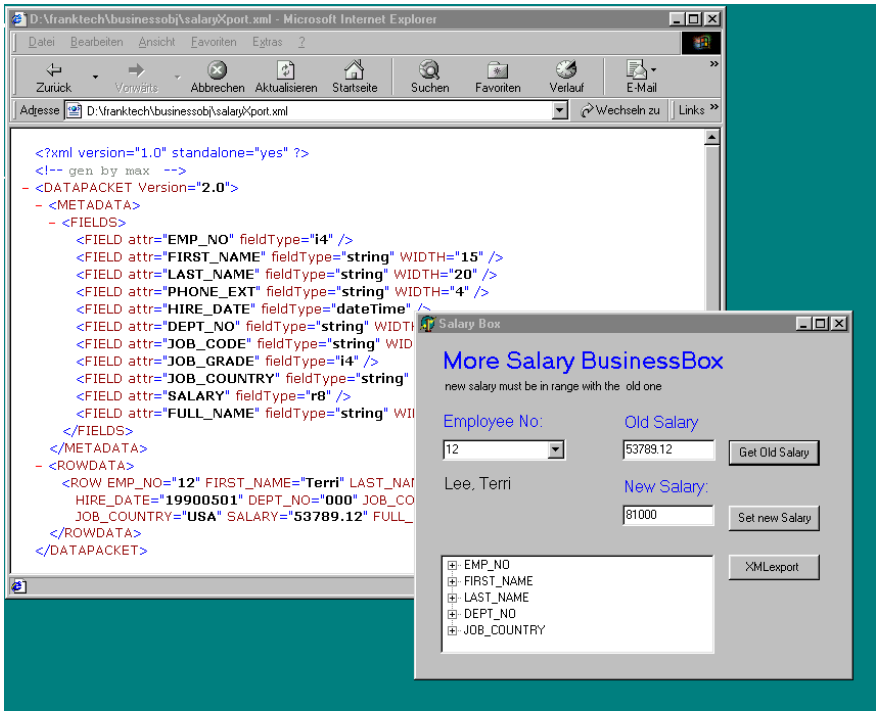


Abb. 6: Unsere Business Box mit XML-Export

Nach diesem explorativen Businessbau wollen wir die erworbenen Kentnisse innerhalb einer kurzen Schrittfolge zusammenfassen, sozusagen ein Strickmuster oder ein Vorgehen der Klassenfindung:

1.2.2 Einteilung der Klassen in Units (Packages) vornehmen

Nachdem die Klassen durch die Techniken der Klassenfindung gefunden wurden (Activity Diagramme oder Design Patterns), bestimmen wir das nähere Verhalten und die Architektur gemäss folgenden Schritten, die aber hochgradig iterativ ablaufen (auch Refactoring genannt):

Document (Model), Controller, View (DCV) ist ein Prinzip zur Verteilung der Funktionalität auf Klassen, das auch als Vorbereitung zu mehrschichtigen (multi-tier) Klassen/Komponenten dient:

Verarbeitungsdominante Klassen realisieren Kontrolle und Steuerung von 1 oder n Use Case und werden auf <entity> oder <control> -Klassen verteilt.

Dialogdominante Klassen sollten wenig «Wissen» enthalten und sind in <boundary> -Klassen zu finden. Es geht also darum die Klassen in entsprechende Units zu verteilen. In unserem Businessbeispiel wurde folgende Aufteilung in 4 Units (Module oder Packages) vorgenommen:

- TDataModule1 und TDataToXML in der Unit bankData
  (enthält den möglichen Zugriff auf InterBase)

- TBusinessObj und TTransactionClass in der Unit bankLogic
  (Verarbeitung und Business Rules)

- TForm1 und TfrmTransMonitor in der Unit bankView
  (die Darstellung der Dialoge und Forms)

- controllAction in der Unit bankControll
  (vermittelt zwischen dem View und der Logik)

1.2.3 Die Klassen mit Sichtbarkeiten und Zugriffsmethoden definieren

Alle Teile von Klassen, einschliesslich der Felder, Methoden und Eigenschaften befinden sich auf einer bestimmten Schutz- oder Sichtbarkeitsebene. Inwieweit der Zugriff auf die Attribute und Methoden erlaubt ist, können Sie an der Kennzeichnung (ab UML 1.1) erkennen:

public:	ein « + » vorangestellt
protected	ein « # » vorangestellt
private:	ein « - » vorangestellt.

Meist werden Sie Methoden in Klassen oder Komponenten als `public` oder `protected` deklarieren. Nur selten ist das Sichtbarkeitsattribut private nötig, das bewirkt, dass nicht einmal abgeleitete Klassen Zugriff erhalten. Deklarieren Sie Elemente als `private`, wenn sie nur in der Klasse verfügbar sein sollen, in der sie auch definiert werden. Deklarieren Sie Elemente als protected, wenn sie nur in dieser Klasse und in ihren Nachkommen verfügbar sein sollen. Bedenken Sie aber, dass ein Element, das an irgendeiner Position innerhalb einer Unit verfügbar ist, in der gesamten Datei zur Verfügung steht. Wenn Sie also zwei Klassen in derselben Unit definieren, können diese auf die als private deklarierten Methoden der jeweils anderen Klasse zugreifen. Man spricht von einer friend-Beziehung.

Zusätzlich lassen sich Properties mit get- und set-Methoden in der Klasse definieren. Somit lassen sich in den read- und write-Abschnitten einer Eigenschaftsdeklaration anstelle eines Feldes die Zugriffsmethoden festlegen.

Angenommen Sie haben soeben eine Eigenschaft `KreditLimit` eingebaut. Die Anforderung wird nun dahingehend erhöht, dass jede Änderung der Kreditlimite eine Autorisierung mit einem Passwort erfordert. Mit einem normalen Feld oder einer mehrfach vorhanden Funktion wären Sie gezwungen, an allen Stellen im System eine entsprechende Validierung einzufügen. Nicht so mit einer Eigenschafts-Kapselung, die einfach und sicher zugleich ist:

```
Protected
  function getKreditLimit: double;
  procedure setKreditLimit(newLimit: double);
public
  property KreditLimit: double
    read getKreditLimit write setKreditLimit;
```

Hier ist wieder das eigentlich Feld `FKreditLimit` geschützt im privaten Bereich, jeder Zugriff muss also zwangsläufig über die Eigenschaft `KreditLimit` erfolgen, die bei Lesen oder Schreiben die entsprechende Methode z.B. `setKreditLimit` feuert:

```
procedure TKontoClass.setKreditLimit(newLimit: double);
begin
  if FKreditLimit <> newLimit then begin
    if getPassword(frmTrans.password) then begin
      FKreditLimit:= newLimit
      end else MessageDlg(frmTrans.Label2.Caption,mtWarning,[mbok],0);
    end;
end;
```

Zunächst wird beim versuchten Schreiben einer neuen Kreditlimite ein Passwortdialog aufgerufen (der übrigens von einer DLL stammt), der bei erfolgreicher Autorisierung das private Feld `FKreditLimit` ändert. Ein direkter Zugriff auf das Feld ist somit nicht mehr möglich und die Methode ist künftig erweiterbar z.B. mit einer Datumskontrolle oder einer Limitenberechnung. Auch dient diese Technik als Bedingung zum Komponentenbau, die alle Voraussetzungen eines echten OO-Objekts erfüllen sollten:

- Sie kapseln Daten und Zugriffsfunktionen (Kapselung).
- Sie erben Daten und Methoden von den Objekten, von denen sie abgeleitet werden (Vererbung).
- Sie arbeiten austauschbar mit anderen Objekten zusammen, die von einem gemeinsamen Vorfahr abgeleitet sind (Polymorphie und Typenverträglichkeit zur Laufzeit).

1.2.4 3. Beziehungen zwischen Klassen festlegen

Weiter lässt sich die Kommunikation zwischen den Klassen wie Vererbung (Polymorphie), Aggregation und Assoziation erstellen und <relationships> zu Klassen einbinden. Gedanklich sind diese Beziehungen meistens schon bei der Klassenfindung erarbeitet worden, werden aber in dieser Phase stets neu hinterfragt und schrittweise implementiert. Denn während das statische Modell die Beziehungen abbildet, in denen Klassen zueinander stehen, beschreibt das spätere dynamische Modell, wie diese Beziehungen benutzt werden, d.h., welche Botschaften entlang diesen Beziehungen fliessen.

Eine Methode umfasst nur in den seltensten Fällen die Ausführung einer einzigen Funktion. Wesentlich häufiger sind verkettete Aufrufe mehrerer Methoden und Kaskadierungen möglich, die aber nicht in derselben Klasse angesiedelt zu sein brauchen. Somit wird klar, dass diese Phase eben stark iterativ geprägt ist und einer wiederholten Selbstprüfung nichts im Wege stehen sollte.

Ich hoffe, mit diesem Workshop Sie ermutigt zu haben, mehr Klassen mit Klasse zu bauen.

1.3 Auswahlkriterien

Die erwähnten Web-Sites (siehe folgender Link) bieten fast ausnahmslos eine einsatzfähige Demo-Version von CASE-Tools an. Zudem befindet sich auf der CD im Heft noch einiges an verwertbaren Tools und Demos. Für Fragen oder Bemerkungen bin ich gerne unter max@kleiner.com erreichbar. Aktuelle Tools finden sie unter: http://max.kleiner.com/umltools.htm.

Bei der Beschaffung eines Tools ist es nützlich, sich vor allem die Plattform und die unterstützen Sprachen näher anzuschauen. Auch die Frage der Teamarbeit muss bei der Auswahl einen Einfluss haben. Mit diversen Tools können Sie über einen einzigen Ressourcenpool oder eine Repository die Modellinformationen in einem Team gemeinsam nutzen. Um einen gemeinsamen Ressourcenpool erstellen zu können, müssen die Projektdateien in irgendeiner Form verknüpft oder zusammengeführt werden.

Auch die Frage nach zusätzlichen Add-On's von Drittanbietern muss man sich stellen Weitere Überlegungen führen dann zu folgenden Kriterien, die es zu bewerten gilt:

Ergonomie / Dialogfluss	Repository / Modelldatenbank
Schnittstellen, Import / Export	Geschäftsprozesse, Workflow
Unterstützte Plattformen	Online Hilfe / Dokumentation
Welche Diagrammtypen	Vorgehensmodell / Prozess
Teamdevelopment	Codegenerierung, Design Patterns
Skripting / Makrosprache	Dokumentengenerator
ERD-Support	Reverse Engineering

Extrakt aus dem Buch
„Patterns konkret“