Vom Nutzen vertikaler Dienste

Jürgen Sieben, ConDeS GmbH & Co. KG

In der letzten Folge wurden Objekte als Schnittstelle verwendet, dieser Artikel lässt es ein wenig schlichter angehen und kümmert sich wieder um normale Programmierung.

Obwohl: Wen der Ansatz der Objektorientierung grundsätzlich interessiert und wer Anwendungsbeispiele sucht, sollte sich einmal die Neufassung der ursprünglich von Steven Feuerstein entwickelten utPLSQL-Suite in Version 3 ansehen. Das extrem elegante Test-Interface ist nur durch Objektorientierung möglich und stellt eine sehr inspirierende Anwendung dieses Konzepts dar. Alternativ kann man auch gern einen Blick auf das PL/SQL Instrumentation Toolkit *(PIT, siehe „http://github.com/j-sieben/PIT“)* aus der Feder des Autors werfen, das dieses Mittel einsetzt, um Ausgabe-Module ansprechen zu können, ohne zur Kompilierzeit wissen zu müssen, welche Ausgabe-Module vorhanden sind. Dieses Package ist im Übrigen ein Beispiel für einen vertikalen Dienst, womit wir dann beim Thema dieser Folge wären.

Unter einem „vertikalen Dienst“ wird in dieser Folge die Funktionalität verstanden, die Schichten-übergreifend (wie View-, Control- oder Model-Layer) vorhanden sein muss. Klassische Bestandteile dieser vertikalen Dienste sind Methoden zur Code-Instrumentierung, Fehlerbehandlung, Performance-Messung etc. Im Zusammenhang dieses Artikels soll der Begriff allerdings allgemeiner als Sammlung von Utilities verstanden werden, die in der Entwicklung von Software innerhalb der Datenbank an allen möglichen Stellen verwendet werden.

Jeder Entwickler sollte ein gut gepflegtes Set dieser Werkzeuge besitzen, um zum Beispiel wiederkehrende Anforderungen einfacher lösen zu können. PIT ist ein solches Werkzeug, das sich um das Logging, die Fehlerbehandlung, aber auch die Ausgabe von Meldungen an die Anwendungen und um Assertionen kümmert. Dazu ein Blick auf den letzten Punkt, die Assertionen, um den Wert solcher Hilfsmittel am praktischen Beispiel zu erfahren. In einem offen zugänglichen Produktiv-Code (programmiert von Oracle) steht das folgende, stark gekürzte Beispiel für eine verbesserungsbedürftige Programmierung *(siehe Listing 1)*.

Dass dies ein schlechtes Beispiel für Programmierung ist, erkennt man schon an der Schwierigkeit zu ermitteln, was dieser Code eigentlich tut. Offensichtlich werden Benutzernamen geprüft und in eine Liste mit guten oder schlechten Benutzern aufgeteilt. Verbesserungspotenzial findet sich zuhauf:

- „L_VALID“ dient nur dem Zweck, eine „CASE“-Anweisung nachzuprogrammieren. Das wäre mit einer einfachen „CASE“-Anweisung deutlich leichter gewesen
- Die dauernde Wiederholung gleicher Code-Blöcke legt die Auslagerung in Hilfsmethoden nahe
- Der Code beschäftigt sich fast ausschließlich mit den Fehlern, die eigentliche Aufgabe wird überhaupt nicht klar, sie versteckt sich gegen Ende in einer der Prüf-Routinen

Wenn wir dies verbessern möchten, sollten wir über eine Assertions-Methode nachdenken, also eine einfache Methode, die einen Sachverhalt prüft, nichts tut, wenn die Bedingung erfüllt ist, und anderenfalls einen Fehler wirft. Im Kern ist so eine Methode trivial *(siehe Listing 2)*. Refaktorisieren wir das Beispiel und lagern die Prüfung gegen die Tabellen in Hilfsmethoden („CHECK_USER_NOT_EXISTING“, „CHECK_DUPLICATE_USER“) aus, sieht das Ergebnis wie in *Listing 3* aus.

Zunächst werden die Assertionen durchgeführt. Schlägt eine Assertion fehl, wird zum „EXCEPTION“-Block verzweigt und der aktuelle Benutzer mit der passgenauen Meldung in die Liste der ungültigen Benutzer eingefügt, ansonsten in die Liste der gültigen. Man sieht, wie drastisch sich diese einfache Hilfsmethode auswirkt:

- Wir benötigen überhaupt keine „CASE“-Anweisung mehr, diese Aufgabe übernimmt die Exception der Assertions-Methode
- Wir überblicken die Aufgabe der Methode sofort: Prüfe, ob der Benutzer valide ist, und speichere ihn anschließend in die entsprechende Liste. Hauptgrund ist, dass die Hilfsmethode „ASSERT“ schon durch ihren Namen anzeigt, was hier passiert

Solche Hilfsmethoden sollten also stets zur Hand sein und nicht jedes Mal neu programmiert werden müssen. Hat man andererseits ein solches Hilfspackage, das auf allen Code-Ebenen verwendet werden kann, lohnt sich dafür ein wenig mehr Aufwand. Hier ist es eine einfache Assertions-Methode, aber vielleicht sind ja speziellere Methoden sinnvoll? Eine Assertions-Methode, die auf „NULL“ prüft etwa, oder eine, die mindestens eine Zeile für eine übergebene „SELECT“-Anweisung erwartet oder im Gegenteil gerade keine Zeile? Das ist der tiefere Sinn dieser Packages: Dinge werden durch spezielle Hilfsmethoden konsistent und einfach gelöst, der hierfür nötige Aufwand muss jedoch nur einmal erbracht werden.

```
declare
  <...>
begin
  for j in 1..l_emails.count loop
      if eba_stds_fw.get_preference_value('USERNAME_FORMAT') = 'EMAIL' then
          if <...> then
              apex_collection.add_member(
                  p_collection_name => 'EBA_STDS_BULK_USER_INVALID',
                  p_c001            => l_username,
                  p_c002            => apex_lang.message('MISSING_AT_SIGN'));
              l_valid := false;
          end if;

          if <...> and l_valid then
              apex_collection.add_member(
                  p_collection_name => 'EBA_STDS_BULK_USER_INVALID',
                  p_c001            => l_username,
                  p_c002            => apex_lang.message('MISSING_DOT'));
              l_valid := false;
          end if;
      end if;

      if <...> and l_valid then
          apex_collection.add_member(
              p_collection_name => 'EBA_STDS_BULK_USER_INVALID',
              p_c001            => upper(l_username),
              p_c002            => apex_lang.message('USERNAME_TOO_LONG'));
          commit;
          l_valid := false;
      end if;

      if l_valid then
          for c1 in (select username
              from eba_stds_users
              where upper(username) = upper(l_username))
      loop
              wwv_flow_collection.add_member(
                  p_collection_name => 'EBA_STDS_BULK_USER_INVALID',
                  p_c001            => upper(l_username),
                  p_c002            => apex_lang.message(,ALREADY_IN_ACL‘));
              l_valid := false;
              exit;
          end loop;
      end if;

      if l_valid then
          for c1 in (select c001
                  from wwv_flow_collections
                  where collection_name = 'EBA_STDS_BULK_USER_VALID'
                  and c001 = upper(l_username))
          loop
              apex_collection.add_member(
                  p_collection_name => 'EBA_STDS_BULK_USER_INVALID',
                  p_c001            => upper(l_username),
                  p_c002            => apex_lang.message(,DUPLICATE_USER‘));
              l_valid := false;
              exit;
          end loop;
      end if;

      if l_valid then
          apex_collection.add_member(
              p_collection_name => 'EBA_STDS_BULK_USER_VALID',
              p_c001            => upper(l_username));
              commit;
      end if;
  end loop;
end;
```

Listing 1: Schlechtes Programmier-Beispiel

Hilfspackages dieser Art sind zudem in allen Projekten wiederverwendbar und vereinheitlichen so die Art, wie programmiert wird. Wichtig bei der Auswahl des passenden Hilfspackage ist allerdings, dass diese sich nicht auf eine konkrete Anwendungsschicht beziehen, etwa auf Apex als View-Layer. Der Grund liegt darin, dass wir mit diesem Ansatz nun wieder gegen die Idee des vertikalen Dienstes verstoßen würden, denn es ist nicht ausgeschlossen, dass eine Methode auf einer tieferen Anwendungsschicht sich nun mit den Anforderungen von Apex konfrontiert sieht, was eigentlich nicht passieren soll.

Insofern ist die Implementierung unserer „ASSERT"-Methode noch generisch genug, allerdings sollten wir überlegen, ob es nicht günstig wäre, die Methode in einen Kontext der Verwaltung von Meldungstexten einzubinden. Der Beispiel-Code delegiert die Verwaltung der tatsächlichen Meldung an Apex („apex_lang.message(<Name der Meldung>)"), was im Kontext einer Apex-Anwendung in Ordnung ist, aber einen Verstoß gegen die Schicht-Unabhängigkeit darstellt: Code der Geschäftslogik müsste sich nun darauf verlassen, dass Apex und damit auch die Verwaltung der Meldungen vorhanden sind, um auf gleiche Weise zu funktionieren.

Da Assertions-Methoden jedoch immer Meldungstexte benötigen, könnte man fordern, dass die entsprechenden Texte einfach übergeben werden müssen. Aber was ist nun mit internationalisierter Software? Wie übergeben wir die Meldung in der passenden Sprache? Was ist mit Ersetzungen innerhalb der Meldungstexte? Schnell wird klar, dass der Aufwand nun doch wieder sehr steigt. Das ist grundsätzlich kein Problem, nur sollte der Aufwand nicht dort steigen, wo die Hilfsmethode verwendet wird, sondern maximal in der Hilfsmethode selbst, denn dann steigt der Aufwand nur einmalig.

PIT macht dies so, indem es eigene Meldungstexte verwaltet und diese – umgekehrt – auch in Apex ausgeben kann. Daher entfällt die Notwendigkeit, Meldungen außerhalb des Codes im Metadatenmodell von Apex vorzuhalten, zudem sind Meldungen nun auch, als Teil des vertikalen Dienstes, unabhängig von den Anwendungsschichten. In PIT sähe der Code von oben dann wie in *Listing 4* aus.

```
procedure assert(
   p_condition in boolean,
   p_message in varchar2)
as
begin
   if not p_condition then
       raise_application_error(-20000, p_message);
   end if;
end assert;
/
```

Listing 2: Keimzelle einer einfachen Assertions-Methode

```
declare
   <...>
begin
   for j in 1..l_emails.count loop
       l_is_email := eba_stds_fw.get_preference_value('USERNAME_FORMAT') = 'EMAIL';

       -- VALIDATION
       if l_email then
           assert(<...>, apex_lang.message('MISSING_AT_SIGN'));
           assert(<...>, apex_lang.message('MISSING_DOT'));
       end if;
       assert(<...>, apex_lang.message('USERNAME_TOO_LONG'));
       assert(check_user_not_existing, apex_lang.message('ALREADY_IN_ACL'));
       assert(check_duplicate_user, apex_lang.message('DUPLICATE_USER'));

       -- PROCESSING
       apex_collection.add_member(
           p_collection_name => 'EBA_STDS_BULK_USER_VALID',
           p_c001            => upper(l_username));
   end loop;

exception
   when others then
       apex_collection.add_member(
           p_collection_name => 'EBA_STDS_BULK_USER_INVALID',
           p_c001            => upper(l_username),
           p_c002            => sqlerrm);
end;
```

Listing 3: Refaktorisierte Version mit ausgelagerten Hilfsmethoden

```
declare
   <...>
begin
   for j in 1..l_emails.count loop
      l_is_email := eba_stds_fw.get_preference_value('USERNAME_FORMAT')
= 'EMAIL';
      if l_email then
         pit.assert(<...>, msg.MISSING_AT_SIGN);
         pit.assert(<...>, msg.MISSING_DOT);
      end if;
      pit.assert(<...>, msg.USERNAME_TOO_LONG);
      pit.assert(check_user_not_existing, msg.ALREADY_IN_ACL);
      pit.assert(check_duplicate_user, msg.DUPLICATE_USER);

      apex_collection.add_member(
         p_collection_name  => 'EBA_STDS_BULK_USER_VALID',
      p_c001                => upper(l_username));

   end loop;
exception
   when others then
      apex_collection.add_member(
         p_collection_name  => 'EBA_STDS_BULK_USER_INVALID',
         p_c001             => upper(l_username),
         p_c002             => sqlerrm);
end;
```

Listing 4: Fassung mit PIT

So kann man beides haben: gute Code-Qualität und geringstmöglichen Programmieraufwand.

Jürgen Sieben
j.sieben@condes.de