Objektorientierte Softwareentwicklung mit Oracle PL/SQL

Thomas Hoekstra
BNS Software AG
Meerbusch

Schlüsselworte

Objektorientierung, PL/SQL, 4 Schicht Architektur, Performance

Zusammenfassung

Ziel dieses Vortrages ist es, die objektorientierten Features von PL/SQL vorzustellen und mit anderen objektorientierten Sprachen zu vergleichen. Außerdem soll anhand eines Beispieles gezeigt werden, wie die Verwendung dieser Features neue Möglichkeiten der Software Architektur eröffnet und welchen Einfluss diese Architektur auf die Stabilität und Performance der Applikation hat.

Einleitung und Motivation

PL/SQL (Procedural Language / Structured Query Language) – schon der Name dieser Sprache legt nahe, dass es sich hier um ein Relikt aus den Zeiten prozeduraler Entwicklung handelt. Außerdem gibt es ja noch Java, wenn man mit Oracle objektorientiert entwickeln möchte. Trotzen habe ich mich in einem meiner letzten Projekte, einem Lagerverwaltungssystem, für PL/SQL und gegen Java entschieden.

Mit der Version Oracle 8i (8.1.5) haben objektorientierte Features Einzug in PL/SQL gehalten und seit Oracle 9i R2 (9.2.0.1) betrachte ich PL/SQL als vollwertige objektorientierte Sprache.

Das bereits erwähnte Lagerverwaltungssystem sollte als 3-Schicht Anwendung unter Verwendung von .Net Remoting und mit Oracle 9i R2 als Datenbanksystem realisiert werden. Es entstand die Idee, die Business Objekte in PL/SQL zu implementieren, die PL/SQL Objekte aus .Net Klassen zu instanzieren und ihre Methoden und Attribute über die .Net Klassen den Clients zugänglich zu machen.

Objektorientierte Features von PL/SQL

1. Klassen (Object Types)

PL/SQL bietet die Möglichkeit, Klassen zu erstellen, die mehrere Datenelemente und die mit diesen Datenelementen möglichen Operationen zusammenfassen. Dabei sind, im Gegensatz zu Sprachen wie Java oder C++, alle Datenelemente und Methoden öffentlich. Eine PL/SQL Klasse besteht aus zwei Teilen: der Spezifikation (TYPE) und der Implementierung (TYPE BODY). Alle Datenelemente müssen in der Spezifikation deklariert werden. Es ist nicht erforderlich, dass eine PL/SQL Klasse Methoden hat. Wenn Sie aber keine Methoden, also auch keinen Konstruktor (siehe 1.4.) implementieren, fügt Oracle einen Standard Konstruktor hinzu.

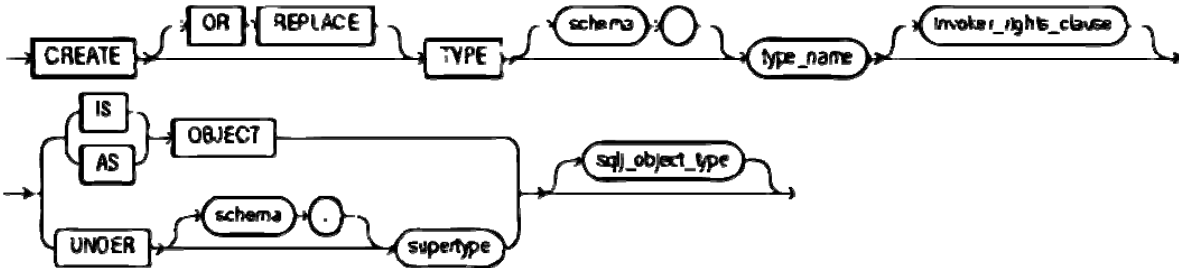

Abb. 1: Syntaxdiagramm CREATE TYPE

```
CREATE TYPE PUNKT AS OBJECT
(
       X       NUMBER,
       Y       NUMBER
)
```

Beispiel 1: Einfacher Object Type ohne Methoden

1.1. Datenelemente

Datenelemente von Objekten können alle eingebauten Oracle Datentypen und alle Benutzerdefinierten Objekttypen sein, auf denen der Besitzer des Objektes Ausführungsrechte besitzt. Nicht erlaubt dagegen sind Benutzerdefinierte Datentypen wie sie in PL/SQL Packages definiert werden können, also z.B. TYPE KundenNummer is NUMBER(10,0).

1.2. Member Methoden und Funktionen

Member Methoden können die Daten einer Instanz verändern oder mit ihnen eine Berechnung durchführen. Dabei bezeichnet das Schlüsselwort SELF die Instanz, zu der die Methode gehört.

```
CREATE TYPE PUNKT AS OBJECT
(
   X       NUMBER,
   Y       NUMBER,
   MEMBER FUNCTION Abstand(P PUNKT) RETURN NUMBER
)
/
```

```
CREATE TYPE BODY PUNKT
AS
   MEMBER FUNCTION Abstand(P PUNKT) RETURN NUMBER
   IS
   BEGIN
       RETURN
           SQRT(POWER(P.X - SELF.X,2) +
               POWER(P.Y - SELF.Y,2));
   END Abstand;
END;
/
```

Beispiel 2: Object Type und Type Body mit Member Methoden

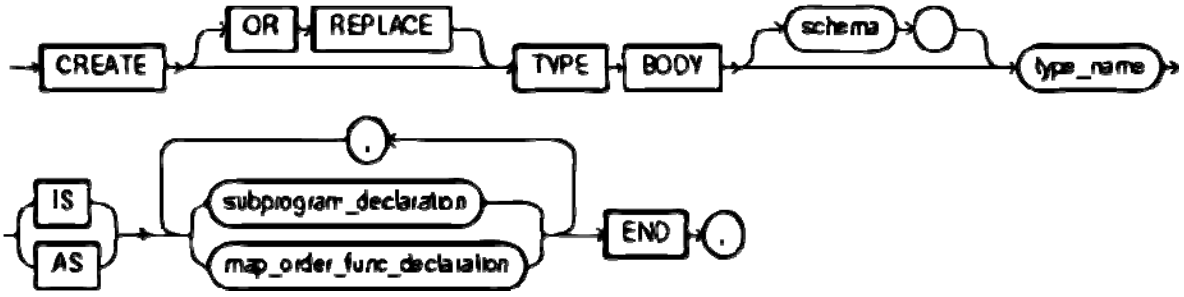

Abb. 2: Syntaxdiagramm CREATE TYPE BODY

1.3. Statische Methoden

Statische Methoden gehören zur Klasse, nicht zu einer einzelnen Instanz. Sie stellen Funktionen zur Verfügung, die sich nicht eindeutig einem einzelnen Objekt zuordnen lassen, oder für deren Ausführung keine Objektinstanz benötigt wird.

Der Aufruf statischer Funktionen erfolgt über den vorangestellten Klassennamen, nicht über den Namen einer Objektvariablen, im folgenden Beispiel also `Punkt.NullPunkt()`.

```
CREATE TYPE PUNKT AS OBJECT
(
   X       NUMBER,
   Y       NUMBER,
   MEMBER FUNCTION Abstand(P PUNKT) RETURN NUMBER,
   STATIC FUNCTION NullPunkt RETURN PUNKT
       /*Diese Funktion gibt den Nullpunkt des
       Koordinatensystems zurück.*/
)
/
CREATE TYPE BODY PUNKT
AS
   (…)
   STATIC FUNCTION NullPunkt RETURN PUNKT
   IS
   BEGIN
       RETURN PUNKT(0,0);
```

```
   END NullPunkt;
END;
/
```

Beispiel 3: Eine statische Funktion

1.4. Konstruktoren und Destruktoren

Konstruktoren sind Funktionen, die ein Objekt initialisieren. Sie werden im Augenblick der Objekterstellung durchlaufen. Eine Klasse kann beliebig viele Konstruktoren haben (siehe 1.5 Überladung (Overloading)). Leider ist PL/SQL die einzige mir bekannte objektorientierte Sprache, die keine Destruktoren unterstützt.

1.5. Überladung (Overloading)

PL/SQL unterstütz das Konzept der Überladung, d. h. eine Klasse kann mehrere Methoden mit dem gleichen Namen haben, die sich nur in ihrer Signatur, also der Anzahl und den Datentypen ihrer Argumente, unterscheiden.

```
CREATE TYPE KREIS AS OBJECT
(
  Mittelpunkt Punkt,
  Radius      NUMBER,
  Constructor FUNCTION Kreis RETURN SELF AS result
  Constructor FUNCTION Kreis(X,Y,R) RETURN SELF AS result
)
/

CREATE TYPE BODY KREIS
AS
   Constructor FUNCTION Kreis RETURN SELF AS result
   IS
    BEGIN
      SELF.Radius := 1;
      SELF.MittelPunkt := Punkt.NullPunkt();
      RETURN;
    END Kreis;

   Constructor FUNCTION Kreis(X,Y,R) RETURN SELF AS result
   IS
    BEGIN
      SELF.Radius := R;
      SELF.MittelPunkt := Punkt(X,Y);
      RETURN;
    END Kreis;

END
/
```

Beispiel 4: Benutzerdefinierte Konstruktoren

1.6. Operatoren

PL/SQL unterstützt zwei Arten benutzerdefinierter Operatoren. Beide dienen dazu, Objekte miteinander zu vergleichen bzw. sie zu sortieren. Welche der beiden Arten Sie in einer Klasse verwenden, hängt davon ab, ob es eine einfache Möglichkeit gibt, den Objekttyp in einen skalaren Datentyp umzuwandeln, oder ob es erforderlich ist, zwei Objekte zu betrachten, wenn man entscheiden will welches „größer“ ist als das andere. Wenn Sie die Wahl haben, sollten Sie eich immer für die MAP Funktionen entscheiden, da hier für jedes Objekt nur ein Funktionsaufruf erforderlich ist. Anschließend können die skalaren Werte miteinander verglichen werden. Wenn Sie keine Methode zum Vergleich implementieren, können Sie Objekte nur auf Gleichheit (d.h. alle Attribute stimmen überein) oder Ungleichheit überprüfen.

1.6.1. Skalare Typumwandlung (Map functions)

MAP Funktionen dienen dem Vergleich von Objekten, indem sie Objekttypen in skalare, also vergleichbare, Datentypen umwandeln. Ein Objekttyp kann höchstens eine MAP Funktion für einen der Datentypen DATE, NUMBER, VARCHAR2 oder die ANSI SQL Typen wie CHARACTER oder REAL haben.

Eine Möglichkeit, geometrische Figuren wie Kreise und Rechtecke miteinander zu vergleichen könnte die Betrachtung ihres Flächeninhaltes sein. Das folgende Beispiel implementiert eine MAP Funktion für den Datentyp NUMBER in der Klasse KREIS.

```
CREATE TYPE KREIS AS OBJECT
(
  Mittelpunkt Punkt,
  Radius      NUMBER,
  MAP MEMBER FUNCTION Flaeche RETURN NUMBER,
  Constructor FUNCTION Kreis RETURN SELF AS result
)
/
CREATE TYPE BODY KREIS
AS
   (…)
   MAP MEMBER FUNCTION Flaeche RETURN NUMBER
    IS
    BEGIN
         RETURN 4*atan(1) * power(SELF.Radius,2);
    END;
END;
/
```

Beispiel 5: Eine MAP Funktion

```
declare
  k1 kreis;
  k2 kreis;
begin
   k1 := Kreis();             /*Erzeugt einen Einheitskreis*/
   k2 := Kreis(punkt.nullpunkt(),2);   /*Kreis mit Radius 2*/
```

```
  IF k1 > k2 THEN
     dbms_output.put_line('k1 ist größer.');
  ELSE
     dbms_output.put_line('k2 ist größer.');
  END IF;
end;
/
>> k2 ist größer.
```

Beispiel 6: Verwendung einer MAP Funktion

1.6.2. Vergleichsoperatoren (Order functions)

Wenn es nicht möglich ist, einem Objekt einen skalaren Wert zuzuweisen, sondern die Frage der „Größe“ nur in einer Einzelfallentscheidung geklärt werden kann, dann **und nur dann** sollten Sie ORDER Funktionen verwenden!

```
CREATE TYPE KREIS AS OBJECT
(
  Mittelpunkt Punkt,
  Radius      NUMBER,
  MAP MEMBER FUNCTION Flaeche RETURN NUMBER,
  Constructor FUNCTION Kreis RETURN SELF AS result
)
/
CREATE TYPE BODY KREIS
AS
   (…)
   ORDER MEMBER FUNCTION Vergleich(EinKreis KREIS)
   RETURN NUMBER
    IS
    BEGIN
         IF SELF.radius > EinKreis.radius THEN
            RETURN 1;
         ELSIF SELF.radius < EinKreis.radius THEN
            RETURN -1;
         ELSE
            RETURN 0;
         END IF;
    END;
END;
/
```

Beispiel 7: Eine ORDER Funktion

2. Vererbung

Vererbung ist das wohl wichtigste Feature objektorientierter Sprachen. Vererbung ermöglicht die Bildung von Klassenhierarchien, in denen weiter unten stehende Klassen die Methoden und Attribute (oder das Verhalten und die Eigenschaften) ihrer Vaterklassen übernehmen und diese funktional erweitern. Je weiter unten eine Klasse in einer Klassenhierarchie steht, desto spezieller ist sie, je weiter oben sie steht, desto allgemeiner ist sie. Wie Java erlaubt auch

PL/SQL nur eine direkte Vaterklasse. Die Mehrfachvererbung, wie sie in C++ üblich ist, wird (leider) nicht unterstützt.

2.1. Vergleichsoperatoren in einer Klassenhierarchie.

Für Vergleichsoperatoren innerhalb einer Klassenhierarchie gibt es einige Einschränkungen, die ich so auch nur aus PL/SQL kenne und daher besonders erwähnen möchte:

Zum einen ist es nur erlaubt, in einer Klasse eine MAP Funktion zu implementieren, wenn auch ihre Vaterklasse eine MAP Funktion beinhaltet, oder wenn sie ganz oben in der Klassenhierarchie steht.

Mit ORDER Funktionen verhält es sich anders: sie werden zwar vererbt, dürfen aber nur in einer Klasse implementiert werden, die selbst nicht von einer anderen Klasse abgeleitet ist. Ein weiterer Gr#und, soweit möglich auf den Einsatz von ORDER Funktionen zu verzichten.

2.2. Rein virtuelle Klassen und Funktionen

Eine Funktion heißt rein virtuell, wenn sie nur deklariert, aber nicht implementiert wurde. Eine solche Funktion wird als NOT INSTANTIABLE deklariert. Eine Klasse, die solche Funktionen oder Methoden enthält, muss selbst ebenfalls als NOT INSTANTIABLE, also nicht instanzierbar, deklariert werden. Sie kann also nur als Vaterklasse für andere, spezialisierte Klassen dienen.

Das folgende Beispiel zeigt die Klasse COraObject, die ich in meinen Applikationen als Vaterklasse aller Oracle Objekttypen verwende.

```
CREATE OR REPLACE TYPE COraObject AS OBJECT
(
     -- This is the base class for all classes.
     ID         NUMBER,
     NOT FINAL STATIC FUNCTION Persist
               (
                   TheObject IN OUT NOCOPY COraObject,
                   CurrentUser IN NUMBER,
                   ErrMessage OUT VARCHAR2
               )
               RETURN NUMBER,
     NOT FINAL STATIC FUNCTION Erase
               (
                   TheObject IN OUT NOCOPY COraObject,
                   CurrentUser IN NUMBER,
                   ErrMessage OUT VARCHAR2
               )
               RETURN NUMBER,
)
NOT FINAL
NOT INSTANTIABLE
```

Beispiel 8: Eine rein virtuelle Klasse

Diese Klasse legt fest, dass jedes Objekt eine ID hat, die es innerhalb seiner Klasse eindeutig kennzeichnet und mit deren Hilfe ich es in einer Tabelle oder einem *Object View* finden kann. Außerdem bekommt jede Klasse zwei statische Methoden: Persist und Erase. Das Persistieren und Löschen von Objekten ist also ein Service der Klasse. Die Methoden – ebenso wie die Klasse COraObject selbst – sind virtuell. Da sie statisch sind, also nicht zu einer Instanz gehören, müssen sie nicht als NOT INSTANTIABLE deklariert werden.

2.3. FINAL und NOT FINAL

Das Schlüsselwort FINAL gibt an, dass eine Methode oder auch eine ganze Klasse endgültig ist. Damit ist es nicht möglich, diese Funktion in einer abgeleiteten Klasse zu übersteuern. Von endgültigen Klassen kann man keine weiteren Klassen ableiten.

```
CREATE OR REPLACE TYPE Final_Class
UNDER COraObject
(
)
FINAL
/

CREATE OR REPLACE TYPE Test_Class
UNDER Final_Class
(
)
/

SHOW ERRORS;
Errors for TYPE TEST_CLASS:

LINE/COL ERROR
-------- -----------------------------------------------------
1/1      PLS-00590: Versuch, einen Subtyp UNTER einem FINAL-
         Typ zu erstellen
```

Beispiel 9: Eine Finale Klasse

2.4. Übersteuerung (Overriding)

Soll das Verhalten einer Funktion, die aus einer Vaterklasse geerbt wurde, in einer abgeleiteten Klasse übersteuert werden, dann muss diese mit dem Schlüsselwort OVERRIDING deklariert werden.

```
CREATE OR REPLACE TYPE Not_Final_Class
UNDER COraObject
(
  FINAL MEMBER PROCEDURE member1,
  NOT FINAL MEMBER PROCEDURE member2
)
NOT FINAL
/
```

```
CREATE OR REPLACE TYPE Test_Class
UNDER Not_Final_Class
(
  OVERRIDING MEMBER PROCEDURE member1
)
/
SHOW ERRORS;
Errors for TYPE TEST_CLASS:

LINE/COL ERROR
-------- -----------------------------------------------------
4/21     PLS-00637: Methode FINAL kann nicht außer Kraft
         gesetzt oder verborgen werden

CREATE OR REPLACE TYPE Test_Class
UNDER Not_Final_Class
(
  MEMBER PROCEDURE member2
)
/
SHOW ERRORS;
Errors for TYPE TEST_CLASS:

LINE/COL ERROR
-------- -----------------------------------------------------
4/10     PLS-00636: Neu angegebene Methode erfordert
         Schlüsselwort OVERRIDING

CREATE OR REPLACE TYPE Test_Class
UNDER Not_Final_Class
(
  OVERRIDING MEMBER PROCEDURE member2
)
/
Type created
```

Beispiel 10: Übersteuerung (Overloading); Außerkraftsetzen des Verhaltens der Vaterklasse

2.5. Typumwandlung (Type Cast /TREAT)

Mit der Funktion Treat können Sie verschiedene Objekt-Typen ineinander umwandeln. Voraussetzung dafür ist, dass sie zur gleichen Objekthierarchie gehören. Mit anderen Worten, `expr` muss ein Sub- oder Supertyp von `type` sein. Wenn Sie versuchen, ein Objekt in einen Objekttyp umzuwandeln, der in der Klassenhierarchie weiter untern steht (also spezieller ist) als der am meisten spezialisierte Typ der auf das Objekt anwendbar ist, dann gibt TREAT NULL zurück. Dasselbe gilt natürlich für Referenzen (REF) auf Objekte.

Abb. 3: Syntaxdiagramm TREAT

2.6. Interfaces

Als Inerfaces bezeichnet man im Allgemeinen rein virtuelle Klassen, die nicht instanzierbar sind und nur als gemeinsame Schnittstelle zu allen von ihnen abgeleiteten Klassen dienen.

Die unter 2.2 (Rein virtuelle Klassen und Funktionen) vorgestellte Klasse COraObejct ist ein solches Interface. Da PL/SQL keine Mehrfachvererbung unterstützt, ist es auch nicht möglich, dass eine Klasse mehrere solcher Interfaces implementiert. Das Interface Konzept ist in PL/SQL daher nicht in dem Maße anwendbar, wie beispielsweise in C++ oder in Sprachen wie C#, Java, VB.Net etc., die zwar ebenfalls keine Mehrfachvererbung unterstützen, dafür aber Interfaces.

Object-Views, eine objektorientierte Sichtweise für relationale Daten

Eine interessante Alternative zum Erstellen von Objekt-Tabellen, ist die Definition von Object-Views. Sie erlauben es, eine vorhandene Anwendung mit relationalen Tabellen und strukturiertem (Prozeduralem) Code schrittweise in eine objektorientierte Anwendung zu migrieren. Die Object-Views dienen dabei als eine Art Classfactory, sie erstellen auf Anfrage Instanzen der Objekt Typen aus den relationalen Daten.

3. Objekt Views definieren

Object-Views werden mit der CREATE VIEW Anweisung erstellt. Dabei wird zwischen `CREAE VIEW` … und `AS SELECT` … eine Object-View-Clause eingefügt:

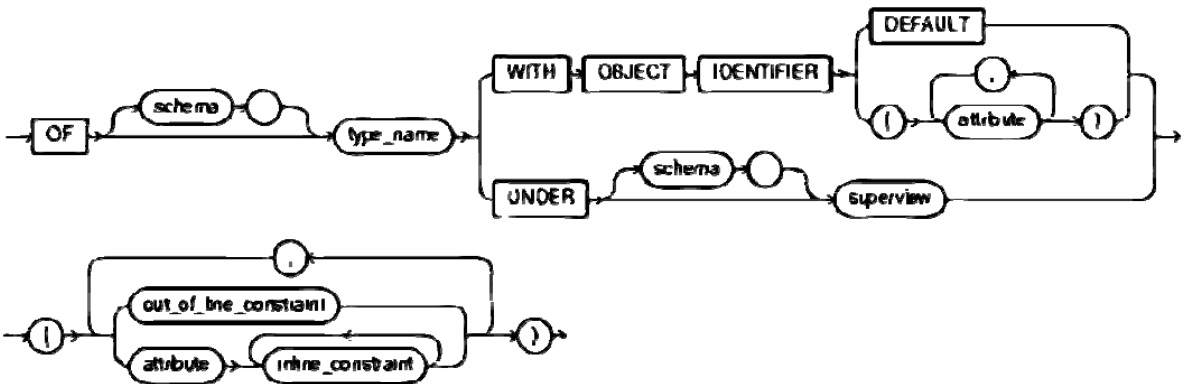

Abb. 4: OBJECT VIEW CLAUSE

Das folgende Beispiel erzeugt ein View, das Objekte der in Beispiel 4 definierten Klasse KREIS enthält.

```
CREATE TABLE KREISE
(
  X NUMBER not null,
  Y NUMBER not null,
  R NUMBER not null
);
```

```
CREATE OR REPLACE VIEW
VW_KREISE
OF KREIS
WITH OBJECT OID (MITTELPUNKT.X, MITTELPUNKT.Y, RADIUS)
AS SELECT
PUNKT(X,Y), R
FROM KREISE
```

Beispiel 11: Ein einfaches Object-View

4. Vererbung in Object-Views: View Hierarchien

Stellen Sie sich vor, Sie schreiben eine Anwendung, die mit geometrischen Figuren arbeitet. In Ihrer Anwendung gibt es Punkte, Polygone, Kreise, Ellipsen usw. Die Daten für diese Figuren liegen in relationalen Tabellen vor, sie verwenden aber Objekttypen um mit den Daten zu arbeiten. Sie könnten jetzt mehrere Object-Views erstellen, für jede Figurenklasse eines. Praktischer ist es aber, wenn Sie nur ein View haben, in dem sie nach verschiedenen Figuren anhand von gemeinsamen Attributen suchen können ohne sich Gedanken darum machen zu müssen, von welchem Typ die Objekte sind. Um eine solche View Hierarchie zu erstellen, legen sie zuerst das View für die Basisklasse an, und dann ein zweites View für die Spezialisierte Klasse.

```
create table TB_FAHRZEUGE
(
  ID          NUMBER not null,
  BESITZER    VARCHAR2(100),
  HERSTELLER  VARCHAR2(100),
  KENNZEICHEN VARCHAR2(30),
  TYP         CHAR(1)
)
;
create or replace type Fahrzeug
UNDER COraObject
(
  Besitzer VARCHAR2(100),
  Hersteller VARCHAR2(100),

  -- Member functions and procedures
  STATIC FUNCTION Persist(…) RETURN NUMBER,
  STATIC FUNCTION Erase(…) RETURN NUMBER
)
NOT FINAL
;
create or replace type Fahrrad
UNDER Fahrzeug
(
  STATIC FUNCTION Persist(…) RETURN NUMBER,
  STATIC FUNCTION Erase(…) RETURN NUMBER
)
```

```
NOT FINAL
;
create or replace type PKW
UNDER Fahrzeug
(
  KENNZEICHEN VARCHAR2(30),
  STATIC FUNCTION Persist(…) RETURN NUMBER,
  STATIC FUNCTION Erase(…) RETURN NUMBER
)
NOT FINAL
;
```

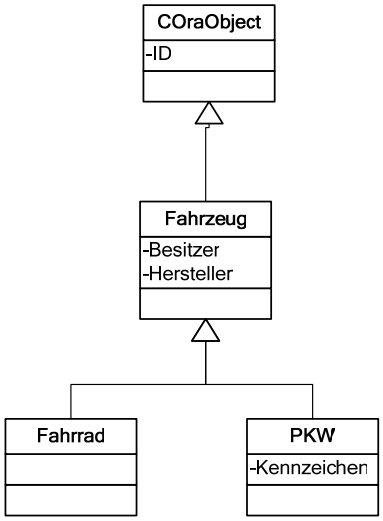

Abb. 5: Eine Klassenhierarchie, die in einer View Hierarchie abgebildet werden kann

Um diese einfache Klassenhierarchie in Object-Views abzubilden, wird zunächst ein View für die Basisklasse benötigt:

```
CREATE OR REPLACE VIEW Fahrzeuge
OF Fahrzeug WITH OBJECT OID(ID)
AS
SELECT Fahrzeug(ID, BESITZER, Hersteller)
FROM tb_fahrzeuge
WHERE typ IS NULL
```

Beispiel 12: Die „Wurzel“ eine View-Hierarchie

Unter diesem View können jetzt die Views für die spezialisierten klassen angelegt werden:

```
CREATE OR REPLACE VIEW
PKWs
OF PKW
UNDER
FAHRZEUGE
AS SELECT PKW(id, besitzer, hersteller, Kennzeichen)
FROM tb_fahrzeuge WHERE typ = 'P'
/
```

```
CREATE OR REPLACE VIEW FAHRRAEDER OF FAHRRAD
UNDER FAHRZEUGE
AS
SELECT FAHRRAD(id, besitzer, hersteller)
FROM tb_fahrzeuge WHERE typ = 'F'
/
```

Beispiel 13: Zwei spezialisierte Views in einer View-Hierarchie

Anwendung objektorientierter Features von PL/SQL

In dem eingangs bereits erwähnten Projekt sollte eine 3-Schicht Architektur, mit .Net-Remoting und Oracle als Datenbankserver, verwendet werden. Mit den neuen Features von PL/SQL haben wir jetzt die Möglichkeit, einen großen Teil der Geschäftslogik dort zu implementieren, wo sich die Daten befinden: in der Datenbank. Daraus ergeben sich einige Vorteile gegenüber der sonst üblichen Verfahrensweise. Zum einen wird der Datenverkehr zwischen Datenbank und Mittelschicht auf das notwendige Minimum, nämlich die Ergebnisse, reduziert, zum anderen wird so die Komponente, die sich bei steigenden Benutzerzahlen am besten skalieren lässt auch am stärksten belastet: Eine in .Net entwickelte Mittelschicht ist nur auf einem Windows System lauffähig und daher nach wie vor auf Hardware beschränkt, die den von Oracle unterstützten UNIX Plattformen unterlegen ist.

Die Mittelschicht ist nur noch für den Teil der Geschäftslogik verantwortlich, der sich aus unterschiedlichen Gründen nicht in PL/SQL implementieren lässt. Es kommt also zu einer Zweiteilung der Mittelschicht: Basisklassen in PL/SQL und spezialisierte Klassen in .Net. Daher bezeichne ich diese Architektur als 4-Schicht Architektur.

5. Softwarestack einer 4–Schicht Anwendung

3-Schicht Anwendungen bestehen aus den folgenden Schichten:

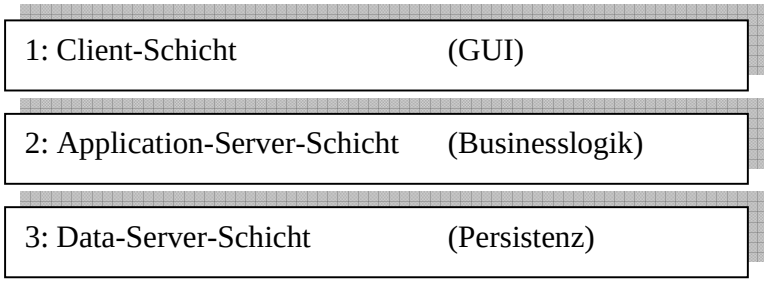

Abb. 6: 3-Schicht Architektur

In einer 4-Schicht Anwendung wir ein Teil der Businesslogik in die Datenbank verlagert und dort in PL/SQL Klassen implementiert:

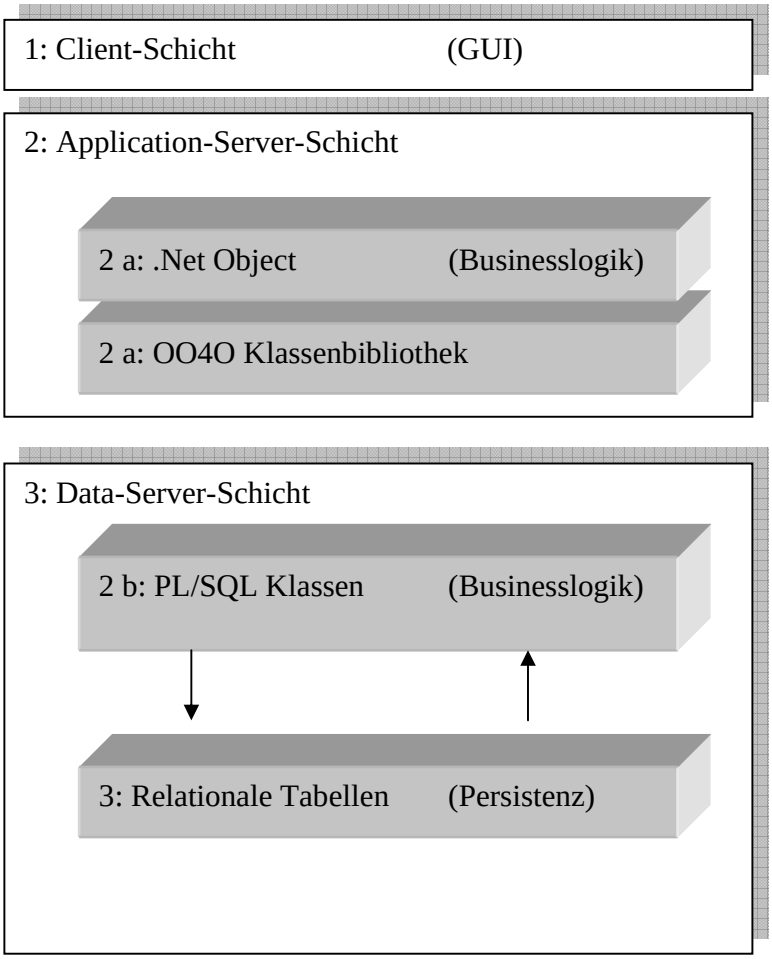

Abb. 7: 4-Schicht Architektur

Die Interaktion zwischen Client, Mittelschicht und Datenbank verläuft dann so, wie in Abb. 8 dargestellt.

Die Performance einer so aufgebauten Anwendung hängt im wesentlichen davon ab, wie die SQL Anweisungen aufgebaut werden, die aus den PL/SQL Klassen heraus die Daten in den relationalen Tabellen manipulieren. Der Overhead für die Objekterstellung ist auch bei großen Datenmengen nicht spürbar, da auch bei Speicherung der Objekte in Objekt Tabellen vor dem Zugriff auf ein Objekt aus den serialisierten Daten ein PL/SQL Objekt erstellt werden muss.

Abb. 8 zeigt auch, dass vergleichsweise wenig SQL ausgeführt wird, die Client Anwendung kommt sogar ganz ohne SQL und ohne Oracle Client aus, da sie ausschließlich über .Net Remoting mit dem Server kommuniziert. Auch der Application Server enthält nur den erforderlichen SQL Code, um die Objekte aus der Datenbank zu holen und wieder zu speichern oder zu löschen.

Die Methode zum Initialisieren eines .Net Objektes aus der Datenbank wird hier in der .Net Klasse COraObject implementiert und an alle abgeleiteten Klassen vererbt. Diese Methode wird auch von den verschiedenen Konstruktoren von COraObject aufgerufen. COraDatabase kapselt dabei ein OraDatabase Objekt aus der OO4O Klassenbibliothek.

```
Public MustInherit Class COraObject
    Inherits System.MarshalByRefObject  ' all instances reside
                                         in the Middle Tier. They are not
                                         serialized and sent to the UI.
    Implements IDisposable
    Private m_ObjectSource As String    ' name of the object view
    Private m_ClassName As String       ' name of the object type

    Protected m_Obj As OracleInProcServer.OraObject
    Protected m_OraDB As COraDB = New COraDB
    Protected m_ErrorInfo As New ErrorInfo  ' errors raised by this object

    Protected m_CurrentUser As System.Int32
    Private m_Disposed As Boolean = False

    Public Sub New(ByVal currentUser As System.Int32, ByVal ID As
                System.Int32, ByVal className As String, ByVal
                objectSource As String, ByVal db As COraDatabase)
    Create(currentUser, ID, className, objectSource, db)
    End Sub

    Private Sub Create(ByVal currentUser As System.Int32, ByVal ID As
                       System.Int32, ByVal className As String, ByVal
                       objectSource As String, ByVal db As COraDatabase)
    m_ObjectSource = objectSource
    m_ClassName = className
    m_CurrentUser = currentUser
    SelectByID(ID, db, False)
    End Sub

    Protected Sub SelectByID(ByVal ID As System.Int32,
                         ByVal db As COraDatabase,
                       ByVal lockForUpdate As Boolean)
    db.Parameters.Add("ID", ID, OracleInProcServer.paramMode.ORAPARM_INPUT,
                       OracleInProcServer.serverType.ORATYPE_NUMBER)
    db.Parameters.Add("OBJ", Nothing,
                       OracleInProcServer.paramMode.ORAPARM_OUTPUT,
                       OracleInProcServer.serverType.ORATYPE_OBJECT,
                       ClassName)
    If lockForUpdate Then
        db.ExecuteSQL("BEGIN SELECT VALUE(o) INTO :OBJ FROM " &
                       ObjectSource & " o WHERE id = :ID FOR UPDATE;
                       END;")
    Else
        db.ExecuteSQL("BEGIN SELECT VALUE(o) INTO :OBJ FROM " &
                       ObjectSource & " o WHERE id = :ID; END;")
    End If
    m_Obj = db.Parameters("OBJ").Value
    db.Parameters.Remove("ID")
    db.Parameters.Remove("OBJ")
End Sub
…
…
End Class
```

Beispiel 14: Ein VB.Net Programm erzeugt PL/SQL Objekte

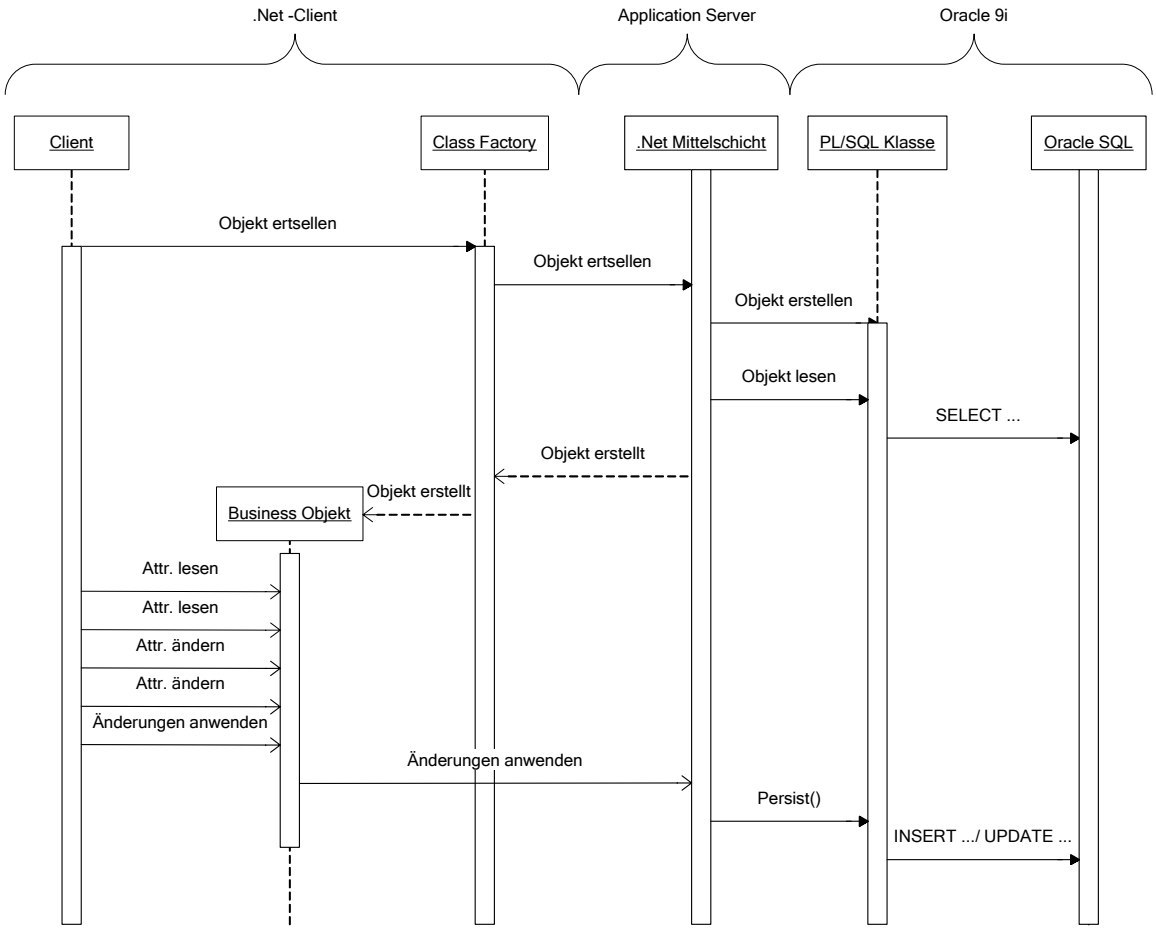

Abb. 8: Interaktion zwischen den Schichten einer 4-Schicht Anwendung

Auch die Persistenz aller Businessobjekte ist in einer Methode der Vaterklasse COraObject implementiert und ist für alle abgeleiteten Klassen gültig.

```
' Update the database with the contents of this object.
    ' <param name="db">A connection that is part of some
transaction.</param>
    ' <returns>Zero upon success, or a non-zero error code upon
failure.</returns>
Friend Overridable Function Persist(ByVal db As COraDatabase)As Int32
    Dim errorString As String
    Flush = -1  ' Assume failure
    Try
        ' Clear errors so that the client will not be confused.
        m_ErrorInfo.Clear()
        db.Parameters.Add("RESULT", Flush,
                          OracleInProcServer.paramMode.ORAPARM_OUTPUT,
                          OracleInProcServer.serverType.ORATYPE_NUMBER)
        db.Parameters.Add("OBJ", m_Obj,
                          OracleInProcServer.paramMode.ORAPARM_BOTH,
                          OracleInProcServer.serverType.ORATYPE_OBJECT,
                          m_Obj.TypeName)
```

```
            db.Parameters.Add("CURRUSER", m_CurrentUser,
                              OracleInProcServer.paramMode.ORAPARM_INPUT,
                              OracleInProcServer.serverType.ORATYPE_NUMBER)
            db.Parameters.Add("ERRSTR", errorString,
                              OracleInProcServer.paramMode.ORAPARM_OUTPUT,
                              OracleInProcServer.serverType.ORATYPE_VARCHAR2)
            db.ExecuteSQL("BEGIN :RESULT :=" & ClassName & ".Persist(:OBJ,
                              :CURRUSER, :ERRSTR); END;")
            If Not System.Convert.IsDBNull(db.Parameters("RESULT").Value) Then
                Flush = db.Parameters("RESULT").Value
            Else
                ' The function did not return properly and this means that an
                error has occurred.
                Flush = -1
            End If
            …
            …
            …
            m_Obj = db.Parameters("OBJ").Value
            Catch e As Exception
                …
                …
                Return -1
            Finally
                db.Parameters.Remove("ERRSTR")
                db.Parameters.Remove("CURRUSER")
                db.Parameters.Remove("OBJ")
                db.Parameters.Remove("RESULT")
        End Try
    End Function
    …
    …
    End Class
```

Beispiel 15: Ein VB.Net Programm speichert PL/SQL Objekte

Wie greift nun eine von COraObject abgeleitete .Net Klasse auf die Datenelemente des Oracle Objektes zu? Schauen wir einmal in das eingangs erwähnte Lagerverwaltungssystem. Es gibt dort eine Klasse CHandlingUnitBase, die die Eigenschaften eines Packstückes – Länge, Breite, Höhe, usw. – implementiert. Das folgende Code Beispiel zeigt den Beginn ihrer Implementierung mit den Konstruktoren und dem Zugriff auf das Attribut „Lengh“.

```
Public MustInherit Class CHandlingUnitBase
    Inherits COraObject
    Protected Const CLASS_NAME As String = "CHandlingUnit"
    Protected Const OBJ_SOURCE As String = "VWHANDLINGUNIT"

    Protected m_CultureName As String

    Friend Sub New(ByVal currentUser As System.Int32, ByVal cultureName As
                   String)
        MyBase.New(currentUser, CLASS_NAME, OBJ_SOURCE)
        m_CultureName = cultureName
        ChangeStatus_ = ChangeStatus.InsertOrUpdate
        SetMutableOraCollnFlags()
    End Sub
    Friend Sub New(ByVal ID As System.Int32, ByVal currentUser As
                   System.Int32, ByVal cultureName As String)
        MyBase.New(currentUser, ID, CLASS_NAME, OBJ_SOURCE)
```

```
        m_CultureName = cultureName
        SetMutableOraCollnFlags()
    End Sub
    Friend Sub New(ByVal ID As System.Int32, ByVal currentUser As
                   System.Int32, ByVal cultureName As String, ByVal db As
                   COraDatabase)
        MyBase.New(currentUser, ID, CLASS_NAME, OBJ_SOURCE, db)
        m_CultureName = cultureName
        SetMutableOraCollnFlags()
    End Sub
    Friend Sub New(ByVal objHandlingUnit As OracleInProcServer.OraObject,
                   ByVal currentUser As System.Int32, ByVal cultureName As
                   String)
        MyBase.New(currentUser, objHandlingUnit, CLASS_NAME, OBJ_SOURCE)
        m_CultureName = cultureName
        SetMutableOraCollnFlags()
    End Sub

    Public Property Length() As Double
        Get
            Return IIf(IsDBNull(m_Obj.Length), 0, m_Obj.Length)
        End Get
        Set(ByVal newValue As Double)
            m_Obj.Length = newValue
        End Set
    End Property…
…
End Class
```

Beispiel 16: Ein VB.Net Business-Objekt das auf einer PL/SQL Klasse basiert.

Fazit

Die Vorteile einer solchen Architektur bestehen im unter Anderem in der Vermeidung von SQL. Frei nach Steven Feuerstein: „The best way to write SQL is not to write SQL“(DOAG Konferenz 2001, PL/SQL - Best practices). Hierdurch verbessert sich die Performance der Datenbank, da weniger SQL geparst werden muss. Außerdem werden alle Operationen, bei denen viele Daten gelesen oder geschrieben werden müssen, unmittelbar in der Datenbank implementiert und so der Transport dieser Daten über das Netzwerk vermieden. Da der Datenbankserver einen großen Teil der Geschäftslogik ausführt, kann man auf wachsende Benutzerzahlen durch entsprechendes Skalieren des Datenbankservers reagieren (Oracle unterstützt Plattformen, auf denen eine .Net Mittelschicht in absehbarer Zukunft nicht lauffähig sein dürfte).

Der gemeinsame Zugriff aller Clients über eine Mittelschicht ermöglicht außerdem die Verwendung eines Datenbank-Verbindungs-Pools, ein Feature von OO4O für dessen Verwendung keine besondere Konfiguration der Datenbank (z.B. MTS) erforderlich ist.

Bei Fragen und Anregungen schreiben Sie mir einfach an eine der unten aufgeführten Email Adressen.

Ich freue mich bereits jetzt auf Ihre Erfahrungsberichte mit den neuen Features von PL/SQL!

Verzeichnis der Beispiele

Beispiel 1: Einfacher Object Type ohne Methoden....2
Beispiel 2: Object Type und Type Body mit Member Methoden....3
Beispiel 3: Eine statische Funktion....4
Beispiel 4: Benutzerdefinierte Konstruktoren....4
Beispiel 5: Eine MAP Funktion....5
Beispiel 6: Verwendung einer MAP Funktion....6
Beispiel 7: Eine ORDER Funktion....6
Beispiel 8: Eine rein virtuelle Klasse....7
Beispiel 9: Eine Finale Klasse....8
Beispiel 10: Übersteuerung (Overloading); Außerkraftsetzen des Verhaltens der Vaterklasse.9
Beispiel 11: Ein einfaches Object-View....11
Beispiel 12: Die „Wurzel“ eine View-Hierarchie....12
Beispiel 13: Zwei spezialisierte Views in einer View-Hierarchie....13
Beispiel 14: Ein VB.Net Programm erzeugt PL/SQL Objekte....15
Beispiel 15: Ein VB.Net Programm speichert PL/SQL Objekte....17
Beispiel 16: Ein VB.Net Business-Objekt das auf einer PL/SQL Klasse basiert.....18

Verzeichnis der Abbildungen

Abb. 1: Syntaxdiagramm CREATE TYPE....2
Abb. 2: Syntaxdiagramm CREATE TYPE BODY....3
Abb. 3: Syntaxdiagramm TREAT....10
Abb. 4: OBJECT VIEW CLAUSE....10
Abb. 5: Eine Klassenhierarchie, die in einer View Hierarchie abgebildet werden kann....12
Abb. 6: 3-Schicht Architektur....13
Abb. 7: 4-Schicht Architektur....14
Abb. 8: Interaktion zwischen den Schichten einer 4-Schicht Anwendung....16

Kontaktadresse:

Thomas Hoekstra

BNS Software AG
Meerbuscher Strasse 64-78
D-40670 Meerbusch

Telefon: +49(0)2159-5270
Fax: +49(0)2159-527527
E-Mail hoekstra@bnsag.de
hoekstra@gmx.net
Internet: www.bnsag.de