Object Types en Collections

Met de kracht ervan gaat een wereld open

Object Types en Collections bestaan in de Oracle Database al sinds versie 8.0. Maar voor zover ik kan bezien worden ze nog niet zo veel gebruikt. De meeste problemen zijn natuurlijk nog steeds op de Oracle 7 manier, zonder Object Types op te lossen. Echter, wanneer je bekend bent met Object Types en de kracht ervan dan gaat er een wereld voor je open. Soms kom je er ook achter dat code een stuk compacter kan wanneer je het oplost op een object georiënteerde manier.

Een van de commentaren die ik altijd heb gehad op de TCA (Trading Community Architecture)-API's van de Oracle E-Business Suite was dat ze te granulair waren. Als er bijvoorbeeld vanuit BPEL Process Manager een klant met bijbehorend adres moest worden opgevoerd, dan waren daar meerdere (ongeveer vier) API-aanroepen met de EBS of DB-adapter voor nodig om een Party met een Party Site op te voeren. In de huidige versie van E-Business Suite (R12) hebben de ontwikkelaars ook het voordeel van het gebruik van object types gevonden. Nu zijn er nl. Business Object BO-API's, gebaseerd op hiërarchieën van Object Types.
Deze types zijn standaard echter tamelijk sober. Ze bestaan uitsluitend uit een (vaak zeer grote) set van attributen en soms een static method functie. De attributen zijn vooral scalair, gebaseerd op simpele datatypen en soms complex, gebaseerd op andere types of Collections. De static method functie (create_object) fungeert als een constructor met parameters met een default null clausule, zodat ze niet allemaal gevuld hoeven te worden. Blijkbaar is er gekozen om geen extra constructor te definiëren.
Ik heb een aantal van deze types uitgebreid met extra functionaliteit. Omdat je in een Object Type de data van een instantie direct voorhanden hebt, is het toevoegen van functionaliteit in de Object Type vaak compacter dan wanneer je hetzelfde probeert te bereiken in een package. Omdat het standaardobjecten zijn, wilde ik ze niet direct aanpassen, maar via een eigen objecttype die de eigenschappen van de betreffende EBS-Objecten overerven.

Het toevoegen van functionaliteit op deze manier is niet zo moeilijk, maar ik liep wel tegen een paar uitdagingen aan. Ik laat zien hoe een Oracle Object Type wordt aangemaakt en via overerving uitgebreid kan worden. Daarnaast behandel ik de problematiek van type-casting.

Extending base types

Laten we eerst eens een parent Object Type als voorbeeld aanmaken:

```
create or replace type xxx_parent as object
(
-- Author  : MAKKER
-- Created : 17-03-2009 08:10:32
-- Purpose :
-- Attributes
  id number,
  name varchar2(30),
  description varchar2(100),
  -- Member functions and procedures
  constructor function xxx_parent return self as result
)
/
create or replace type body xxx_parent is
-- Version : $Id$
/* Member constructor, procedures and functions  */
  constructor function xxx_parent return self as result is
  begin
    return;
  end;
end;
/
```

Zoals je ziet, is het een simpel objecttype met een paar attributen en een parameterloze constructor. Ik ben gewend om altijd zo'n parameterloze constructor toe te voegen. Als je geen constructor opneemt, dan is er altijd een default constructor. Deze bevat alle attributen als parameter welke ook allemaal verplicht zijn. En als je een vrij uitgebreide type hebt met veel attributen die niet allemaal verplicht zijn dan is het vaak handig om een constructor te hebben zonder of eventueel met slechts enkele verplichte attribuut-parameters. Dan hoef je alleen de verplichte mee te geven. Later in de code kun je dan, wanneer nodig, de andere attributen vullen.

Ik wil dit Object Type uitbreiden door middel van een child of sub-type. Om dit te kunnen doen moet dit super-type 'not final' zijn. Wanneer het niet expliciet wordt aangegeven, worden Object Typen in Oracle echter impliciet als final gedeclareerd. Dit is om redenen van backward-compatibiliteit.
Nu kun je een type explicitiet als 'not final' declareren door het toevoegen van de clausule 'not final' aan de type specification. Bijvoorbeeld (vanuit de Oracle Documentatie):

```
CREATE TYPE person_typ AS OBJECT (
idno            NUMBER,
name            VARCHAR2(30),
phone           VARCHAR2(20),
MAP MEMBER FUNCTION get_idno RETURN NUMBER,
STATIC FUNCTION show_super (person_obj in person_typ) RETURN VARCHAR2,
MEMBER FUNCTION show RETURN VARCHAR2)
NOT FINAL;
/
```

Het gebruik van Not Final zou ik willen adviseren bij alle eigen standaard typen. Zo geef je dan andere ontwikkelaars net als in Java standaard de mogelijkheid jouw objecten te hergebruiken. Tenzij je expliciet wilt dat ze niet uitgebreid worden. Maar ook dat zou ik dan wel expliciet aangeven.
Maar in dit geval wil ik de EBS BO-types uitbreiden, zonder de broncode aan te passen. Een andere methode daarvoor is het aanpassen 'altering' van het type:

```
alter type xxx_parent not final;
```

Na uitvoeren van dit statement wordt dit statement toegevoegd als extra regel aan de source.
Probeer na het uitvoeren van het statement bijvoorbeeld eens:

```
select type, text from user_source where name = 'XXX_PARENT'
order by type, line
```

In elk geval kan er nu een child object/sub type worden gemaakt dat de vorige uitbreidt:

```
create or replace type xxx_child under xxx_parent
(
-- Author  : MAKKER
-- Created : 17-03-2009 08:12:43
-- Purpose :
  member procedure show,
-- Member functions and procedures
  constructor function xxx_child return self as result
)
/
create or replace type body xxx_child is
-- Version : $Id$
/* Member constructor, procedures and functions  */
  member procedure show is
  begin
    dbms_output.put_line('Id: ' || self.id || ', name: ' || self.name
||
                         ', description: ' || self.description);
  end;
  constructor function xxx_child return self as result is
  begin
    return;
  end;
end;
/
```

Dit type 'extends' de parent door toevoeging van een eenvoudige 'show' method, welke de attributen met dbms_output afdrukt. Op deze manier kun je dus standaard, meegeleverde Object Typen uitbreiden, zonder de bron-code daarvan aan te passen (afgezien van de 'Not Final' clausule).

Casting super-types to sub-types en vice versa

Het bovenstaande laat echter nog een Type Casting probleem open. Laten we daar nu eens naar kijken. Het volgende werkt:

```
declare
  l_child  xxx_child;
  l_parent xxx_parent;
begin
  l_child := xxx_child(id => 1, name => 'Martien', description =>
'Dad');
  l_child.show;
end;
```

Met als output:

```
Id: 1, name: Martien, description: Dad
```

Hier zie je dat er een variabele voor de child en de parent wordt aangemaakt. De child wordt geïnstantieerd via de default constructor (die ik niet heb aangemaakt) en vervolgens wordt de show methode aangemaakt.
Nu is het zo dat ik het sub-type heb aangemaakt om op basis van de data in de supertype extra functionaliteit te implementeren. En in mijn situatie ook dat ik uit BO-API's van EBS geïnstantieerde Super-types krijg, die ik op een sub-type manier wil behandelen. De geïmplementeerde Sub-Type functionaliteit is bedoeld om gebruikt te worden op de Super-Types.
Dus ik wil iets bereiken zoals het volgende:

```
declare
l_child xxx_child;
l_parent xxx_parent;
begin
  l_parent := xxx_parent(id => 1, name => 'Martien', description =>
'Dad');
  l_child := l_parent; -- PLS-00382: expression of wrong type
  l_child.show;
end;
```

Hier instantieer ik een parent volgens de default-constructor van het parent-type. Dus het geïnstantieerde object is van het super-type. Die ken ik toe aan de variabele van het sub-type. Dit levert een foutmelding op.
Het is niet mogelijk om een super-type aan een variabele van het sub-type toe te kennen, wanneer het is geïnstantieerd als een super-type. Maar het volgende is wel mogelijk:

```
declare
  l_child xxx_child;
  l_parent xxx_parent;
begin
```

```
  l_parent := xxx_child(id => 1, name => 'Martien', description =>
'Dad');
  l_child := treat(l_parent as xxx_child);
  l_child.show;
end;
```

Nu instantieer ik de parent als een child-type. En in de assignment geef ik aan Pl/SQL expliciet aan om de parent als een child object te behandelen. Dit is alleen mogelijk als het parent object expliciet als een child is geinstantieerd. Als je hem instantieert als een parent en het probeert te behandelen ('treat') als een child dan krijg je de melding: "ORA-06502: PL/SQL numeric or value error: cannot assign supertype instance to subtype".

Uitdaging: hoe cast je een super-type naar een sub-type

Eigenlijk is het best logisch. Want je kunt, om de vergelijking te trekken, ook een 'Auto' niet casten naar een Opel Zafira of een Ford Focus. Wel andersom. Want een Opel Zafira (al zullen sommige autofreaks het niet met mij eens zijn) is wel een auto, maar een auto hoeft geen Opel Zafira te zijn. Er zouden extra attributen (twee extra stoelen bijvoorbeeld) toegevoegd kunnen zijn. En de runtime-engine weet niet wat hij daar mee moet. Toch is mijn uitdaging ook wel logisch. Want ik wil standaard functionaliteit uitbreiden zonder de super-types aan te passen. En hoewel ik zeker weet dat mijn super-type in mijn sub-type past, gaat me dat zo niet lukken.

Mijn probleem is dus: Hoe cast ik mijn super-type naar een sub-type?.

Wanneer ik bijvoorbeeld met behulp van EBS BO-Create-API's een in EBS een customer account site aanmaak, dan heb ik geen probleem. Ik creëer een eigen sub-type die de EBS BO uitbreidt en wanneer ik het instantieer als mijn eigen child object type en het overdraag aan de API, zou het moeten werken. Want het is geïnstantieerd als mijn sub-type. Maar het probleem ligt in de 'update' en 'get' API's. Want voor een update moet ik eerst een 'get' doen, die de BO's uit EBS opvraagt. De get-api zal een object instantiëren als een EBS BO-type. En die wil ik dan 'casten' naar mijn eigen custom types om mijn eigen methods te gebruiken. En in beginsel is dat onmogelijk.
Maar na wat 'piekeren en prutsen' vond ik mezelf een oplossing. Wat er in feite gebeuren moet is dat er een child object wordt geïnstantieerd met de attribuut-waarden van de parent. Er moet dus een slimme constructor komen waar een parent object type in gaat en die een child object teruggeeft.

Stel nu dat ik een collectie van parents heb, zoals:

```
create or replace type xxx_parent_tbl is table of xxx_parent
```

Dan is het volgende mogelijk:

```
declare
  l_child xxx_child;
  l_parent xxx_parent;
  l_parent_tbl xxx_parent_tbl;
begin
  l_parent_tbl := xxx_parent_tbl();
  l_parent := xxx_parent(id => 1, name => 'Martien', description =>
'Dad');
  l_parent_tbl.extend;
  l_parent_tbl(l_parent_tbl.count) := l_parent;
  select xxx_child( ID,NAME,DESCRIPTION) into l_child  from table(l_
parent_tbl)  where rownum = 1;
  l_child.show;
end;
```

Hier wordt eerst een collection-variabele aangemaakt en geinstantieerd. Merk op: een Collection is ook een Object-type die geinstantieerd moet worden. Vervolgens wordt deze uitgebreid (met de extend method) en wordt er een geinstantieerde parent aan toegekend (op de nieuw aangemaakte plaats). Vervolgens wordt met een 'select into' de collection als een tabel uitgelezen (via de table-functie). Met de geselecteerde waarden wordt de default constructor van het child-type aangeroepen. Het resultaat komt dan in de child-variabele terecht. De where-clause (where rownum=1) is hier natuurlijk absoluut onnodig. Maar ik heb hem wel opgegeven omdat theoretisch een collectie meerdere waarden kan hebben.
Om mijn casting oplossing portable en herbruikbaar te maken, wil ik het op een iets meer dynamische manier doen. De Object-Typen van EBS kunnen een groot aantal attributen hebben en ik wil ze niet allemaal expliciet benoemen. Een upgrade van EBS zou me daarnaast dwingen om ook een upgrade van mijn custom types te doen.
Om de attributen van mijn parent op te vragen kan ik de volgende query gebruiken:

```
select attr_name from
user_type_attrs att
where att.type_name = 'XXX_PARENT'
order by attr_no;
```

De 'order by' is belangrijk, want de default constructor heeft alle attributen als een parameter in deze volgorde.
Voor een 'execute immediate' welke selecteert uit de parent-collection 'into' het child object heb ik een SQL statement als de volgende nodig:

```
begin select xxx_child( ID,NAME,DESCRIPTION) into :1  from table(:2
where rownum = 1;   end;
```

Ook hier is de where clause niet nodig, omdat de Collection maar een entry zal bevatten. Maar zo wordt gegarandeerd dat de select into in precies 1 rij resulteert.
Hiermee kan een functie worden gemaakt die de benodigde SQL statements genereert:

```
create or replace function xxx_get_type_cast_sql(p_parent in varchar2,
p_child  in varchar2)
  return varchar2 is
```

```
  l_sql varchar2(32767);
  cursor c_att(b_parent varchar2) is
  select *
  from all_type_attrs
  where type_name = b_parent
  order by attr_no;
begin
  l_sql := 'begin select '||p_child||'( ';
  for r_att in c_att(b_parent=>p_parent) loop
    if c_att%rowcount = 1 then
      l_sql := l_sql || r_att.attr_name;
    else
      l_sql := l_sql ||','|| r_att.attr_name;
    end if;
  end loop;
  l_sql := l_sql || ') into :1  from table(:2)  where rownum = 1;
end;';
  return(l_sql);
end xxx_get_type_cast_sql;
```

Hier mee kan ik een andere constructor aan mijn child object toevoegen met de het parent object-type als parameter:

```
create or replace type xxx_child under xxx_parent
(
-- Author  : MAKKER
-- Created : 17-03-2009 08:12:43
-- Purpose :
  member procedure show,
-- Member functions and procedures
  constructor function xxx_child return self as result,
  constructor function xxx_child(p_parent xxx_parent) return self as
result
)
/
create or replace type body xxx_child is
-- Version : $Id$
/* Member constructor, procedures and functions  */
  member procedure show is
  begin
    dbms_output.put_line('Id: ' || self.id || ', name: ' || self.name
||
                         ', description: ' || self.description);
  end;
  constructor function xxx_child return self as result is
  begin
    return;
  end;
  constructor function xxx_child(p_parent xxx_parent) return self as
result is
    l_sql         varchar(32767);
    l_parent_tab xxx_parent_tbl;
  begin
    -- Put parent in a collection;
    l_parent_tab := xxx_parent_tbl();
    l_parent_tab.extend;
    l_parent_tab(l_parent_tab.count) := p_parent;
    -- Create the sql
    l_sql := xxx_get_type_cast_sql(p_parent => 'XXX_PARENT'
                                  ,p_child  => 'XXX_CHILD');
    -- executed it dynamically
    execute immediate l_sql
      using in out self, in out l_parent_tab;
    return;
  end;
end;
/
```

Hier wordt de parent object parameter in een collection gestopt. Vervolgens wordt de sql gegenereerd en dynamisch uitgevoerd. De execute-immediate-parameter 'self' is een impliciete object variabele die in elke niet statische method van een type beschikbaar is en verwijst naar de 'eigen' instantie. Hiermee kan ik nu succesvol mijn parent 'casten' naar een child en de child methods gebruiken:

```
declare
  l_parent xxx_parent;
  l_child xxx_child;
begin
  l_parent := xxx_parent(1, 'Berend', 'Son');
  l_child :=  xxx_child(p_parent=>l_parent);
  l_child.show();
end;
```

Wat de volgende output oplevert:

```
Id: 1, name: Berend, description: Son
```

Conclusie en afrondende gedachten

Het direct 'casten' van super-types naar sub-types is niet mogelijk. Daarom wordt er in mijn oplossing niet direct gecast, maar dynamisch een child geïnstantieerd met de attributes van de parent. Deze manier gaat er wel vanuit dat aan de child geen extra attributen worden toegevoegd, of dat er op zijn minst een constructor is met dezelfde parameters als de parent. Alleen dan moeten alsnog alle attributen van de parent worden opgegeven en dat wilde ik nu juist vermijden. Misschien dat er met wat extra doordenken een en ander net wat slimmer gemaakt kan worden om ook deze requirement te omzeilen.
Bij elke instantiatie wordt het SQL-statement opgebouwd middels de function en de onderliggende query op user_type_attrs. Eigenlijk zou dat maar een keer gedaan hoeven worden. Als daar performance problemen bij worden ervaren dan kan deze functie eenmalig worden uitgevoerd en de uitkomst in de source van het object gekopieerd worden of in een globale package variabele. Maar ik verwacht hier niet direct een performance probleem. In mijn geval werd dit gebruikt in een batch interface waar potentieel een significante hoeveelheid records worden verwerkt.
Gebruik execute immediate constructies met beleid. In dit geval ben ik er zeker van dat dit niet tot rare performance problemen leidt. De SQL die de functie oplevert zal altijd dezelfde zijn zolang het parent-object-type niet gewijzigd wordt voor waar het de attributen betreft. Aangezien ook gebruik wordt gemaakt van bind-variables in de execute immediate (door de 'using...' clause) zal het sql-statement ook maar eenmalig in de SQL-Area voorkomen.

Martien van den Akker is Technical Architect bij Darwin IT-Professionals.