Objekte machen das Leben leichter – reloaded

Jürgen Sieben, ConDeS GmbH & Co. KG

In der letzten Folge wurden Objekte in sehr einfacher Form eingesetzt, um den Code eleganter zu machen. Eine Erweiterung dieser Strategie setzt ein komplett neues Denken über Code in Gang, das zwar in objektorientierten Programmier-Umgebungen seit Langem bekannt ist, im Umfeld von PL/SQL allerdings nur sehr selten eingesetzt wird: das Interface.

Als Beispiel nehmen wir an, in einem Programm müssten die Metadaten von Bankanweisungen verwaltet werden. Diese können unterschiedlichen Typs sein, wie zum Beispiel eine Überweisung, ein Dauerauftrag oder ein Bankeinzug. Diesen Bankanweisungen gemein sind einige Attribute, wie etwa die Bankverbindung, der Betrag, der Adressat etc. Andere Angaben sind auf den jeweiligen Typ beschränkt, so ist beispielsweise nur bei einem Dauerauftrag die Angabe eines Zeitraums erforderlich, vielleicht könnte ein Dauerauftrag auch für eine bestimmte Zeit ausgesetzt werden.

In unserer Anwendung benötigen wir nun Code, der mit diesen verschiedenen Bankanweisungs-Typen umgehen kann. Nehmen wir an, wir müssten eine Überweisung erfassen, signieren und ausführen können; je nach Zustand, in dem sich die Überweisung befindet, könnte es möglich sein, sie zu stornieren. Wir benötigen also Packages, die diese Methoden implementieren. Da die unterschiedlichen Bankanweisungen jedoch unterschiedliche Geschäftslogik erfordern (der Dauerauftrag unterliegt vielleicht anderen Freigabeverfahren als eine Überweisung, Bankeinzüge dürfen nur von bestimmten Mitarbeitern erfasst werden etc.), endet das Ganze mit mehreren Packages für jede Art der Banküberweisung, die zum Teil extrem ähnliche Arbeiten verrichten.

Auf der anderen Seite werden wir Code schreiben, der mit allen Typen von Bankanweisungen umgehen können soll, etwa in zentralen Such-Dialogen oder in zentralisierten Packages zur Signatur von Bankanweisungen. In diesen Packages benötigen wir nun „CASE“-Anweisungen, die – je nach Typ – unterschiedliche Packages aufrufen. Da sich die Funktionalität so stark ähnelt, wird man eventuell

```
create or replace type bankanweisung
  authid definer
as object(
  id number,
  iban char(22 byte),
  bic varchar2(11 byte),
  empfaenger varchar2(50 char),
  betrag number,
  -- ...
  member function get_iban(
    self in out nocopy bankanweisung)
    return varchar2,
  member function get_bic_11
    return varchar2,
  member procedure speichern(
    self in out nocopy bankanweisung),
  member procedure signieren(
    self in out nocopy bankanweisung),
  member procedure anweisen(
    self in out nocopy bankanweisung),
  member procedure widerrufen(
    self in out nocopy bankanweisung,
    p_grund in varchar2)
  -- ...
) not final not instantiable;
/
```

Listing 1: Basistyp der Bankanweisung

darüber nachdenken, die „CASE"-Anweisungen in den Fällen, in denen mehrere Typen von Bankanweisungen gleich arbeiten, auf den gleichen Code zeigen zu lassen, was dazu führt, dass nun die einfache Zuordnung „Ein Typ - ein Package" nicht mehr durchgehalten wird. Das Chaos ist komplett.

Um dieses Chaos zu umgehen, wird bei objektorientierten Sprachen ein Interface verwendet: Ein Objekt beinhaltet alle Attribute, die alle Bankanweisungen enthalten sowie die Methoden, die unsere Anweisungen aufweisen können sollen. Dieses Objekt dient anschließend als „Blaupause" zur Erstellung der unterschiedlichen Arten von Anweisungen. Dieses Objekt nennen wir „BANKANWEISUNG" und definieren es wie in *Listing 1*.

Der Code ist wie eine Package-Spezifikation zu lesen. Die Attribute „IBAN", „BIC" etc. sind mit öffentlichen Package-Variablen vergleichbar, die Methoden entsprechen den Package-Methoden. Zusätzlich enthalten all diese Methoden einen Verweis auf sich selbst („SELF"). Es ist darauf zu achten, dass dieser Parameter genau so heißen muss und nicht etwa „P_SELF" oder ähnlich. Zum Schluss folgen noch zwei Klauseln, die festlegen, dass wir von diesem Objekt andere Objekte ableiten können („NOT FINAL") und dass es nicht möglich ist, dieses Objekt selbst zu benutzen. Dazu im Folgenden mehr.

Zu dieser Deklaration gehört nun noch, wie auch bei Packages, eine Implementierung des Objekts, in der festgelegt wird, was das Objekt tun soll, wenn eine der gezeigten Methoden aufgerufen wird. Dabei orientieren wir uns an folgender Regel: Methoden, die für alle Typen von Bankanweisungen gleich sind, werden komplett implementiert; Methoden, die je nach Typ unterschiedlich sind, werden nur als Stub angelegt („begin null; end;"). Im Beispiel können die Zugriffsmethoden „GET_IBAN" und „GET_BIC_11" (die Methoden liefern eine lesbare Version der IBAN gemäß DIN 5008 mit Leerzeichen sowie eine auf elf Zeichen erweiterte BIC) direkt im Objekt implementiert werden, während die Methoden zum Speichern, Signieren, Ausführen und Widerrufen wohl eher als Stub implementiert werden.

Aufbauend auf diesem Objekt können nun weitere Bankanweisungs-Typen erstellt werden. Das Besondere: Diese Objekte basieren auf dem oben erstellten Objekt und erweitern beziehungsweise überschreiben es. Die bereits implementierten Methoden müssen nicht neu implementiert werden, sondern können bei den abgeleiteten Objekten weggelassen werden. Benötigen wir also ein Objekt „UEBERWEISUNG", wird dies von „BANKANWEISUNG" abgeleitet und überschreibt die bislang nicht implementierten Methoden. *Listing 2* zeigt ein solches Objekt.

Dass dieser Typ von „BANKANWEISUNG" abgeleitet ist, erkennen Sie an der Klausel „UNDER" in der Deklaration. Da dieser Typ „INSTANTIABLE" ist (die Klausel „NOT INSTANTIABLE" fehlt), kann von ihm ein konkretes Objekt abgeleitet werden. Dieses Objekt wird über eine Methode erzeugt, die als „Konstruktor" bekannt ist und genauso heißt wie der Typ, der durch die Methode erzeugt wird. Dieses Prinzip hatten wir schon bei den Nested Tables kennengelernt. Der Konstruktor erwartet einige der Attribute wie etwa die IBAN, eventuell eine BIC, den Empfänger sowie den Betrag und belegt, nach entsprechender Prüfung, die internen Attribute mit den ermittelten Werten. Eine Sequenz könnte in dieser Methode eine ID ermitteln, unter der das Objekt später in einer Tabelle abgelegt werden kann.

Auch dieser Typ wird einen Objektkörper erhalten. Er benötigt allerdings eine komplexere Implementierung und ist es sinnvoll, nicht in den Objektkörpern die Logik zu implementieren, sondern diese in entsprechende Packages auszulagern. Der Grund dafür ist, dass Packages über mächtigere Möglichkeiten verfügen als Objektkörper (zum Beispiel private Hilfsmethoden und Packagevariablen). Wie das funktioniert, steht im Code zum Artikel unter *„https://github.com/j-sieben/Oracle_OO/blob/master/Objekte%20als%20Interface/Script.sql"*. Nun kann also für das Objekt „UEBERWEISUNG" ein Package „UEBERWEISUNG_PKG" angelegt werden, das die Implementierung der Methoden übernimmt. Bei eher trivialem Code lassen sich jedoch auch die Methoden direkt im Objekt implementieren. Auch dieses Objekt ist „NOT FINAL", sodass zum Beispiel ein Dauerauftrag von diesem Typ abgeleitet werden könnte: Er erbt alle Methoden der Überweisung, ergänzt um die Dinge, die bei einer Dauerauszahlung anders gemacht werden müssen.

Was bringt uns das nun? Oberhalb der Packages, die die Logik für die verschiedenen Arten der Bankanweisungen implementieren, sitzt nun eine Objekthierarchie mit aufeinander aufbauenden Objekttypen. Dieser zusätzliche Over-

```
create or replace type UEBERWEISUNG under BANKANWEISUNG(
  overriding member procedure speichern(
    self in out nocopy ueberweisung),
  overriding member procedure signieren(
    self in out nocopy ueberweisung),
  overriding member procedure anweisen(
    self in out nocopy ueberweisung),
  overriding member procedure widerrufen(
    self in out nocopy ueberweisung,
    p_grund in varchar2),
  constructor function ueberweisung(
    self in out nocopy ueberweisung,
    p_iban in varchar2,
    p_bic in varchar2 default null,
    p_empfaenger in varchar2,
    p_betrag in number)
    return self as result
) not final;
/
```

Listing 2: Implementierung des Typs „UEBERWEISUNG“

```
declare
  l_anweisung BANKANWEISUNG;
  l_betrag number := 123.45;
begin
  l_anweisung := UEBERWEISUNG('DE…', 'ABCDEF…', 'Willi Mueller', l_betrag);
end;
/
```

Listing 3

head bringt uns mehrere entscheidende Vorteile: Zunächst ist es möglich, eine Variable vom Typ „BANKANWEISUNG“ zu erstellen und in dieser ein beliebiges, von diesem Typ abgeleitetes Objekt zu speichern. In der Variablen dieses Typs könnte also auch eine Überweisung gespeichert werden *(siehe Listing 3)*.

Der Vorteil: Wir können nun Code schreiben, der mit beliebigen Bankanweisungen umgehen kann, auch mit solchen, die wir bislang noch gar nicht definiert haben: Eine Methode mit einem Parameter vom Typ „BANKANWEISUNG“ kann mit Instanzen aller abgeleiteten Typen aufgerufen werden.

Dann stellt dieses Verfahren sicher, dass eine Überweisung unmittelbar die IBAN ausgeben kann, obwohl der Typ „UEBERWEISUNG“ die Funktion gar nicht implementiert, und zwar einheitlich nach dem Verfahren, dass der Obertyp implementiert. Der Vorteil hier: Gleiche Funktionalität muss nicht mehrfach erstellt oder in Utility-Packages ausgelagert werden. Wenn der Obertyp etwas bereits kann, muss ein abgeleiteter Typ dies nicht neu implementieren. Erst, wenn er etwas anders machen muss als sein hierarchischer Vorgänger, überschreibt er die Implementierung in einer eigenen Methode.

Nebenbemerkung: Im obigen Code-Abschnitt kommt eine Hilfsvariable „L_BETRAG“ vor, um den Betrag zu übergeben. Hier, wie auch an anderen Stellen in SQL und PL/SQL (XML und JSON im Beispiel), tritt ansonsten ein Problem überschießender Internationalisierungsfreude zutage: Wird ein Betrag mit einem Dezimalpunkt übergeben, wird dieser nicht als gültige Zahl erkannt, weil dieser in den NLS-Einstellungen als Tausender-Trenner angesehen wird. Wenn irgendjemand von Oracle dies liest: Kann man dieses Problem nicht endlich einmal lösen?

Durch die Referenz auf einen Vorgänger ist zudem sichergestellt, dass alle Methoden aller hierarchischen Vorgänger im aktuellen Objekt vorhanden sind. Der Vorteil ist hier, dass Sie keine Fallunterscheidungen beim Ansprechen eines Objektes mehr benötigen. Hierzu als Beispiel: Der Typ „DAUERAUFTRAG“ benötigt eine Methode zum Aussetzen von Überweisungen für einen bestimmten Zeitraum. Daher implementiert der Obertyp „BANKANWEISUNG“ diese Methode als Stub. Nun kann jedes beliebige Objekt angewiesen werden, eine Aussetzung durchzuführen, ohne dass dies zu einem Kompilierfehler führt. Wenn Sie mögen, werfen Sie in der Oberklasse einen Fehler, dass für diesen Typ der Aufruf dieser Methode nicht zulässig sei, um diesen Weg zu sichern, aber der aufrufende Code muss nun nicht mehr zwischen den verschiedenen Ausprägungen der Bankanweisung unterscheiden können. Kann ein konkreter Typ eine Zahlung aussetzen, wird er implementieren, wie dies geschehen soll, ansonsten fällt die Ausführung auf den Stub des Obertyps zurück und macht entweder nichts oder wirft einen Fehler – ganz, wie Sie wollen.

Fazit

Code wird durch dieses Vorgehen deutlich einfacher zu schreiben, wenn auch zu Beginn nicht notwendigerweise einfacher zu verstehen. Dennoch ist die Flexibilität bestechend, die darin besteht, dass Sie einen neuen Bankanweisungs-Typ nachträglich ganz einfach integrieren können, ohne bestehenden Code ändern zu müssen. Schön ist auch, dass wir Funktionalität dort implementieren können, wo sie am sinnvollsten ist, in unserem Beispiel in dem Package, das die Methode des jeweiligen Typs implementiert. Hat man sich an diese Systematik erst einmal gewöhnt, wird komplexer Code deutlich eleganter.

Hinweis: Das Skript zur Dokumentation der Objekt-Hierarchie finden Sie unter *„https://github.com/j-sieben/Oracle_OO/blob/master/Objekte%20als%20Interface/Script.sql“*.

Jürgen Sieben
j.sieben@condes.de