Oracle PL/SQL. Wprowadzenie

Oracle PL/SQL. Wprowadzenie
Autorzy: Bill Pribyl, Steven Feuerstein
Tłumaczenie: Bartłomiej Garbacz
ISBN: 83-7197-727-1
Tytuł oryginału: Learning Oracle PL/SQL
Format: B5, stron: 412
Przykłady na ftp: 118 kB

PL-SQL – język programowania systemu Oracle, przeznaczony do tworzenia procedur magazynowanych – zapewnia ogromne możliwości piszącym oprogramowanie baz danych. PL/SQL rozszerza standard języka relacyjnych baz danych SQL poprzez umożliwienie korzystania z takich konstrukcji, jak: pętle, instrukcje IF-THEN, złożone struktury danych czy szerokie możliwości kontroli operacji transakcyjnych. Wszystkie z nich są ściśle zintegrowane z serwerem bazy danych Oracle.

„Oracle PL/SQL. Wprowadzenie" daje Czytelnikowi możliwość pełnego zrozumienia języka PL/SQL bez względu na to, czy jest początkującym, czy doświadczonym programistą. W niniejszej książce przedstawiono następujące zagadnienia:

- cechy języka PL/SQL i korzyści wynikających z jego używania;
- składnia i przykłady zastosowania wszystkich głównych konstrukcji języka;
- tworzenie i wykorzystywanie procedur, funkcji oraz pakietów magazynowanych;
- tworzenie aplikacji opartych na sieci Internet;
- zabezpieczanie programów w języku PL/SQL przed atakami z zewnątrz;
- korzyści wynikające z wykorzystania narzędzi wspomagających programowanie, pochodzących od innych dostawców;
- wykorzystanie języka PL/SQL do programowania zadań związanych z użyciem poczty elektronicznej, języka Java oraz sieci Internet.

„Oracle PL/SQL. Wprowadzenie" zawiera szczegółowy opis konstrukcji języka we wszystkich wersjach od Oracle7 do Oracle9i, podparty przykładami programów dostępnych także pod adresem http://oracle.oreilly.com. Autorami jej są eksperci języka PL/SQL Bill Pribyl oraz Steven Feuerstein. Książka daje solidne podstawy każdemu programiście baz danych i administratorowi, który zmuszony jest do poznania języka PL/SQL.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

3

Programowanie

W niniejszym rozdziale omówiono następujące zagadnienia:

- Informacje o programie przykładowym
- Pierwsze zadanie programistyczne
- Pobieranie informacji o liczbie książek za pomocą funkcji
- Tworzenie elastycznego kodu
- Wykorzystanie pakietów PL/SQL w celu organizacji kodu
- Przejście na wyższy poziom
- Dalsza droga

Po zapoznaniu się z podstawami języka PL/SQL Czytelnik jest przygotowany do tworzenia programów bardziej rozbudowanych niż zwykłe wyświetlanie komunikatu. W niniejszym rozdziale przedstawiono sposób rozpoczęcia tworzenia aplikacji obsługi katalogu biblioteki. W kolejnych częściach książki opisano sposób jej dalszej rozbudowy. Nowymi elementami języka PL/SQL, które przedstawiono w niniejszym rozdziale, są *procedury* (*procedures*), *funkcje* (*functions*) oraz *pakiety* (*packages*). Czytelnik dowie się, do czego te elementy służą, jak je konstruować oraz w jaki sposób używać w celu osiągnięcia wymaganych celów.

Informacje o programie przykładowym

Przykładowe zadanie programistyczne, opisane w niniejszej książce, polega na próbie utworzenia systemu, który służyłby do katalogowania oraz wyszukiwania książek w bibliotece — jest to rodzaj elektronicznego katalogu. W przypadku tej hipotetycznej biblioteki zakłada się, że wszelkie dane operacyjne znajdują się w bazie danych Oracle. Istnieje więcej niż jeden sposób gromadzenia danych — dotyczących tytułów, autorów itp. — w bazie danych. Jednym z nich jest ręczne wpisywanie danych przez bibliotekarza. W kolejnych rozdziałach zostaną opisane metody automatycznego ładowania danych z odległych źródeł oraz metody wyszukiwania przez użytkownika danych znajdujących się w katalogu.

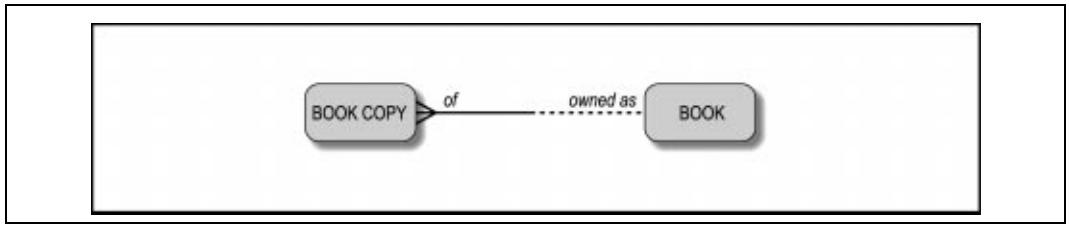
Istnieje konieczność spełnienia dwóch wymagań o podstawowym znaczeniu dla działania opisywanej aplikacji:

- Należy umożliwić tworzenie wpisów do katalogu dla każdej nowej książki.
- Należy udostępnić możliwość określania liczby egzemplarzy danej książki, znajdujących się w zasobach biblioteki.

W celu spełnienia pierwszego wymagania należy utworzyć procedurę PL/SQL, służącą do wprowadzania danych do bazy danych. Program umożliwiający spełnienie drugiego warunku wymaga zastosowania funkcji PL/SQL. Przed opisaniem sposobu tworzenia tych elementów należy najpierw przedstawić metody projektowania struktury samej bazy danych.

Model danych

Podobnie jak w przypadku większości projektów tworzonych przez programistów PL/SQL, struktura bazy danych dotyczących książek w bibliotece została wcześniej zaprojektowana i utworzona na podstawie wymagań postawionych przez przyszłego użytkownika. Podzbiór logicznego projektu bazy danych, związany ze wspomnianymi wcześniej wymaganiami, można ograniczyć do informacji o każdym egzemplarzu książki w bibliotece. Rysunek 3.1 przedstawia tak zwany *diagram związków encji* (*entity-relationship diagram, ERD*).



Rysunek 3.1. Zależność między wpisem (informacją) dotyczącym książki a jej fizycznie istniejącymi egzemplarzami

Diagramy takie w zwięzłej i uproszczonej formie prezentują informacje o realnym świecie. Opisane w odpowiedni sposób prostokąty są *encjami* (*entities*) bazy danych, a linie pomiędzy nimi oznaczają *związki* (*relationships*) pomiędzy encjami. Relacyjne bazy danych przedstawiają rzeczywistość w formie zbioru struktur danych (które przechowują informacje o pewnych obiektach) oraz zbioru metod (które definiują powiązania między obiektami). Encja `book` reprezentuje podstawowe informacje o książce (tytuł, autor itd.), które zawiera każda biblioteka. Encja `book_copy` zawiera informacje o fizycznie istniejących egzemplarzach danej książki.

Pewne zdziwienie może wywołać fakt, że całość rozdzielono na dwie encje zamiast użycia jednej. Wykorzystanie tylko jednej encji spowodowałaby jednak problemy w późniejszym czasie, ponieważ wystąpiłaby konieczność kopiowania informacji dotyczących książki razem z informacjami dotyczącymi jej pojedynczych egzemplarzy, co z pewnością byłoby niewłaściwym sposobem wykorzystania zasobów komputera i wysiłku ludzkiego. Pełna dyskusja na temat *normalizacji* (*normalization*) baz danych (procesu organizowania danych w postaci tabel zgodnie z ich wewnętrzną strukturą) wykracza poza zakres treści niniejszej książki. Warto jednak podkreślić, że główna idea polega na przechowywaniu ważnych informacji w jednym i tylko w jednym miejscu. Każdy kolejny egzemplarz danej książki wymaga utworzenia jedynie dodatkowego rekordu zawierającego identyfikator, który w powyższym przypadku jest numerem identyfikacyjnym pochodzącym z samoprzylepnej naklejki z kodem kreskowym.

W tym miejscu należy wyjaśnić znaczenie linii przedstawiających relację, pokazaną na rysunku 3.1. Z treści tego rysunku wynikają następujące fakty dotyczące realnej sytuacji:

- Każdy egzemplarz danej książki jest egzemplarzem tyko jednej książki.
- Każda książka może posiadać jeden lub większą liczbę swoich egzemplarzy.

Relacja taka jest znana jako *relacja jeden-do-wielu* (*one-to-many relationship*): jedna pozycja książkowa i wiele egzemplarzy danej książki.

Fakty powyższe wydają się być truizmem. Jednak przed przystąpieniem do właściwego projektowania bazy danych należy dokonać analizy rozwiązywanego problemu na takim właśnie, podstawowym poziomie. Rozłożenie informacji dotyczących projektowanej aplikacji na szereg pozornie banalnych faktów i ustalenie wszystkich podstawowych związków pomiędzy jej elementami z pewnością ułatwi pracę podczas tworzenia kodu programu.

Projektowanie fizycznej struktury bazy danych

Faktyczna struktura bazy danych odzwierciedla model związków encji — każda encja odpowiada tabeli w bazie danych. Poniżej przedstawiono kod w języku SQL, służący do utworzenia takiej tabeli:

```
CREATE TABLE books (
   isbn VARCHAR2(13) NOT NULL PRIMARY KEY,
   title VARCHAR2(200),
   summary VARCHAR2(2000),
   author VARCHAR2(200),
   date_published DATE,
   page_count NUMBER
);

CREATE TABLE book_copies(
   barcode_id VARCHAR2(100) NOT NULL PRIMARY KEY,
   isbn VARCHAR2(13) NOT NULL,
   CONSTRAINT book_copies_isbn_fk FOREIGN KEY (isbn) REFERENCES books (isbn)
);
```

Z powodów zrozumiałych dla specjalistów od modelowania danych, encjom nadaje się nazwy w formie rzeczowników w liczbie pojedynczej (`book`, `book_copy`), natomiast tabelom — w liczbie mnogiej (`books`, `book_copies`). Ponadto warto wykonać graficzną reprezentację tabeli — na rysunku 3.2 przedstawiono możliwą postać zdefiniowanych tabel zapełnionych pewnymi danymi.

Zapoznawszy się z danymi zawartymi w tabelach, Czytelnik z pewnością zwrócił uwagę na pewne problemy. Przykładowo, dane w kolumnie *author* są nieprawidłowe i nie jest możliwe poprawne przechowanie informacji o kilku autorach jednej książki. W dalszej części niniejszej książki zostanie opisany sposób poprawienia tych wad.

Pierwsze zadanie programistyczne

Pierwszym, przykładowym zadaniem będzie utworzenie programu w języku PL/SQL, który posłuży do dodawania nowych książek do bazy danych. Oczywiście, zamiast takiego programu można po prostu wykonać instrukcję `INSERT` (lub dwie takie instrukcje) języka SQL:

books

ISBN	title	summary	author	date_published	page_count
1-56592-335-9	Programowanie w Oracle PL/SQL	Podręcznik encyklopedyczny dla programistów PL/SQL wraz z przykładami oraz wskazówkami na temat programowania.	Feuerstein, Steven, Bill Pribyl	01-WRZ -1997	987
0-14071-483-9	Ryszard III	Współczesne wydanie znanego dramatu historycznego Szekspira, w którym zdradziecki król próbuje zdobyć koronę, lecz ginie na bitwie po stracie konia.	William Shakespeare	01-SIE-2000	158
1-56592-457-6	Oracle PL/SQL Leksykon podręczny	Leksykon języka PL/SQL systemu Oracle.	Feuerstein, Steven, Bill Pribyl oraz Chip Davis	01-KWI-1999	94

book_copies

ISBN	barcode_id
1-56592-335-9	100000001
1-56592-335-9	100000002
0-14071-483-9	100000015
0-14071-483-9	100000016
1-56592-457-6	100000030
1-56592-457-6	100000022
1-56592-457-6	100000020

Rysunek 3.2. Przykład danych w postaci relacyjnej, rozbitych na wiersze i kolumny

```
INSERT INTO books (isbn, title, author)
VALUES ('0-596-00180-0', 'Oracle PL/SQL. Wprowadzenie',
   'Bill Pribyl, Steven Feuerstein');
```

W związku z powyższym Czytelnik z pewnością zastanawia się nad sensem tworzenia programu w języku PL/SQL.

Uzasadnienie projektu

Należy założyć, że istnieje konieczność zapewnienia stosowania dwóch sposobów dodawania książek do katalogu: po pierwsze w sposób interaktywny, poprzez ręczne wpisanie danych, po drugie w sposób automatyczny, pobierając informacje o książce z innej bazy danych. Nasuwa się zatem pytanie, czy w takiej sytuacji należy skopiować instrukcje `INSERT` do dwóch programów. Jednakże później może pojawić się potrzeba napisania trzeciego programu, który dodawałby informacje o książkach, przykładowo, odczytując je z płyty CD-ROM. A zatem należałoby uwzględnić kolejne kopiowanie instrukcji `INSERT`. Zakładając jednak, że zaistnieje potrzeba zmiany struktury tabel, trzeba by wówczas wszystkie dotąd utworzone programy oddzielnie uaktualnić.

Istnieje kilka powodów, dla których instrukcje `INSERT` warto umieścić w programie PL/SQL. Poniżej wymieniono najważniejsze z nich:

- Umożliwia to ograniczenie, a nawet wyeliminowanie męczącej pracy związanej z aktualizacją oprogramowania po zmianie struktury bazy danych, podczas wykonywania której istnieje możliwość popełniania błędów.

- Pozwala to na zwiększenie wydajności działania serwera bazy danych.
- Pozwala to umieścić rozwiązanie problemu programistycznego w tylko jednym miejscu.

Tworzenie poprawnych instrukcji w języku SQL może wymagać interpretowania oraz zapisywania w formie kodu wielu skomplikowanych reguł zarządzania. Ewentualna konieczność dokonania zmian we wszystkich napisanych programach oddzielnie oznacza bezzasadne marnotrawstwo czasu i wysiłku. Ogólna zasada mówi, że:

> *Instrukcje SQL należy umieszczać w jednym, możliwym do wielokrotnego wykorzystania, programie napisanym w języku PL/SQL, a nie kopiować je w wielu aplikacjach*.

Nawet jeśli Czytelnik jest jedynym programistą w swoim miejscu pracy, powinien trzymać się tej zasady. Nie ogranicza się ona zresztą tylko do wykorzystania języka SQL — wszelkie zadania programistyczne powinno się zapisywać tylko raz, a potem w razie konieczności jedynie wykonywać odpowiedni program. Przez definiowanie czynności wykonywanych przez każdą jednostkę programu, którą można wielokrotnie wywoływać, można utworzyć własny *interfejs tworzenia oprogramowania* (*application programming interface, API*).

Tak prosty, jak to możliwe, ale nie prostszy

Prawdziwy bibliotekarz uznałby niniejszy model za banalnie prosty, nawet nie biorąc pod uwagę nie uwzględnionych w nim kwestii związanych z czytelnikami, operacjami wypożyczenia/oddawania i kupowania książek. W rzeczywistości funkcjonowanie biblioteki jest znacznie bardziej skomplikowane:

- Poza przechowywaniem książek biblioteki magazynują także gazety i czasopisma, nagrania muzyczne i kasety wideo.
- Wiele dzieł, na przykład starsze książki, nie posiada numeru ISBN (*International Standard Book Number*), co ogranicza jego zastosowanie jako jednoznacznego identyfikatora.
- Dzieła mogą mieć różne tytuły i wielu autorów.
- Należy przechowywać o wiele więcej informacji: o działach, wydawcach, ilustratorach, wydaniach, dziełach wielotomowych i pochodnych.
- Biblioteki zazwyczaj udostępniają sobie nawzajem swoje katalogi drogą elektronicznej wymiany.

Podobnych przykładów jest więcej, dlatego należy wyjaśnić powody uwzględnienia w niniejszej książce tak prostego przykładu.

Wykonanie bazy danych, której schemat pokazano na rysunku 3.1, w zasadzie jest zbyt skomplikowanym zadaniem dla zupełnie początkujących programistów. Relacja *jeden do wielu* stanowi podstawę większości aspektów projektowania baz danych (tyle że w realnych sytuacjach chodzi o wiele takich relacji) oraz programowania w języku PL/SQL. W dalszej części niniejszej książki zostanie opisany sposób rozszerzania funkcjonalności projektowanej bazy danych, a także sposoby wykorzystania PL/SQL w bardziej „realnych" zadaniach.

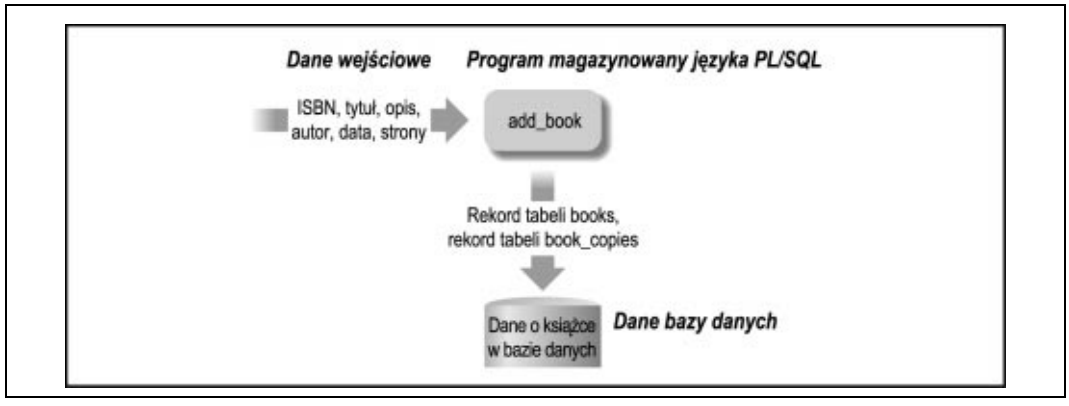
Innym powodem wprowadzenia pewnych uproszczeń jest fakt, że nauka nowego materiału przychodzi najłatwiej, jeśli można studiować jedno zagadnienie w jednym czasie. W przeciwnym razie nauka przychodzi dużo trudniej.

W niniejszym rozdziale przedstawiono informacje o sposobach lokalizowania kodu SQL (stosowanie kodu w jednym miejscu) za pomocą *obwolut tabel* (*table wrappers*), które są programami odpowiedzialnymi za wszelkie operacje zmiany zawartości każdej z tabel bazy danych.

Identyfikacja danych wejściowych, procesu przetwarzania oraz danych wyjściowych

Zapewne większość programistów, niezależnie od stopnia zaawansowania, odczuwa niepewność w sytuacji, gdy ma napisać program od podstaw. Z reguły w takiej sytuacji można rozpocząć pracę od przerabiania już istniejących lub przykładowych programów. W tym przypadku zostanie przedstawiony sposób tworzenia programu od samego początku, ale prawdopodobnie Czytelnik rzadko będzie do tego zmuszony.

Godnym polecenia zwyczajem jest rozpoczęcie tworzenia programu od narysowania schematu. Rysunek 3.3 przedstawia możliwą reprezentację graficzną omawianego programu oraz jego danych wejściowych i wyjściowych.



Rysunek 3.3. Schemat programu służącego do dodawania książek do bazy danych

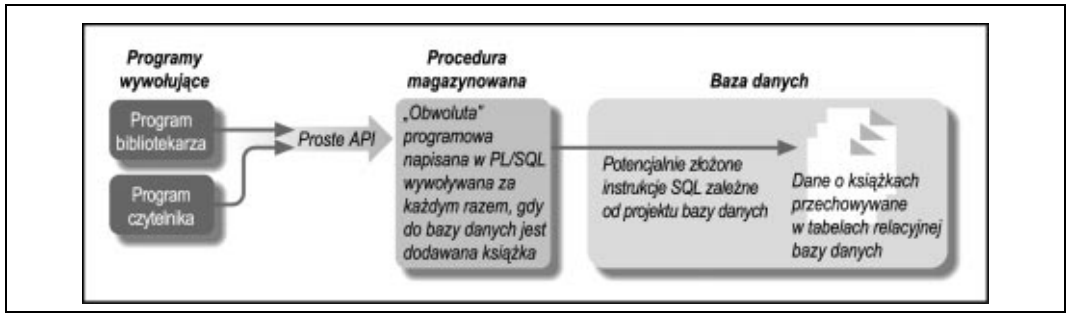
W przeciwieństwie do wcześniej prezentowanego diagramu związków i encji, powyższy diagram nie został skonstruowany zgodnie z jakimiś zasadami projektowania. Najważniejszą rzeczą jest przedstawienie najbardziej znaczących elementów. Z powyższego rysunku wynika, że trzeba skonstruować program zdolny do pobierania pewnych informacji, a następnie do umieszczania ich w tabelach bazy danych. Choć zapewne nie jest to oczywiste, uwzględniono możliwość poszerzenia funkcjonalności aplikacji w przyszłości.

> *Ogólnie rzecz biorąc, jednostka programu powinna służyć do wykonania jednego, konkretnego zadania i należy zapewnić prawidłowe działanie tej jednostki w swoim zakresie. Wskazane jest, aby miała on niewielki rozmiar — ułatwi to jej zapis i późniejsze modyfikacje.*

Trzeba jeszcze znaleźć odpowiedź na pytanie, jakiej konstrukcji języka PL/SQL należałoby użyć w celu zdefiniowania operacji dodawania książki `add_book`. Z uwagi na fakt, że program nie będzie zwracał żadnej wartości, najbardziej rozsądnym rozwiązaniem jest zastosowanie procedury.

Zastosowanie procedury magazynowanej w celu dodania książki

W tym podrozdziale zostanie przedstawiony sposób napisania procedury, która ma wstawiać informacje na temat danej książki do bazy danych. Procedura taka może być nazwana *obwolutą*. Można też powiedzieć, używając terminologii pochodzącej z języka greckiego, że będą wykorzystywane *abstrakcje* (*abstraction*), procesy *enkapsulacji* (*encapsulation*) oraz *ukrywania informacji* (*information hiding*) — definicje wymienionych pojęć znajdują się w Słowniczku. Na rysunku 3.4 przedstawiono więcej szczegółów dotyczących takiego rozwiązania.



Rysunek 3.4. Jedynie z poziomu programu PL/SQL będzie możliwe dodawanie informacji o książce do bazy danych

Użytkownicy gotowego programu nie muszą wiedzieć na jego temat zbyt wiele, jednak twórca danej aplikacji musi z pewnością posiadać gruntowną wiedzę dotyczącą struktury bazy danych. Dlatego też pierwszą rzeczą, jaką należy wykonać, jest określenie tabel i ich kolumn związanych z procesem wykonania reguły zarządzania (w tym przypadku – dodania książki). Trzeba odpowiedzieć sobie na pytanie, jakie informacje są potrzebne do manipulowania zawartością tych tabel.

Analiza projektu

W celu określenia istotnych dla wykonania planowanej operacji tabel i kolumn trzeba przyjrzeć się projektowi bazy danych (przedstawionemu we wcześniejszej części niniejszego rozdziału). Listę tych kolumn można z łatwością utworzyć za pomocą instrukcji `DESCRIBE`, dostępnej w SQL*Plus (w formie skróconej `DESC`):

```
SQL> DESC ksiazki
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------
 ISBN                                      NOT NULL VARCHAR2(13)
 TITLE                                              VARCHAR2(200)
 SUMMARY                                            VARCHAR2(2000)
 AUTHOR                                             VARCHAR2(200)
 DATE_PUBLISHED                                     DATE
 PAGE_COUNT                                         NUMBER

SQL> DESC egzemplarze_ksiazki
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------
 BARCODE_ID                                NOT NULL VARCHAR2(100)
 ISBN                                               VARCHAR2(13)
```

Po przeanalizowaniu listy kolumn można przyjąć, że większość zawartych w nich informacji powinna być znana osobie wykonującej katalogowanie i dlatego nie trzeba niczego syntetyzować, sprawdzać lub obliczać. Procedura będzie więc bardzo prosta.

Podstawowym problemem do rozwiązania jest podjęcie decyzji, czy użytkownik będzie wstawiał dane do obydwóch tabel jednocześnie, czy też oddzielnie. Na tym etapie pracy odpowiedź na to pytanie jest jeszcze nieznana. Najprawdopodobniej w bibliotekach odbywa się to w ten sposób, że informacje o książkach są dodawane do bazy danych w momencie pojawienia się pierwszego egzemplarza. Dlatego też podczas dodawania danej książki po raz pierwszy wszystkie informacje wymagane dla wypełnienia obydwóch tabel są od razu znane: od numeru ISBN po identyfikator kodu kreskowego. Jednak istnieje także konieczność katalogowania nowych egzemplarzy książki już znajdującej się w zbiorach biblioteki — tę potrzebę również należy uwzględnić podczas planowania programu.

Rozpoczynając opracowywanie nowego projektu wielu programistów zapisuje tzw. *pseudokod* w postaci zwykłych zdań, które opisują z grubsza działanie programu. W tym przypadku można by napisać:

```
Sprawdź, czy dane wejściowe są prawidłowe.
Wstaw nowy rekord do tabeli "books".
Wstaw nowy rekord do tabeli "book_copies".
```

Kolejnym etapem będzie przedstawienie składni konstrukcji językowych potrzebnych do utworzenia procedury. Następnie zapisany pseudokod zostanie zamieniony na konkretne instrukcje.

Składnia tworzenia procedury

Poniżej znajduje się opis różnych części składowych procedury. Zazwyczaj procedury tworzy się za pomocą instrukcji o następującej składni:

```
CREATE [ OR REPLACE ] PROCEDURE nazwa_procedury
   (parametr1 TRYB TYP_DANYCH [ DEFAULT wyrażenie ],
    parametr2 TRYB TYP_DANYCH [ DEFAULT wyrażenie ],
    ...)
AS
[   zmienna1 TYP_DANYCH;
    zmienna2 TYP_DANYCH;
    ... ]
BEGIN
   instrukcje_wykonawcze
[ EXCEPTION
   WHEN nazwa_wyjątku
   THEN
      instrukcje_wykonawcze ]
END;
/
```

Powyższy wzorzec zawiera kombinację słów kluczowych języka PL/SQL (pisane dużymi literami i nie kursywą) oraz wyrażeń do zastąpienia (pisane kursywą) w kodzie tworzonego programu.

`CREATE [ OR REPLACE ]`

Jest to szczególna instrukcja SQL, służąca do utworzenia procedury. Fraza `OR REPLACE` (*lub zastąp*) jest opcjonalna i pozwala na uniknięcie konieczności usuwania już istniejącej procedury w przypadku tworzenia jej nowej wersji. Zastosowanie wyrażenia `OR REPLACE` zachowuje także wszelkie synonimy (*synonyms*) oraz *granty* (*grants*), jakie zostały uprzednio utworzone, a są zależne od działania procedury. Jest to duża zaleta.

`PROCEDURE` *`nazwa_procedury`*

W sekcji nagłówka podaje rodzaj tworzonej jednostki programowej (w tym wypadku — procedura) oraz nadaje się jej nazwę.

`parametr1 TRYB TYP_DANYCH` `[ DEFAULT` *`wyrażenie`* `]`

Aby umożliwić użytkownikowi podawanie parametrów wywołania procedury, należy utworzyć listę definicji parametrów oddzielonych przecinkami, a całą listę objąć w nawiasy. *`TRYB`* może mieć wartość: `IN`, `OUT` lub `IN OUT`. Poniżej znajduje się opis wszystkich opcji.

`IN`

Słowo kluczowe oznaczające tryb tylko do odczytu. Wywołujący procedurę podaje wartość parametru, a PL/SQL nie pozwala na jej zmianę wewnątrz programu.

`OUT`

Słowo kluczowe oznaczające tryb tylko do zapisu. Oznacza to, że procedura nadaje parametrowi pewną wartość, która jest odczytywana przez program wywołujący. Podanie jakiejkolwiek wartości takiego parametru podczas wywołania procedury jest ignorowane.

`IN OUT`

Słowo kluczowe oznaczające tryb do odczytu lub zapisu. Słowo to jest używane w sytuacji, gdy wartość zmiennej, przekazywanej do procedury jako parametr, ma być zarówno odczytywana, jak i zmieniana, a następnie zwracana do programu wywołującego.

`TYP_DANYCH`

Znaczenie tego parametru przedstawiono w rozdziale 2. — dopuszczalnymi wartościami są na przykład: `NUMBER`, `INTEGER`, `VARCHAR2`, `DATE`. Jedyna różnica polega na tym, że w tym miejscu należy podać jedynie *rodzaj* typu, bez jego dokładnej specyfikacji. Innymi słowy, należy użyć `VARCHAR2` zamiast `VARCHAR2(30)` i `NUMBER` zamiast `NUMBER(10,2)`.

`DEFAULT` *`wyrażenie`*

Pozwala na przypisanie parametrowi wartości domyślnej w przypadku, gdy wywołanie procedury jej nie określa. Można także użyć symbolu „`:=`” (dwukropek i znak równości) zamiast słowa kluczowego `DEFAULT`.

`AS`

Słowo kluczowe `AS` oddziela nagłówek od reszty jednostki programowej. Opcjonalnie można użyć słowa kluczowego `IS`, które jest równoważne `AS`.

`BEGIN..END`

Słowa `BEGIN` i `END` oddzielają zwykłe, czyli wykonawcze instrukcje od reszty programu.

`EXCEPTION`

Oznacza początek kodu obsługi wyjątku — tej części programu, która jest wykonywana tylko w przypadku przechwycenia wyjątku w sekcji wykonawczej. Wszystkie elementy umieszczone po słowie kluczowym `EXCEPTION` i przed instrukcją `END` są częścią obsługi wyjątku.

`WHEN nazwa_wyjątku THEN instrukcje_wykonawcze`

Możliwym do wystąpienia sytuacjom błędnego działania programu zazwyczaj są nadawane nazwy — albo przez system Oracle, albo przez programistę. Dzięki temu można w kodzie programu zapisać instrukcje „wychwytujące" te sytuacje i umożliwić odpowiednią na nie reakcję, którą jest wykonanie pewnego fragmentu kodu. Jeśli taka nazwa jest nieznana lub w celu wychwytywania błędów, które nie zostały nazwane, można posłużyć się słowem `OTHERS`, które dotyczy wszystkich sytuacji wyjątkowych. W zapisie wyglądałoby to następująco: `WHEN OTHERS THEN...`

Pewne elementy kodu procedury są opcjonalne. Najprostsza, możliwa do utworzenia procedura posiada następującą formę:

```
CREATE PROCEDURE nie_rob_nic AS
BEGIN
   NULL;
END;
```

Z powyższego wynika, że parametry, zmienne i kod obsługi wyjątków są opcjonalne. Słowo kluczowe `NULL` zostało tutaj zastosowane jako instrukcja wykonawcza. Oznacza ona tyle, co „nie rób nic".

Procedura add_book

Korzystając z potrzebnych elementów podanego wcześniej wzorca można zapisać pseudokod w formie prawdziwego kodu:

```
CREATE OR REPLACE PROCEDURE add_book (isbn_in IN VARCHAR2,
   barcode_id_in IN VARCHAR2, title_in IN VARCHAR2, author_in IN VARCHAR2,
   page_count_in IN NUMBER, summary_in IN VARCHAR2 DEFAULT NULL,
   date_published_in IN DATE DEFAULT NULL)
AS
BEGIN
   /* sprawdzenie poprawności danych wejściowych */

   IF isbn_in IS NULL
   THEN
      RAISE VALUE_ERROR;
   END IF;

   /* wstawienie rekordu do tabeli "books" */

   INSERT INTO books (isbn, title, summary, author, date_published,
page_count)
   VALUES (isbn_in, title_in, summary_in, author_in, date_published_in,
      page_count_in);

   /* jeśli to konieczne, wstawienie rekordu do tabeli "book_copies" */

   IF barcode_id_in IS NOT NULL
   THEN
      INSERT INTO book_copies (isbn, barcode_id)
      VALUES (isbn_in, barcode_id_in);
   END IF;

END add_book;
/
```

Poniżej znajduje się opis działania utworzonej procedury.

Nazwa procedury i kolejność parametrów. Nazwą procedury jest wyrażenie czasownikowe, które opisuje działanie tej procedury. Podano tu także parametry wejściowe — na każdą z kolumn tabel, do których mają być wprowadzane dane, przypada jeden parametr. W celu wyeksponowania najważniejszych parametrów (`isbn_in` czy `barcode_id_in`) oraz w celu umieszczenia parametrów o wartościach domyślnych na końcu, zmieniono nieco ich kolejność w odniesieniu do kolejności kolumn w tabelach.

Nazwy parametrów. Warto przyjąć pewną konwencję nazewnictwa w programowaniu i do nazw parametrów dodać się nazwę ich trybu (`IN`, `OUT` lub `IN OUT`). Ponieważ wszystkie parametry procedury `add_book` są parametrami w trybie `IN`, ich nazwy zawierają końcówkę `_in`. Taka konwencja nadawania nazw nie jest obowiązkowa, ale jest pomocna w unikaniu konfliktów z nazwami kolumn podczas używania instrukcji SQL. Gdyby pozostały one identyczne, otrzymana instrukcja miałaby następującą postać:

```
INSERT INTO book_copies (barcode_id, isbn)
VALUES (barcode_id, isbn);
```

W takim przypadku odróżnianie nazwy kolumn od nazw zmiennych podczas odczytywania kodu mogłoby okazać się problematyczne nawet dla doświadczonego programisty. Jednakże sama instrukcja zostałaby wykonana prawidłowo, gdyż PL/SQL zinterpretowałby wszystko następująco:

```
INSERT INTO book_copies (barcode_id, isbn)     /* nazwy kolumn */
VALUES (barcode_id, isbn);                     /* zmienne PL/SQL */
```

Jednakże instrukcja:

```
UPDATE ksiazki
   SET summary = summary  /* Błąd! */
 WHERE isbn = isbn;       /* Nie wolno tak robić! */
```

nie zostanie wykonana prawidłowo. PL/SQL zinterpretuje każde wystąpienie `summary` oraz `isbn` jako nazwy kolumn.

Weryfikacja danych wejściowych. Zapis pierwszej linii zaprezentowanego wcześniej pseudokodu mówił: „Sprawdź, czy dane wejściowe są prawidłowe”. Konieczne jest jednak podjęcie jeszcze kilku ustaleń. Przykładowo, można założyć, że użytkownicy systemu będą domagać się minimalnych wymagań ze strony procedury w momencie jej wywoływania. Może to oznaczać, że jedynym absolutnie wymaganym parametrem tworzonej procedury jest ISBN. Stąd też sekcja weryfikacji danych wejściowych została zapisana jako:

```
IF isbn_in IS NULL
THEN
   RAISE VALUE_ERROR;
END IF;
```

Bez dokładniejszej znajomości powyższych identyfikatorów nie jest możliwe przeprowadzenie bardziej wymyślnej weryfikacji poprawności danych wejściowych. Przedstawiony wyżej fragment kodu oznacza, że brak ISBN powoduje zatrzymanie wykonywania programu.

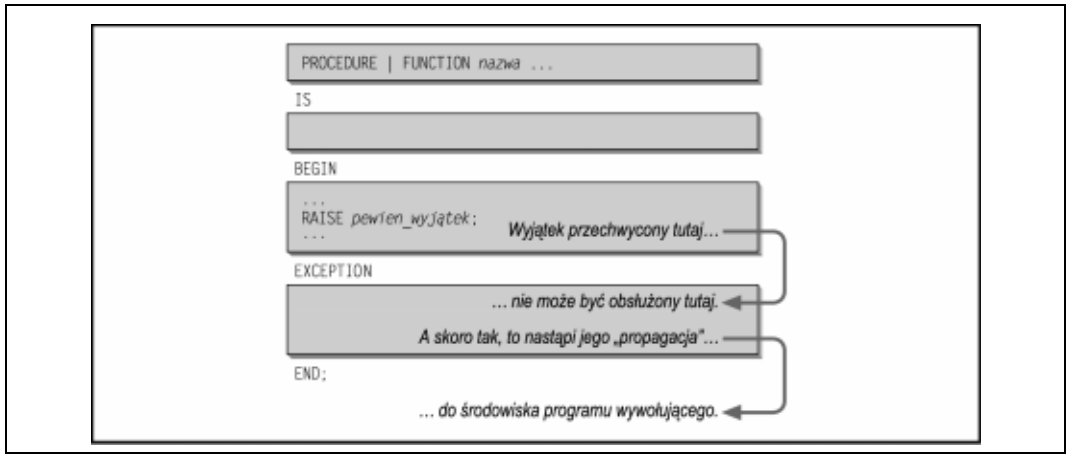
W dalszej części kodu jest także wykonywane sprawdzenie kodu kreskowego. Program został skonstruowany w ten sposób, aby uniknąć wykonania niepoprawnej instrukcji `INSERT`, co spowodowałoby błąd.

Obsługa sytuacji wyjątkowych

W rozdziale 2. wyjaśniono, że w razie wystąpienia błędu PL/SQL zatrzymuje wykonywanie programu za pomocą mechanizmu zwanego *przechwytywaniem wyjątku* (*raising an exception*). Jak wynika z rysunku 3.5, instrukcja:

```
RAISE VALUE_ERROR
```

zatrzymuje wykonywanie instrukcji sekcji wykonawczej i przekazuje sterowanie do kodu obsługi wyjątku. Jeśli taki kod nie istnieje, wyjątek zwraca pewien błąd do programu wywołującego, a ten reaguje w określony przez programistę sposób.



Rysunek 3.5. Prosty przykład obsługi wyjątku

Wyjątek `VALUE_EXCEPTION` należy do grupy wyjątków wewnętrznych, które w pewnych sytuacjach są przechwytywane przez system Oracle. Program wywołujący procedurę `add_book` powinien umożliwiać obsługę wszelkich możliwych do przewidzenia wyjątków i na podstawie informacji dotyczących napotkanego błędu zdecydować o dalszym funkcjonowaniu programu. W tym przypadku pożądanym działaniem byłoby przerwanie procesu dodawania do bazy danych niekompletnych informacji. Najlepiej byłoby, gdyby program wywołujący zwracał użytkownikowi informację o błędzie.

Być może Czytelnik zastanawia się na celowością podejmowania takich działań zamiast, przykładowo, umożliwienia samej procedurze `add_book` wyświetlania komunikatu o napotkanym błędzie. Celowość użycia mechanizmu wyjątków staje się jasna po uświadomieniu sobie, że przekazanie obsługi sytuacji wyjątkowych do procedury ograniczyłoby w znacznym stopniu możliwości jej ponownego wykorzystania. Jeśli procedura `add_book` obsługuje wyjątek i wyświetla komunikat o błędzie, niemożliwe jest wykorzystanie jej przez program, dla którego działania takie komunikaty są niewskazane. Przykładowo, można założyć istnienie programu, który odczytuje informacje o książkach z pliku i jednorazowo wprowadza do bazy danych informacje dotyczące tysięcy książek.

Program ten ma wywoływać procedurę `add_book` dla każdej kolejnej pozycji i w razie wystąpienia błędu zapamiętać informację o jego przyczynie, nie przerywając jednak swojego działania, aż do zakończenia wprowadzania danych dotyczących ostatniej książki. Dopiero wówczas można by przedstawić użytkownikowi podsumowanie informacji o napotkanych problemach. Wykorzystanie procedury `add_book` w taki sposób jest możliwe jedynie poprzez propagację wyjątków na zewnątrz — do programu wywołującego.

Uogólniając, jeśli istnieje prawdopodobieństwo napotkania przez program błędu, którego nie da się naprawić w prosty sposób, należy zapewnić możliwość przechwycenia wyjątku. W dalszej części niniejszej książki opisano wyjątki, które należy brać pod uwagę.

Jeżeli weryfikacja poprawności danych wejściowych przebiega pomyślnie, program kontynuuje swoje działanie wykonując instrukcje `INSERT` języka SQL. Nie ma potrzeby zwracania jakichkolwiek danych do programu wywołującego. Jednak gdyby zaszła taka konieczność, można uwzględnić jedną z dwóch możliwości: zastosowanie funkcji zamiast procedury lub wykorzystanie parametrów w trybie `OUT`. Więcej informacji Czytelnik znajdzie w dalszej części niniejszego rozdziału.

Użycie procedury w celu dodania książki do katalogu

Najłatwiejszym sposobem wywołania procedury z poziomu kodu PL/SQL jest napisanie jej nazwy wraz z potrzebnymi argumentami, umieszczonymi w nawiasach i oddzielonymi przecinkami:

```
BEGIN
   nazwa_procedury (argument1, argument2, ...);
END;
```

Przykładowo, aby dodać do katalogu nową książkę, należy napisać:

```
BEGIN
   add_book('1-56592-335-9',
      '100000001',
      'Programowanie w Oracle PL/SQL',
      'Feuerstein, Steven, Bill Pribyl',
      987,
      'Kompendium informacji o języku PL/SQL, '
         || 'wraz z przykładami i poradami na temat programowania.',
      TO_DATE('01-WRZ-1997','DD-MON-YYYY'));
END;
/
```

Takie wywołanie procedury spowoduje wstawienie po jednym rekordzie do tabel `books` oraz `book_copies`. W powyższym przykładzie argumentami są tak zwane wyrażenia literalne, a PL/SQL przekazuje te wartości do programu jako parametry wejściowe[1], zgodnie z ich kolejnością. Oznacza to, że zazwyczaj wartości trzeba podawać w takiej samej kolejności, w jakiej występują parametry w programie wywoływanym. Informacje te przedstawiono w poniższej tabeli:

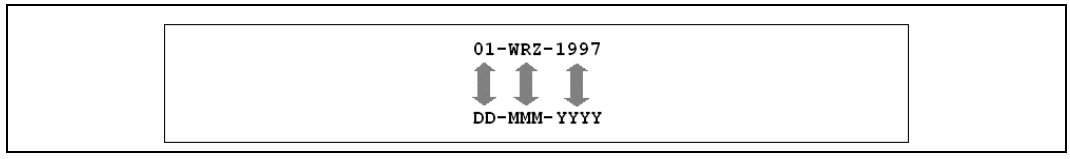
[1] Wyrażenie przekazywane przez program wywołujący nosi nazwę *argumentu* lub *parametru aktualnego*, podczas gdy zmienne zdefiniowane w nagłówku programu wywołującego są nazywane *parametrami formalnymi* lub po prostu *parametrami*. W tym kontekście „formalny” oznacza „formujący” postać zmiennej.

Pozycja	Nazwa parametru	Typ danych parametru	Wartość użyta w przykładowym wywołaniu
1	isbn_in	**VARCHAR2**	'1-56592-335-9'
2	barcode_id_in	**VARCHAR2**	'100000001'
3	title_in	**VARCHAR2**	'Programowanie w Oracle PL/SQL'
4	author_in	**VARCHAR2**	'Feuerstein, Steven, Bill Pribyl'
5	page_count_in	**NUMBER**	987
6	summary_in	**VARCHAR2**	'Kompendium informacji o języku PL/SQL wraz z przykładami i poradami na temat programowania.'
7	date_published_in	**DATE**	TO_DATE('01-WRZ-1997','DD-MON-YYYY')

Warto zwrócić uwagę, że podane wartości są zgodne z typami danych, jakich wymaga procedura. Oznacza to, że każdemu parametrowi typu VARCHAR2 odpowiada ciąg znakowy, parametrowi typu NUMBER odpowiada seria cyfr. Czytelnik zapewne zauważył, że parametrowi typu DATE odpowiada wartość o dość dziwnym wyglądzie.

Praca z typem DATE

Parametr date_published_in wymaga podania wartości typu DATE systemu Oracle. W rzeczywistości jest to seria bitów o skomplikowanym formacie wewnętrznym — z pewnością nie jest to zwykle używana kombinacja roku, miesiąca i dnia. Często stosowaną metodą otrzymania takiej wartości jest wykorzystanie wewnętrznej funkcji systemu Oracle o nazwie TO_DATE. Podczas wywoływania tej funkcji należy podać datę, zapisaną w formie ciągu znakowego (w tym przypadku — '01-WRZ-1997'), oraz dodatkowy element, zwany *maską formatu* (*format mask*) — 'DD-MON-YYYY'. Funkcja TO_DATE próbuje dopasować do siebie poszczególne elementy wyspecyfikowanego ciągu znaków z odpowiednimi elementami maski formatującej. W przypadku pomyślnego wykonania funkcja TO_DATE zwraca wartość binarną, którą system Oracle rozpoznaje jako wartość typu DATE. W przeciwnym przypadku następuje przechwycenie wyjątku.



Wróćmy do omawianego przykładu: funkcja TO_DATE przekazuje wspomnianą wartość binarną jako argument do procedury add_book. Dla tej procedury sposób utworzenia wartości typu DATE jest bez znaczenia — bezpośrednie podawanie wartości literałów jest równie dobrym rozwiązaniem, co uwzględnienie wyniku działania funkcji, takiej jak na przykład TO_DATE.

Zastosowanie funkcji TO_DATE nie zawsze jest konieczne, ponieważ system Oracle automatycznie podejmuje próbę konwersji ciągu znakowego na datę. W sytuacji takiej trzeba opierać się na domyślnej masce formatu systemu Oracle, która może zostać zmieniona przez administratora bazy danych. Zwiększa to prawdopodobieństwo wystąpienia pewnych problemów, jak na przykład interpretacja wartości roku (jaki rok jest oznaczony wartością '00'?). Ogólnie można powiedzieć, że w celu konwersji na format daty systemu Oracle należy skorzystać z funkcji TO_DATE.

Powyższy fragment kodu zawiera argumenty podane w formie *literałów* (*literals*), to znaczy z zastosowaniem konkretnych wartości. Jest to dobry sposób w przypadku przeprowadzania testów, jednakże podczas programowania w języku PL/SQL zazwyczaj używa się zmiennych w wywołaniach. Wartości zmiennych są zazwyczaj ustalane nie poprzez bezpośredni zapis w kodzie programu, ale na przykład przez użytkownika korzystającego z klawiatury i ekranu, wyświetlającego odpowiednie informacje.

Argumenty opcjonalne

Warto się zastanowić nad sposobem wywoływania procedury w przypadku posiadania niepełnej informacji dotyczącej dodawanej książki — na przykład w przypadku braku daty wydania lub opisu. Przykładowo, w takim przypadku można podać wartości `NULL` dla tych parametrów i sprawdzić, czy nie spowoduje to problemów z wykonywaniem programu:[2]

```
add_book('1-56592-335-9', '100000001', 'Programowanie w Oracle PL/SQL',
   'Feuerstein, Steven, Bill Pribyl', 987, NULL, NULL);
```

Jest to jedna z możliwości. Można także w ogóle nie podawać wartości argumentów i oprzeć się na wartościach domyślnych. W tym przypadku będzie to wywołanie poprawne, gdyż w sekcji nagłówkowej procedury dla dwóch ostatnich parametrów zdefiniowano wartości domyślne:

```
...summary_in IN VARCHAR2 DEFAULT NULL,
date_published_in IN DATE DEFAULT NULL);
```

Wartości `NULL` w tym kontekście są w pełni poprawnymi wartościami.

Korzystając z wartości domyślnych, można przedstawione powyżej wywołanie uprościć i pominąć ostatnie dwa argumenty:

```
add_book('1-56592-335-9', '100000001', 'Programowanie w Oracle PL/SQL',
   'Feuerstein, Steven, Bill Pribyl', 987);
```

W tym przypadku mechanizm odpowiedzialny za wykonanie kodu PL/SQL w miejsce brakujących argumentów podstawi wartości domyślne i wykonanie procedury odbędzie się identycznie, jak we wcześniejszym przypadku.

Możliwość pomijania argumentów dla parametrów posiadających wartość domyślną jest niezwykle przydatną cechą języka PL/SQL, ponieważ umożliwia prawidłowe działanie jednostki programowej nawet bez podania tych argumentów.

Ponadto mechanizm ten może w ogromnym stopniu zredukować negatywny wpływ ewentualnych, przyszłych modyfikacji. Przykładowo, jeśli zaistnieje potrzeba dodania do programu pewnych parametrów, to dzięki zdefiniowaniu ich wartości domyślnych nie będzie trzeba przeglądać całego kodu, aby zmienić każde występujące w nim wywołanie.

[2] Ze względu na oszczędność miejsca w niniejszej książce nie zawsze jest przedstawiany pełny kod programu. W tym przypadku należałoby dodać instrukcję EXECUTE lub objąć wywołanie słowami kluczowymi BEGIN i END oraz dodać końcowe "/" w celu uruchomienia tej procedury z poziomu SQL*Plus.

Warto też wiedzieć, że wywołując procedurę można jednocześnie pomijać parametry posiadające wartości domyślne oraz podawać wartości NULL:

```
add_book('1-56592-335-9', '100000001', 'Programowanie w Oracle PL/SQL', NULL,
987);
```

Czytelnik być może ma pewne problemy z zapamiętaniem, która wartość odpowiada któremu argumentowi, ale jest to rzecz normalna w przypadku *notacji pozycyjnej* (*positional notation*), co oznacza konieczność podawania argumentów w takiej samej kolejności, w jakiej zapisano dane parametry w kodzie programu.

Notacja imienna

Rozwiązaniem tych problemów może być zastosowanie innego, bardzo użytecznego mechanizmu — *notacji imiennej* (*named notation*). Sposób działania notacji imiennej najlepiej można wyjaśnić na przykładzie. W przypadku poniższego wywołania procedury add_book wykorzystano właśnie ten mechanizm:

```
add_book(isbn_in => '1-56592-335-9',
   title_in => 'Programowanie w Oracle PL/SQL',
   summary_in => 'Kompendium informacji o języku PL/SQL, ' ||
      'wraz z przykładami i poradami na temat programowania.',
   author_in => 'Feuerstein, Steven, Bill Pribyl',
   date_published_in => NULL,
   page_count_in => 987,
   barcode_id_in => '100000001');
```

Z powyższego wynika, że dla każdego argumentu podaje się nazwę parametru zdefiniowanego w procedurze add_book, symbol => oraz wartość danego parametru. Zaletą tej metody jest to, że poszczególne argumenty można podawać w dowolnej kolejności — nie trzeba pamiętać kolejności, w jakiej zdefiniowano parametry w procedurze.

Wybór metody całkowicie zależy od programisty — dla kompilatora jest to bez znaczenia. Argumentem przemawiającym za stosowaniem notacji imiennej może być fakt, że w ten sposób kod staje się dużo bardziej przejrzysty. Znaczenie wyrażenia page_count => 987 nie pozostawia wątpliwości, a na pewno nie jest tak w przypadku podania samej wartości 987. Można także, oczywiście, pomijać parametry opcjonalne (domyślne) — w tym przypadku może to być, przykładowo, date_published_in:

```
add_book(isbn_in => '1-56592-335-9',
   title_in => 'Programowanie w Oracle PL/SQL',
   summary_in => 'Kompendium informacji o języku PL/SQL, ' ||
      'wraz z przykładami i poradami na temat programowania.',
   author_in => 'Feuerstein, Steven, Bill Pribyl',
   page_count_in => 987,
   barcode__is_in => '100000001');
```

Poza koniecznością pisania większej ilości kodu istnieje także pewna niedogodność, wynikająca ze stosowania tej notacji. W razie potrzeby zmiany nazwy parametru konieczne jest uaktualnienie nie tylko samej procedury, ale także każdego jej wywołania w notacjiimiennej. Z drugiej jednak strony zmiana nazwy parametru jest raczej rzadko wykonywaną operacją.

Istnieje także możliwość wykorzystania obydwóch notacji w jednym wywołaniu. Należy pamiętać jedynie o paru zasadach. Należy zacząć od notacji pozycyjnej, a po przejściu do notacji imiennej trzeba przy niej pozostać do końca. Oto przykład:

```
add_book('1-56592-335-9', '100000001',
   'Programowanie w Oracle PL/SQL',
   summary_in => NULL, author_in => 'Feuerstein, Steven, Bill Pribyl',
   page_count_in => 987);
```

Używanie notacji imiennej

W razie konieczności dokonania wyboru należy stosować notację wymieniającą wszędzie tam, gdzie znaczenie poszczególnych argumentów nie jest oczywiste. Przykładowo:

```
uaktualnij_moj_profil(ulub_ksiazka_isbn => '1-56592-335-9');
```

Notacji pozycyjnej należy używać w przypadku często używanych programów użytkowych, które posiadają jeden lub dwa parametry, a znaczenie tych parametrów jest oczywiste:

```
DBMS_OUTPUT.PUT_LINE('Witaj Muddah.');
```

Zastosowanie obydwóch notacji może okazać się przydatne w sytuacji, gdy na przykład pierwszy argument jest oczywisty, zaś kolejne nie:

```
moje_put_line('Witaj Fadduh.', linie_do_pominiecia => 2);
```

W tym podrozdziale przedstawiono przykład konstruowania i wykorzystywania procedury magazynowanej języka PL/SQL, która wykonuje pojedyncze zadanie. Zadanie to polega na dodaniu do bazy danych informacji o książce identyfikowanej za pomocą odpowiedniego kodu kreskowego. Kolejne zadanie polega na określeniu liczby egzemplarzy danej książki. Potrzeba pobierania informacji o takiej pojedynczej wartości stanowi idealną sposobność wykorzystania funkcji języka PL/SQL, która ze swej definicji zwraca jakąś wartość (lub kończy działanie nie obsłużonym wyjątkiem).

Pobieranie informacji o liczbie książek za pomocą funkcji

Przed podjęciem próby zapisania kodu funkcji trzeba najpierw zapoznać się z ogólną składnią tworzenia funkcji. Funkcje stanowią drugi typ programów magazynowanych.

Składnia tworzenia funkcji

Poniżej przedstawiono wzorzec definiowania funkcji. Elementy, które oznaczono za pomocą czcionki pogrubionej, mogą być jeszcze Czytelnikowi nieznane.

```
CREATE [ OR REPLACE ] FUNCTION nazwa_funkcji
   (parametr1 TRYB TYP_DANYCH [ DEFAULT wyrażenie ],
    parametr2 TRYB TYP_DANYCH [ DEFAULT wyrażenie ],
    ...)
RETURN TYP_DANYCH
AS
```

```
[   zmienna1 TYP_DANYCH;
    zmienna2 TYP_DANYCH;
    ... ]
BEGIN
   instrukcje_wykonawcze;
   RETURN wyrażenie;

[ EXCEPTION
   WHEN nazwa_wyjątku
   THEN
      instrukcje_wykonawcze ]
END;
/
```

Różnice pomiędzy wzorcem dotyczącym funkcji i wzorcem procedury są minimalne. Poza zamianą słowa kluczowego `PROCEDURE` na `FUNCTION` w instrukcji `CREATE`, kod różni się tylko w dwóch miejscach. Pierwszym z nich jest nagłówek, gdzie podaje się zwracany typ danych, a drugim jest część wykonawcza, gdzie następuje bezpośrednie przekazanie zwracanej wartości do programu wywołującego.

`RETURN` *`TYP_DANYCH`*

W nagłówku wyrażenie `RETURN` stanowi część deklaracji funkcji. Jest to informacja o typie danych wartości, które mają być zwracane w wyniku wywołania funkcji.

`RETURN` *`wyrażenie`*

Wewnątrz sekcji wykonawczej wyrażenie `RETURN` jest instrukcją i oznacza, że działania zostały zakończone i że należy zwrócić (*return*) wartość, którą definiuje *`wyrażenie`*. Instrukcja `RETURN` może znaleźć się także w sekcji `EXCEPTION`.

Obydwie wymienione instrukcje są wymagane. W następnym podrozdziale przedstawiono kod gotowej funkcji.

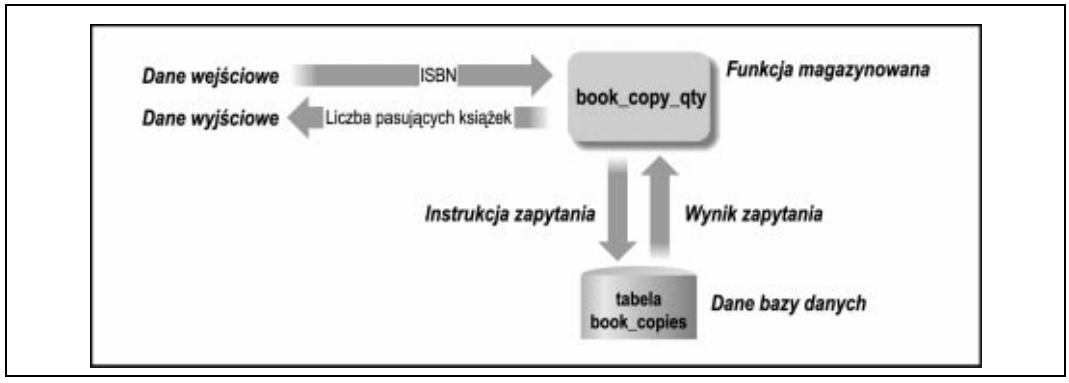
Kod funkcji book_copy_qty

Funkcja `book_copy_qty` zwraca liczbę książek, których numer ISBN odpowiada podanemu. W poniższym, przykładowym kodzie funkcji do pobrania danych z bazy danych wykorzystano *kursor* (*cursor*). Szczegółowe omówienie tej struktury znajduje się dopiero w rozdziale 5. Uogólniając, kursor jest określonym miejscem w pamięci, do którego program może przenieść pewne dane pobrane z bazy danych. W sekcji deklaracji następuje powiązanie pewnej instrukcji `SELECT` z kursorem. Powiązanie to zachodzi za pomocą instrukcji `CURSOR`. Warto zwrócić uwagę, że parametr wejściowy `isbn_in` występuje po prawej stronie wyrażenia `WHERE`. W celu pobrania danych należy kursor otworzyć (*open*), pobrać (*fetch*) z niego dane, a w końcu go zamknąć (*close*).

```
CREATE OR REPLACE FUNCTION book_copy_qty(isbn_in IN VARCHAR2)
RETURN NUMBER
AS
   number_o_copies NUMBER := 0;
   CURSOR bc_cur IS
      SELECT COUNT(*)
        FROM book_copies
       WHERE isbn = isbn_in;
BEGIN
```

```
    IF isbn_in IS NOT NULL
    THEN
        OPEN bc_cur;
        FETCH bc_cur INTO number_o_copies;
        CLOSE bc_cur;
    END IF;
    RETURN number_o_copies;
END;
/
```

Konstrukcja funkcji niewiele odbiega od procedury. Różnią się one jednak swoim zachowaniem — szczegóły dotyczące działania funkcji przedstawiono na rysunku 3.6.



Rysunek 3.6. Wykorzystanie funkcji języka PL/SQL w celu zwrócenia pewnej wartości do programu wywołującego; dla danej wartości ISBN funkcja zwraca liczbę egzemplarzy znajdujących się w bazie danych

Zasadnicza różnica pomiędzy wywołaniem funkcji i procedury wynika z tego, że funkcje zwracają pewną wartość.

Używanie funkcji

Najprostszym sposobem wykorzystania wartości wynikowej, zwracanej przez funkcję, jest jej przypisanie do pewnej zmiennej, której typ odpowiada typowi zwracanej wartości. Zwykle wygląda to w następujący sposób:

```
DECLARE
   zmienna_lokalna TYP_DANYCH;
BEGIN
   zmienna_lokalna := nazwa_funkcji (argument1, argument2, ...);
END;
/
```

Jest to zatem typowa instrukcja przypisania, gdzie wywołanie funkcji znajduje się po prawej stronie operatora przypisania, a zmienna lokalna po lewej. Można więc napisać:

```
DECLARE
   ile INTEGER;
BEGIN
   ile := book_copy_qty('1.56592-335-9');
END;
/
```

Z informacji przedstawionych w rozdziale 1. wynika, że w celu wyświetlenia wyników na ekranie można przekazać je do funkcji DBMS_OUTPUT.PUT_LINE:

```
SET SERVEROUTPUT ON
BEGIN
   DBMS_OUTPUT.PUT_LINE('Liczba kopii 1-56592-335-9: '
      || book_copy_qty('1-56592-335-9'));
END;
/
```

Funkcja book_copy_qty zwraca wartość typu VARCHAR2, co umożliwia konkatenację z innym ciągiem znaków i użycie funkcji PUT_LINE. Takie zagnieżdżanie funkcji wewnątrz innych instrukcji jest często stosowaną techniką.

Niektóre zasady dotyczące wykorzystania funkcji

Poniżej wymieniono kilka wartych zapamiętania zasad dotyczących wykorzystania funkcji:

1. Nie można tworzyć niezależnych (*standalone*) funkcji o takich samych nazwach, jakie noszą niezależne procedury. Jednakże przyjęcie konwencji nazywania procedur wyrażeniem czasownikowym, a funkcji — wyrażeniem rzeczownikowym może uchronić przed potencjalnymi problemami.
2. Nieuwzględnienie w nagłówku wyrażenia RETURN spowoduje, że funkcja nie zostanie skompilowana. Jest to działanie pozytywne, gdyż błędy czasu kompilacji uważa się za metodę wczesnego ostrzegania przed ewentualnymi problemami. Jeśli jednak słowo kluczowe RETURN zostanie pominięte w sekcji wykonawczej, system Oracle poinformuje o tym dopiero po uruchomieniu funkcji. Zostanie wtedy wygenerowany komunikat o błędzie *ORA-06503: PL/SQL: Function returned without value*. Dlatego zawsze warto przeprowadzać dogłębne testy napisanych programów.

Procedury a funkcje

Uogólniając, funkcja służy do przeprowadzenia pewnych operacji, których celem jest otrzymanie pewnej wartości. Oczywiście, jest możliwe wykorzystanie procedury zamiast funkcji i zastosowanie pojedynczego parametru OUT, jednak takie postępowanie nie jest zalecane.

Warto też pamiętać, że nie należy pisać programów, które zwracają wartość informującą o stanie zakończenia. Postępowanie takie jest uzasadnione w języku takim jak C, lecz w przypadku PL/SQL informowanie o błędach powinno się odbywać poprzez przechwytywanie wyjątków. Jeśli program magazynowany nie przechwyci wyjątku, zakłada się, że podczas wykonania tego programu nie napotkano żadnych problemów.

Czasami okazuje się, że istnieje potrzeba zwrócenia więcej niż jednej wartości — w takim przypadku należy utworzyć procedurę. W celu zwrócenia wielu wartości trzeba wykorzystać kilka parametrów w trybie OUT lub IN OUT, których wartość może być odczytana przez program wywołujący po zakończeniu działania procedury. Zdefiniowanie kilku parametrów w trybie OUT może wprowadzić nieco zamieszania, jednak istnieje metoda zapisywania programów, które muszą zwracać większą ilość danych w bardziej czytelny sposób. Można bowiem różne elementy połączyć w jedną całość przez zastosowanie jednego ze złożonych typów danych, jak na przykład rekord, kolekcja lub tak zwany typ obiektowy. Zagadnienia te opisano w rozdziale 5.

3. Program wywołujący po wywołaniu funkcji musi w jakiś sposób wykorzystać zwróconą wartość, na przykład przypisać ją pewnej zmiennej. PL/SQL, w przeciwieństwie do języka C, nie dopuszcza wywołania funkcji i zignorowania zwracanej wartości.
4. Po wykonaniu instrukcji `RETURN` w sekcji wykonawczej do programu wywołującego zostaje przekazana nie tylko wartość, ale także sterowanie wykonaniem programu. Innymi słowy, kod zapisany po instrukcji `RETURN` nie jest wykonywany.

Warto również zwracać uwagę na stosowaną terminologię. Czasami, gdy ktoś mówi o „procedurach magazynowanych", naprawdę ma na myśli programy magazynowane, czyli albo procedury, albo funkcje. Z drugiej jednak strony, może chodzić właśnie o procedury, nie zaś funkcje. Jeśli ogólny kontekst rozmowy nie pozwala na rozpoznanie, o którą możliwość chodzi, z pewnością prośba o wyjaśnienie nie będzie objawem dyletanctwa.

Wyjątki przechwytywane przez funkcje w sekcji deklaracji i związane z nimi możliwe problemy

Powyżej przedstawiono następujący sposób wywołania funkcji:

```
DECLARE
   ile INTEGER;
BEGIN
   ile := book_copy_qty('1-56592-335-9');
END;
/
```

Wydawałoby się, że powyższy kod można zapisać w bardziej zwięzły sposób poprzez wywołanie funkcji podczas inicjalizacji zmiennej:

```
DECLARE
   ile INTEGER := book_copy_qty('xyz');
BEGIN
```

Z pozoru wszystko jest w porządku. Jednak jeśli funkcja przechwyci wyjątek, to okazuje się, że żaden kod obsługi wyjątku w tej sekcji nie zdoła przechwycić wyjątku, który ewentualnie może pojawić się po takim wywołaniu funkcji `book_copy_qty`. A zatem:

```
DECLARE
   ile INTEGER := book_copy_qty('xyz');
BEGIN
  ...cokolwiek...
EXCEPTION
   WHEN OTHERS
   THEN
      /* NIESPODZIANKA! Wyjątki przechwycone w sekcji deklaracji
      || NIE MOGĄ być obsłużone w tym miejscu!
      */
...
END;
/
```

Wytłumaczenie powodu takiego zachowania wykracza poza zakres niniejszej książki. Dlatego należy po prostu zapamiętać, że nie należy dokonywać prób inicjalizowania zmiennych za pomocą funkcji.

Tworzenie elastycznego kodu

Z pewnością Czytelnik chciałby kontynuować rozwijanie tworzonej, przykładowej aplikacji. Nie można jednak zapominać o konsekwentnym upewnianiu się, że zapisany od nie zawiera błędów. Z tego też względu poniżej zamieszczono pewnego rodzaju dygresję.

Z pewnością każdy zetknął się już z wyrażeniem *garbage in, garbage out* (GIGO). Jest to wyrażenie dość często używane lub słyszane przez telefon podczas rozmowy z obsługą techniczną. Oznacza ono mniej więcej tyle, co „wprowadzasz błędne dane — uzyskujesz błędne wyniki".

Uniknięcie problemu GIGO jest sprawą dość problematyczną. Istnieje tendencja do nieuzasadnionego optymizmu w kwestii wykorzystywania utworzonych aplikacji. Chętniej przyjmuje się założenie *tidy in, tidy out* (wprowadzasz poprawne dane — uzyskujesz poprawne wyniki)[3]. Nikt nie lubi z góry planować działań zapobiegawczych względem zastosowania niepoprawnych danych wejściowych.

W celu uniknięcia nieprzewidzianych załamań programu trzeba przeprowadzić serie testów. Przeprowadzenie dobrych testów oznacza generowanie przeróżnych kombinacji danych testowych, które potencjalnie powinny „zawiesić" program. Potem należy sprawdzić oczekiwany wynik, uruchomić program, porównać z danymi wyjściowymi, poprawić program i wreszcie… ponownie przeprowadzić testy.

Z pewnością warto uprościć ten cały proces.

Monotonne wykonywanie fragmentu kodu z różnymi danymi wejściowymi jest dobrą okazją do napisania pewnych programów użytkowych. Poniżej przedstawiono sposób zautomatyzowania tego męczącego procesu za pomocą takich programów.

Narzędzie sprawdzające wyniki

Działanie pierwszego, sprawdzającego programu polega na porównywaniu dwóch wartości i wyświetlaniu komunikatu o pomyślnym lub niepomyślnym przebiegu tego testu. U podstaw testowania leży właśnie porównywanie wartości — wartości faktycznej ze spodziewaną wartością wyjściową. Dlatego też wspomniany program jest bardzo użyteczny. Zawierał on także opis poszczególnych iteracji, tak aby umożliwić otrzymanie informacji o tym, dla jakich wartości wykonanie programu będzie niepomyślne. Poniżej znajduje się kod tej procedury „test równości" (`testrownosci`):

```
CREATE OR REPLACE PROCEDURE reporteq (description IN VARCHAR2,
      expected_value IN VARCHAR2, actual_value IN VARCHAR2)
AS
BEGIN
   DBMS_OUTPUT.PUT(description || ': ');

   IF expected_value = actual_value
      OR (expected_value IS NULL AND actual_value IS NULL)
   THEN
      DBMS_OUTPUT.PUT_LINE('PASSED');
```

[3] Nieprzetłumaczalna gra słów: *garbage* to po angielsku śmieci, bzdury; *tidy* — czysty, schludny — *przyp. tłum.*

```
   ELSE
      DBMS_OUTPUT.PUT_LINE('FAILED.  Expected ' || expected_value
         || '; got ' || actual_value);
   END IF;
END;
/
```

W powyższej procedurze porównywaniu podlegają ciągi znaków (`VARCHAR2`), ale można oczywiście napisać analogiczne procedury obsługujące liczby, daty czy też wartości logiczne.

Jednostka testująca procedury add_book

Kolejnym krokiem będzie napisanie programu wywołującego procedurę `add_book` na wiele sposobów — także w razie zastosowania nieprawidłowych danych wejściowych. Poniższy program ze względu na jego obszerność (89 linii kodu) podzielono na fragmenty. Dodatkowo ponumerowano poszczególne linie w celu ułatwienia odwoływania się do nich.

```
 1 DECLARE
 2  l_isbn VARCHAR2(13) := '1-56592-335-9';
 3  l_title VARCHAR2(200) := 'Oracle PL/SQL Programming';
 4  l_summary VARCHAR2(2000) := 'Reference for PL/SQL developers, ' ||
 5      'including examples and best practice recommendations.';
 6  l_author varchar2(200) := 'Feuerstein, Steven, and Bill Pribyl';
 7  l_date_published DATE := TO_DATE('01-WRZ-1997', 'DD-MON-YYYY');
 8  l_page_count NUMBER := 987;
 9  l_barcode_id VARCHAR2(100) := '100000001';
10
11   CURSOR bookCountCur IS
12      SELECT COUNT(*) FROM books;
13
14   CURSOR copiesCountCur IS
15      SELECT COUNT(*) FROM book_copies;
16
17   CURSOR bookMatchCur IS
18      SELECT COUNT(*) FROM books
19       WHERE isbn = l_isbn AND title = l_title AND summary = l_summary
20         AND author = l_author AND date_published = l_date_published
21         AND page_count = l_page_count;
22
23   CURSOR copiesMatchCur IS
24      SELECT COUNT(*) FROM book_copies
25       WHERE isbn = l_isbn AND barcode_id = l_barcode_id;
26
27   how_many NUMBER;
28   l_sqlcode NUMBER;
```

Poniżej znajduje się kilka wyjaśnień dotyczących przedstawionego fragmentu kodu.

Linie 2. – 9. Są to deklaracje zmiennych lokalnych, które reprezentują wartości używane w różnych testach. Przechowywanie tych wartości w zmiennych ułatwia programowanie z uwagi na możliwość ich wielokrotnego użycia. Przedrostek `l_` oznacza, że są to zmienne lokalne.

Linie 11. – 12. Jest to deklaracja pierwszego z kursorów programu. Kursor umożliwia pobieranie wartości z bazy danych za pomocą instrukcji języka SQL `SELECT`. Ta konkretna instrukcja oblicza całkowitą liczbę rekordów w tabeli `ksiazki`.

Linie 14. – 15. Jest to również deklaracja kursora programu — umożliwia on obliczanie całkowitej liczby egzemplarzy książki.

Linie 17. – 21. Ten kursor służy do obliczania liczby książek, dla których wartości kolumn odpowiadają wartościom zmiennych lokalnych.

Linia 27. Zmienna lokalna `ile` przechowuje tymczasowo rezultat działania powyższych zapytań.

Linia 28. Zmienna `l_sqlcode` przechowuje tymczasowo wartość wyjściową wewnętrznej funkcji PL/SQL — `SQLCODE`. Jej działanie zostanie omówione nieco dalej.

Poniżej przedstawiono sekcję wykonawczą omawianego programu. Rozpoczyna ona swoje działanie od usunięcia zawartości obydwóch tabel w bazie danych. W ten sposób uzyskuje się pewność, że wszelkie operacje zliczania będą dotyczyć jedynie bieżących danych testowych, a nie innych, które mogły pozostać po wcześniejszych wykonaniach. Oczywiście, cała ta procedura powinna dotyczyć pewnej testowej bazy danych, a nie bazy podstawowej.

```
29 BEGIN
30    DELETE book_copies;
31    DELETE books;
32
33    add_book(isbn_in => l_isbn, barcode_id_in => l_barcode_id,
34       title_in => l_title, summary_in => l_summary, author_in => l_author,
35       date_published_in => l_date_published, page_count_in =>
l_page_count);
36
37    OPEN bookMatchCur;
38    FETCH bookMatchCur INTO how_many;
39    reporteqbool('add procedure, book fetch matches insert',
40       expected_value => TRUE, actual_value => bookMatchCur%FOUND);
41    CLOSE bookMatchCur;
42
```

Linie 33. – 41. W tym miejscu następuje pierwsze wywołanie procedury `add_book`. Wykorzystywane są wszystkie zdefiniowane wcześniej parametry i oczekiwane jest poprawne zakończenie procedury. Wywołanie to rozpoczyna proces sprawdzający, czy procedura dodawania książki działa poprawnie. Następuje to przez otwarcie kursora, pobranie z niego danych i sprawdzenie, czy, zgodnie z oczekiwaniami, rekord został utworzony. W liniach 39. – 40. znajduje się wywołanie procedury `reporteqbool` — jest to odmiana procedury `reporteq`, która wykorzystuje wartości logiczne (Boolean) zamiast ciągów znaków. Jeśli pobranie danych do kursora kończy się powodzeniem, wartością `bookMatchCur%FOUND` jest TRUE (więcej informacji na temat wykorzystania tej techniki znajduje się w rozdziale 5.). Jak wynika z treści linii 41., dobrym zwyczajem jest zamykanie kursora po jego użyciu.

```
43    BEGIN
44       add_book(isbn_in => NULL, barcode_id_in => 'foo', title_in => 'foo',
45          summary_in => 'foo', author_in => 'foo',
46          date_published_in => SYSDATE, page_count_in => 0);
47       l_sqlcode := SQLCODE;
48    EXCEPTION
49    WHEN OTHERS THEN
50       l_sqlcode := SQLCODE;
51    END;
52
```

```
53    reporteq('add procedure, detection of NULL input',
54       expected_value => '-6502', actual_value => TO_CHAR(l_sqlcode));
55
```

Linie 43. – 54. Ten fragment kodu służy do przeprowadzenia kolejnego testu — tym razem następuje próba określenia wartości NULL dla parametru `isbn` w celu sprawdzenia prawidłowego działania detekcji błędnych danych wejściowych. Jeśli procedura `add_book` działa poprawnie, powinna przechwycić wyjątek `NO_DATA_FOUND`. Programista powinien zostać poinformowany o tym fakcie, a zatem tekst został umieszczony w bloku zagnieżdżonym. W ten sposób można obsłużyć wyjątek niejako w kolejnej instrukcji zamiast przeskakiwać do końcowej części bloku głównego (sekcji obsługi wyjątków).

W celu zachowania zgodności z innymi przeprowadzanymi testami, tu również trzeba przeprowadzić porównanie otrzymanej wartości z pewną wartością oczekiwaną. PL/SQL zawiera specjalną funkcję wewnętrzną `SQLCODE`, która wywołana z poziomu kodu obsługi wyjątku zawsze zwraca określoną wartość różną od zera. Z uwagi na fakt, że wartość ta ma zostać wykorzystana poza kodem obsługi wyjątku, w linii 50. następuje przypisanie tej wartości zmiennej `l_sqlcode`, która następnie jest wykorzystywana w liniach 53. i 54. podczas wywołania procedury `reporteq`.

Z linii 54. wynika, że wartość oczekiwana wynosi –6502. Jest to wartość, którą PL/SQL przypisuje `SQLCODE` w sytuacji, gdy pojawia się wyjątek `NO_DATA_FOUND`.

```
56    OPEN bookCountCur;
57    FETCH bookCountCur INTO how_many;
58    reporteq('add procedure, book_record count', expected_value => '1',
59       actual_value => how_many);
60    CLOSE bookCountCur;
61
62    OPEN copiesCountCur;
63    FETCH copiesCountCur INTO how_many;
64    reporteq('add procedure, book_copy record count', expected_value => '1',
65       actual_value => how_many);
66    CLOSE copiesCountCur;
67
68    OPEN copiesMatchCur;
69    FETCH copiesMatchCur INTO how_many;
70    reporteqbool('add procedure, book copy fetch matches insert',
71       expected_value => TRUE, actual_value => copiesMatchCur%FOUND);
72    CLOSE copiesMatchCur;
73
```

Linie 56. – 72. Kolejne linie kodu służą do sprawdzenia, czy w tabeli znajduje się oczekiwana liczba rekordów.

```
74    BEGIN
75       add_book(isbn_in => l_isbn, barcode_id_in => l_barcode_id,
76          title_in => l_title, summary_in => l_summary, author_in =>
l_author,
77          date_published_in => l_date_published,
78          page_count_in => l_page_count);
79       l_sqlcode := SQLCODE;
80    EXCEPTION
81       WHEN OTHERS THEN
82          l_sqlcode := SQLCODE;
83    END;
```

```
84    reporteq('add procedure, detection of duplicate isbn',
85       expected_value => '-1', actual_value => l_sqlcode);
86 END;
87 /
```

Linie 74. – 85. Kolejny test, uwzględniony w omawianym programie, polega na sprawdzeniu, czy próba wprowadzenia drugi raz tej samej wartości `isbn` spowoduje przechwycenie wyjątku. System Oracle powinien w takiej sytuacji ustawić wartość `SQLCODE` na –1, co oznacza właśnie próbę wstawienia rekordu o takim samym kluczu głównym, co rekord już istniejący (w rzeczywistości jest to test poprawności projektu bazy danych).

Jest to w zasadzie ostatni punkt kontrolny testu poprawności procedury `add_book`. Jeżeli przed uruchomieniem programu wykonano komendę `SERVEROUTPUT ON` (dokładniejszy opis znajduje się w rozdziale 2.), wywołanie omawianego programu z poziomu SQL*Plus powoduje wyświetlenie następujących informacji:

```
add procedure, book fetch matches insert: PASSED
add procedure, detection of null input: PASSED
add procedure, book record count: PASSED
add procedure, book_copy record count: PASSED
add procedure, book_copy fetch matches insert: PASSED
add procedure, detection of duplicate isbn: PASSED
```

Być może Czytelnik zauważył, że powyższy kod służy jako test jednostki programowej (*unit test*)[4]. Może on także posłużyć jako zapisany przykład wywoływania programu, co może okazać się przydatne w razie zaistnienia wątpliwości dotyczących sposobu przeprowadzenia podobnej procedury.

Testowanie funkcji book_copy_qty

Poniższy program służy do testowania pracy funkcji `book_copy_qty`. Zasadniczo sposób jego działania jest taki sam, jak w przypadku poprzedniego programu.

```
 1 DECLARE
 2    l_isbn VARCHAR2(13) := '1-56592-335-9';
 3    l_isbn2 VARCHAR2(13) := '2-56592-335-9';
 4    l_title VARCHAR2(200) := 'Programowanie w Oracle PL/SQL';
 5    l_summary VARCHAR2(2000) := 'Kompendium informacji o języku PL/SQL ' ||
 6        'wraz z przykładami i poradami na temat programowania.';
 7    l_author varchar2(200) := 'Feuerstein, Steven, Bill Pribyl';
 8    l_date_published DATE := TO_DATE('01-WRZ-1997', 'DD-MON-YYYY');
 9    l_page_count NUMBER := 987;
10    l_barcode_id VARCHAR2(100) := '100000001';
11    l_barcode_id2 VARCHAR2(100) := '100000002';
12    l_barcode_id3 VARCHAR2(100) := '100000003';
13
14    how_many NUMBER;
15 BEGIN
16    DELETE book_copies;
17    DELETE books;
```

[4] Wyrażenie „jednostka programowa” (*unit*) odnosi się tu do pojedynczej jednostki programowej w przeciwieństwie do innych rodzajów testów, które pozwalają upewnić się, co do prawidłowej współpracy poszczególnych jednostek w ramach całej aplikacji.

```
18
19    reporteq('book_copy_qty function, zero count', '0',
20       TO_CHAR(book_copy_qty(l_isbn)));
21
22    /* Zakładamy, że procedura add_book działa prawidłowo */
23    add_book(isbn_in => l_isbn, barcode_id_in => l_barcode_id,
24       title_in => l_title, summary_in => l_summary, author_in => l_author,
25       date_published_in => l_date_published, page_count_in =>
l_page_count);
26
27    reporteq('book_copy_qty function, unit count', '1',
28       TO_CHAR(book_copy_qty(l_isbn)));
29
30    add_book_copy(isbn_in => l_isbn, barcode_id_in => l_barcode_id2);
31    add_book_copy(isbn_in => l_isbn, barcode_id_in => l_barcode_id3);
32
33    reporteq('book_copy_qty function, multi count', '3',
34       TO_CHAR(book_copy_qty(l_isbn)));
35
36    reporteq('book_copy_qty function, null ISBN', '0',
37       TO_CHAR(book_copy_qty(NULL)));
38 END;
39 /
```

Linie 30. – 31. Ten fragment kodu zawiera wywołania procedury, która jeszcze nie została Czytelnikowi przedstawiona. Procedura ta służy do wstawienia rekordu do tabeli `book_copies`.

Jeżeli uruchomienie jednostki testowej powoduje wyświetlenie wyników pokazanych poniżej (przy założeniu, że przed uruchomieniem programu wykonano polecenie `SERVEROUTPUT ON`), oznacza to prawidłowe działanie funkcji `book_copy_qty`:

```
book_copy_qty function, zero count: PASSED
book_copy_qty function, unit count: PASSED
book_copy_qty function, multi count: PASSED
book_copy_qty function, null ISBN: PASSED
```

Po co tyle dodatkowej pracy?

Być może Czytelnik się zastanawia nad zasadnością tworzenia takiej ilości dodatkowego kodu. Być może po prostu wystarczyłoby przeglądanie programu w celu wyszukania ewentualnych błędów.

Takie sprawdzanie kodu jest sposobem postępowania przeciętnego programisty. Istotę problemu najlepiej oddaje sprawdzenie wartości `NULL`, wykonywane w liniach 40. – 41. omawianego programu sprawdzającego. Co prawda, ostateczny test wersji procedury `add_book` przebiegł pomyślnie, jednak początkowo Autorzy wcale nie brali pod uwagę konieczności sprawdzania wartości `NULL`. Dopiero w trakcie pisania jednostki testowej okazało się, że należy uwzględnić co najmniej trzy przypadki wprowadzania danych wejściowych: określenie danych poprawnych, niepoprawnych oraz uwzględnienie tej problematycznej wartości `NULL`. Ustaleń tych dokonano jedynie dzięki analizie możliwych danych wejściowych, które należy uwzględniać podczas planowania testów. Wcześniejsze poprawne działanie programu okazało się kwestią przypadku. Testowanie i sposób jego zaplanowania umożliwia analizę tworzonego kodu z zupełnie odmiennego punktu widzenia, co pozwala na uzyskanie nowych, ważnych informacji.

Czas przeznaczony na testowanie aplikacji jest niekiedy ograniczany z powodów finansowych lub w rezultacie decyzji kierownictwa firmy. Częściej jednak testowanie aplikacji jest pomijane przez programistów, którzy postrzegają tę czynność jako nudną lub nawet niepotrzebną. Bezwzględnie należy jednak wyrobić sobie nawyk testowania pisanych przez siebie aplikacji.

Wykorzystanie pakietów PL/SQL w celu organizacji kodu

Dotychczas Czytelnik zapoznał się z kodem umożliwiającym przeprowadzanie kilku operacji związanych z obsługą katalogu. Przedstawiono także kod kilku narzędzi służących do testowania.

Poniżej opisano pewne uproszczenia, jakie dotychczas dopuszczano wobec omawianego problemu. Przedstawiono także kilka sposobów pokonania tych ograniczeń.

Kod, który zaprezentowano we wcześniejszych podrozdziałach, nie rozwiązuje jeszcze pewnych ważnych zagadnień, takich jak:

- Możliwość, że rekord wstawiany do tabeli `books` już istnieje. Trzeba sprawdzić, czy jest to traktowane tak samo, jak wprowadzenie informacji o nowym egzemplarzu.
- Sposób modyfikowania danych w katalogu.
- Sytuacja, w której dany rekord katalogu biblioteki zostanie przypadkowo usunięty lub utracony. Trzeba przeanalizować możliwości wykorzystania PL/SQL do przekazania tej informacji do systemu bazy danych.
- Sytuacja, w której zaistnieje potrzeba wykonania wielu różnych zapytań, np. wyszukiwanie książek na podstawie kilku różnych kryteriów.

Oczywiście, w dalszej części niniejszej książki zostaną przedstawione sposoby rozwiązania wielu z powyższych problemów. Zaprezentowane będą odpowiednie fragmenty kodu i sposób ich dodawania. Z powyższego wynika, że wygodnym rozwiązaniem byłoby posiadanie mechanizmu pozwalającego na organizowanie kodu w taki sposób, aby umożliwić jego łatwiejsze rozwijanie i utrzymywanie. Mechanizmem takim są *pakiety* (*packages*).

Pakiet języka PL/SQL jest pewnym nazwanym zbiorem, który może zwierać dowolną liczbę procedur i funkcji. Pakiety mogą także przechowywać inne obiekty, na przykład wyjątki, zmienne czy deklaracje typów danych. W trakcie lektury dalszej części niniejszej książki Czytelnik przekona się o ogromnych korzyściach, które płyną z tych dodatkowych możliwości. W tym rozdziale jednak zostanie przedstawiony tylko sposób wykorzystania teoo mechanizmu do wstawienia już utworzonego programu do pakietu.

W przypadku innych języków programowania — na przykład Java lub Ada — także istnieją konstrukcje zwane pakietami. Jednak pakiety w języku PL/SQL charakteryzują specyficzne definicje i zachowania. Warto o tym pamiętać w przypadku zapoznawania się z tymi strukturami w innych językach.

Części pakietu

Z powodów, które staną się niedługo oczywiste, pakiety składają się zazwyczaj z dwóch części: *specyfikacji* (*specification*) — w skrócie *spec* — oraz *ciała* (*body*).

Specyfikacja pakietu

Specyfikacja pakietu zawiera jedynie informacje o jego możliwościach (przeznaczeniu), ale nie ma tu żadnych wiadomości o sposobie jego wykonania. Specyfikacja zawiera jedynie nagłówki jednostek programowych, a nie kod wykonywalny. Jest to rodzaj sekcji deklaracyjnej, występującej w jednostkach programowych. Uproszczony wzorzec składni definicji pakietu przedstawiono poniżej:

```
CREATE OR REPLACE PACKAGE nazwa_pakietu
AS

   nagłówek_programu1;
   nagłówek_programu2;
   nagłówek_programu3;

END nazwa_pakietu;
/
```

`nazwa_pakietu` jest opisową nazwą, jaką użytkownik nadaje danemu pakietowi (nazwa ta podlega zasadom nadawania nazw obowiązującym w języku PL/SQL).

Poniżej przedstawiono sposób utworzenia pakietu, służącego do zarządzania książkami w bazie danych. Pierwszym problemem jest nadanie odpowiedniej nazwy. Część programistów uważa, że taki rodzaj pakietu pełni rolę zarządcy (*manager*) i dlatego zastosowaną przez nich nazwą byłoby `book_mgr` lub `book_man`. Inni raczej podkreślają rodzaj obiektu i użyliby nazwy `book_pkt`. Autorzy preferują jednak nazwy krótkie i proste — a zatem wybrali prostą nazwę `book`[5].

Poniżej przedstawiono przykładową specyfikację pakietu:

```
CREATE OR REPLACE PACKAGE book
AS
   PROCEDURE add(isbn_in IN VARCHAR2, title_in IN VARCHAR2,
      author_in IN VARCHAR2, page_count_in IN NUMBER,
      summary_in IN VARCHAR2 DEFAULT NULL,
      date_published_in IN DATE DEFAULT NULL,
      barcode_id_in IN VARCHAR2 DEFAULT NULL);

   PROCEDURE add_copy(isbn_in IN VARCHAR2, barcode_id_in IN VARCHAR2);

   FUNCTION book_copy_qty(isbn_in IN VARCHAR2)
   RETURN NUMBER;

   PROCEDURE change(isbn_in IN VARCHAR2, new_title IN VARCHAR2,
      new_author IN VARCHAR2, new_page_count IN NUMBER,
      new_summary IN VARCHAR2 DEFAULT NULL,
      new_date_published IN DATE DEFAULT NULL);
```

[5] Wywoływanie programów umieszczonych w pakietach odbywa się w następujący sposób: *nazwa_pakietu.nazwa_programu* — stąd możliwe okazuje się używanie zwięzłych, lecz zrozumiałych wywołań w rodzaju `book.add(...)`.

```
   PROCEDURE remove_copy(barcode_id_in IN VARCHAR2);

   PROCEDURE weed(isbn_in IN VARCHAR2);
END book;
/
```

W większości przypadków (także w powyższym) nie ma potrzeby umieszczania w specyfikacji pakietu procedur i funkcji w określonym porządku.

Warto zauważyć, że instrukcja definicji specyfikacji pakietu rozpoczyna się od wyrażenia `CREATE OR REPLACE`, zaś poszczególne programy w nagłówku mają postać:

```
FUNCTION nazwa ...
```

lub:

```
PROCEDURE nazwa ...
```

Jest to spowodowane faktem, że w przypadku języka PL/SQL cały kod pojedynczego pakietu jest tworzony, uaktualniamy lub usuwany jednocześnie, tak więc sensowne jest tylko zastosowanie jednej instrukcji `CREATE`.

Z powyższego wynika, że w planach rozwoju aplikacji uwzględniono dodanie kilku brakujących funkcji (jednakże metody niezbędne do pobierania danych z bazy danych nie zostaną na razie utworzone).

Specyfikacja pakietu pełni rolę interfejsu programowania aplikacji (*application programming interface, API*) w odniesieniu do pakietu. Nazwa ta jest być może nieco nieścisła, lecz oddaje istotę idei stosowania pakietu. Każdy API służy bowiem jako forma komunikacji między użytkownikami (np. między piszącymi program i wykorzystującymi go). Sens takiego postępowania polega na ukryciu niejednokrotnie bardzo skomplikowanych rozwiązań programistycznych za ich prostą prezentacją.

Ciało pakietu

Ciało pakietu zawiera elementy strukturalne jednostek programowych, czyli instrukcje wykonawcze odpowiadające nagłówkom podanym w specyfikacji pakietu. Uproszczony wzorzec składniowy ciała pakietu znajduje się poniżej:

```
CREATE OR REPLACE PACKAGE BODY nazwa_pakietu
AS
   programy_prywatne;  /* opcjonalnie */

   ciało_programu1;
   ciało_programu2;
   ciało_programu3;
END nazwa_pakietu;
/
```

Innymi słowy, ciało pakietu jest tą lokalizacją, gdzie się umieszcza implementacje programów wymienionych w specyfikacji.

Niemal zawsze ciało pakietu tworzy się po zdefiniowaniu jego specyfikacji. Kompilator sprawdza, z którą specyfikacją jest związane dane ciało poprzez sprawdzenie jego nazwy, która musi być identyczna z wymienioną w specyfikacji. Ponadto ciało pakietu musi zawierać definicje wszystkich

programów, jakie zostały wymienione w specyfikacji — w przeciwnym wypadku kompilacja się nie powiedzie. Jednakże warto zauważyć, że zasada ta nie działa odwrotnie — ciało pakietu może zawierać pewne dodatkowe programy, których nie wymieniono w specyfikacji pakietu. Na przedstawionym powyżej wzorcu opisano je jako *programy_prywatne*.

Kod ciała pakietu `book` zawiera jedną procedurę prywatną (*private*):

```
CREATE OR REPLACE PACKAGE BODY book
AS
   /* "prywatna" procedura używana tylko w ciele tego pakietu */
   PROCEDURE assert_notnull (tested_variable IN VARCHAR2)
   IS
   BEGIN
      IF tested_variable IS NULL
      THEN
         RAISE VALUE_ERROR;
      END IF;
   END assert_notnull;

   FUNCTION book_copy_qty(isbn_in IN VARCHAR2)
   RETURN NUMBER
   AS
      number_o_copies NUMBER := 0;
      CURSOR bc_cur IS
         SELECT COUNT(*)
           FROM book_copies
          WHERE isbn = isbn_in;
   BEGIN
      IF isbn_in IS NOT NULL
      THEN
         OPEN bc_cur;
         FETCH bc_cur INTO number_o_copies;
         CLOSE bc_cur;
      END IF;
      RETURN number_o_copies;
   END;

   PROCEDURE add(isbn_in IN VARCHAR2, title_in IN VARCHAR2,
      author_in IN VARCHAR2, page_count_in IN NUMBER,
      summary_in IN VARCHAR2, date_published_in IN DATE,
      barcode_id_in IN VARCHAR2)
   IS
   BEGIN
      assert_notnull(isbn_in);

      INSERT INTO books (isbn, title, summary, author, date_published,
         page_count)
      VALUES (isbn_in, title_in, summary_in, author_in, date_published_in,
         page_count_in);

      IF barcode_id_in IS NOT NULL
      THEN
         add_copy(isbn_in, barcode_id_in);
      END IF;

   END add;

   PROCEDURE add_copy(isbn_in IN VARCHAR2, barcode_id_in IN VARCHAR2)
   IS
```

```
    BEGIN
       assert_notnull(isbn_in);
       assert_notnull(barcode_id_in);
       INSERT INTO book_copies (isbn, barcode_id)
       VALUES (isbn_in, barcode_id_in);
    EXCEPTION
       WHEN DUP_VAL_ON_INDEX
       THEN
          NULL;
    END;

    PROCEDURE change(isbn_in IN VARCHAR2, new_title IN VARCHAR2,
       new_author IN VARCHAR2, new_page_count IN NUMBER,
       new_summary IN VARCHAR2 DEFAULT NULL,
       new_date_published IN DATE DEFAULT NULL)
    IS
    BEGIN
       assert_notnull(isbn_in);
       UPDATE books
          SET title = new_title, author = new_author, page_count =
          new_page_count,
              summary = new_summary, date_published = new_date_published
        WHERE isbn = isbn_in;
       IF SQL%ROWCOUNT = 0
       THEN
          RAISE NO_DATA_FOUND;
       END IF;
    END change;

    PROCEDURE remove_copy(barcode_id_in IN VARCHAR2)
    IS
    BEGIN
       assert_notnull(barcode_id_in);
       DELETE book_copies
        WHERE barcode_id = barcode_id_in;
    END remove_copy;

    PROCEDURE weed(isbn_in IN VARCHAR2)
    IS
    BEGIN
       assert_notnull(isbn_in);
       DELETE book_copies WHERE isbn = isbn_in;
       DELETE books WHERE isbn = isbn_in;
       IF SQL%ROWCOUNT = 0
       THEN
          RAISE NO_DATA_FOUND;
       END IF;
    END weed;

END book;
/
```

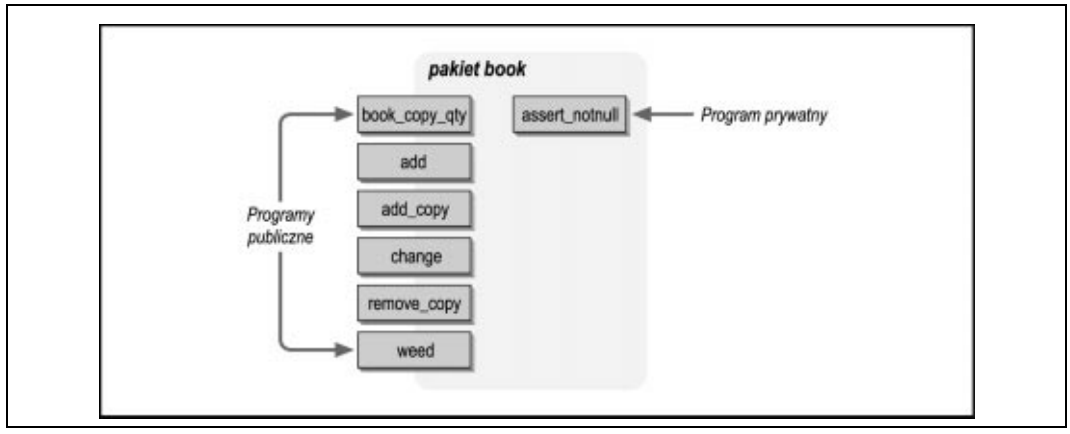
W powyższym, dość długim przykładzie znajduje się kilka dotąd nie omówionych konstrukcji, przykładowo, konstrukcja `SQL%ROWCOUNT`. Zostaną one jednak omówione w dalszych rozdziałach.

Większość programistów umieszcza programy w ciele pakietu w takiej samej kolejności, jak w przypadku specyfikacji (programy prywatne uwzględnia się na początku, gdyż ze względu na wymagania kompilatora ich definicja musi się znajdować przed ewentualnym wywołaniem). W razie niezdefiniowania którejś z jednostek programowych, wymienionych w specyfikacji, kompilator nie przeprowadzi kompilacji, dopóki błąd ten nie zostanie naprawiony.

`programy_prywatne` są „prywatne” w sensie dostępności jedynie z poziomu pakietu. Jego użytkownicy nie mogą bezpośrednio wywoływać żadnego z programów prywatnych. Pozwala to na konstruowanie narzędzi o specjalnym przeznaczeniu, które będą dostępne tylko dla innych programów, zdefiniowanych w ramach pakietu. W podanym wyżej przykładzie dodano tylko jedną, prostą procedurę `assert_notnull`, która pozwala na niepowtarzanie kilku linii kodu w każdej definiowanej procedurze. Używanie takich lokalnych programów często jest bardzo wygodne. Najczęściej spełniają one następujące zadania:

- służą jako dodatkowe metody, wykorzystywane przez inne programy zawarte w pakiecie;
- pozwalają na uniknięcie powtarzania tego samego kodu;
- są dobrą metodą przechowywania lub zmieniania wartości pewnej zmiennej wewnętrznej.

Z kolei programy, które są wymienione w specyfikacji pakietu, mogą być wywoływane w dowolny sposób i nazywa się je programami publicznymi (*public*). Rysunek 3.7 przedstawia graficzną reprezentację porównania programów publicznych z prywatnymi.



Rysunek 3.7. Pakiety udostępniają interfejs publiczny ukrywający wewnętrzną implementację

Wielu programistów nie dostrzega ważności zagadnień związanych z wykorzystaniem programów publicznych i prywatnych, mimo że jest to jedna z technik pozwalających na tworzenie trwałego, zrozumiałego i łatwego do ponownego wykorzystania kodu. Informatycy, którzy wszystkiemu nadają pewne formalne nazwy, powiedzieliby, że oznacza to tworzenie „abstrakcji” (*abstraction*) książki poprzez ukrywanie informacji (*information hiding*) — patrz: słowniczek.

Korzyści wynikające z wykorzystywania pakietów

Poniżej wymieniono najistotniejsze korzyści wynikające z wykorzystywania pakietów:

Organizacja

Większość ludzi docenia dobre zorganizowanie działania. Grupowanie powiązanych ze sobą jednostek programowych w pakiety pozwala na tworzenie kodu zorganizowanego w wygodne do stosowania struktury. Używając terminologii fachowej można powiedzieć, że pakiety pozwalają na użycie mechanizmów abstrakcji, enkapsulacji oraz ukrywania informacji.

Zrozumiałość

Pakiety w ogromnym stopniu ułatwiają zarządzanie dużą liczbą jednostek programowych. Pewne badania z dziedziny psychologii dowiodły, że człowiek w jednym czasie może zajmować się jedynie około siedmioma zagadnieniami. Oczywiście, nie zawsze można ograniczyć liczbę komponentów pakietu do takiej liczby, ale ich grupowanie z pewnością ułatwia pracę.

Możliwości projektowe

Rozwiązanie skomplikowanego problemu odbywa się zazwyczaj na drodze rozbicia go na mniejsze składowe. Dwa najpopularniejsze, zupełnie od siebie różne, sposoby rozwiązywania tego problemu (czyli *dekompozycji*) to podejście *funkcjonalne* (*functional*) oraz *obiektowe* (*object-based*). Zastosowanie pakietów wspomaga obydwie te techniki projektowe.

Wydajność

Podczas pierwszego w danej sesji uruchomienia programu zapisanego w pakiecie system Oracle wczytuje do pamięci całą zawartość pakietu, a nie tylko program właśnie wywołany. Pozwala to na znaczące zwiększenie wydajności wywołania innych programów pakietu, gdyż nie będzie konieczne korzystanie z danych zapisanych na wolniejszym dysku twardym. Z powyższego jednocześnie wynika, że w pakiecie warto przechowywać tylko te komponenty, które są ze sobą związane.

Wygoda pracy sesyjnej

Czasem jest przydatna możliwość przechowywania w pamięci pewnych tymczasowych lub stałych wartości w trakcie trwania sesji (przyjęto, że sesją nazywa się okres zalogowania użytkownika w systemie bazy danych). Dzięki wykorzystaniu pakietów wartości takie można przechowywać w globalnych lub lokalnych zmiennych. Trzeba jednak pamiętać, że cecha ta staje dużo mniej przydatna w przypadku aplikacji działających w sieci Internet. Aplikacje te zazwyczaj nie są zależne od sesji — więcej informacji dotyczących tego tematu znajduje się w rozdziale 4. Bez zastosowania pakietów wartości takie musiałyby być przechowywane w samej bazie danych, co miałoby niekorzystny wpływ na wydajność systemu. Problematyczne też stałoby się odwoływanie się do pewnej wartości po wycofaniu transakcji.

Szczególne cechy PL/SQL

Jedną z najciekawszych możliwości uzyskiwanych dzięki wykorzystaniu pakietów jest możliwość tworzenia wielu jednostek programowych o takiej samej nazwie. Pozwala to na wywoływanie jednego programu, który może obsługiwać dane wejściowe o różnych typach. Zastosowanie tej techniki, zwanej *przeciążaniem* (*overloading*) i niedostępnej dla programów napisanych bez użycia pakietów, pozwala na uczynienie programu pozornie bardziej ogólnym. Przykładowo, funkcja `TO_CHAR` systemu Oracle jest przeciążana w celu umożliwienia przyjmowania parametrów o różnym typie danych, na przykład `NUMBER` lub `DATE`. Opis techniki przeciążania oraz przykładowy program znajdują się w końcowej części niniejszego rozdziału, w podrozdziale „Uproszczenie poprzez przeciążenie”.

Mniejsze problemy z rekompilacją

W dużych systemach składających się z dziesiątków lub setek programów dużym problemem jest zazwyczaj rekompilowanie części składowych programu po dokonaniu pewnych zmian. Dzięki pakietom można uniknąć wielu trudności dzięki temu, że można rekompilować zawartość

ciała pakietu bez konieczności rekompilacji programów, które wykorzystują zmodyfikowany program. W przypadku zmiany specyfikacji pakietu trzeba już jednak ponownie skompilować każdy program uwzględniony w pakiecie, ale często system Oracle robi to automatycznie.

Usuwanie programów magazynowanych

Warto zwrócić uwagę, że kod procedury `add_book` wprowadzony do pakietu nie różni się od procedury oryginalnej poza zmienioną nazwą `dodaj`, gdyż będzie ona stosowana tylko w kontekście pakietu `book`.

Oryginalna procedura `add_book` wciąż istnieje, ale można ją usunąć — po części dlatego, że dzięki wykorzystaniu pakietu procedura ta już nie jest potrzebna. Poniżej zatem pokazano sposób usuwania procedury. Odpowiednią instrukcją SQL, która w nieodwracalny sposób usuwa procedurę z bazy danych, jest instrukcja `DROP PROCEDURE`:

```
DROP PROCEDURE add_book;
```

Jest to instrukcja SQL, a nie PL/SQL. Stąd poniższy sposób wykonania:

```
BEGIN
   DROP PROCEDURE add_book; /* nie zadziała w TYM miejscu */
END;
```

spowoduje błąd kompilacji i wyświetlenie komunikatu:

```
drop procedure add_book;
*
BŁĄD w linii 2:
ORA-06550: linia 2, kolumna 1:
PLS-00103: Napotkano symbol "DROP" gdy oczekiwano jednego z następujących:
...ciach...
```

Z powyższego wynika, że PL/SQL rzeczywiście nie jest pełnym nadzbiorem języka SQL.

Po usunięciu procedury nie jest możliwe jej odzyskanie bez pomocy administratora bazy danych, który musi przeprowadzić pewne działania związane z odzyskiwaniem danych. Jeśli jednak istnieje plik zawierający kod źródłowy, można po prostu utworzyć ponownie daną procedurę.

W rzeczywistości nie powinno się usuwać obiektów bazy danych bez upewnienia się, że nie są już potrzebne — w przeciwnym wypadku wcześniej czy później spowoduje to problemy.

Dobrym zwyczajem jest przechowywanie specyfikacji oraz ciała pakietu w dwóch oddzielnych plikach. W ten sposób łatwo można poddać rekompilacji samo ciało po dokonaniu pewnych zmian. Niepotrzebne wykonanie instrukcji zawartych w specyfikacji pakietu sprawiłoby, że nieaktualne stałyby się powiązania innych programów z pakietem, a zatem także wymagana ich rekompilacja.

Jak się okazuje, istnieje wiele powodów, dla których warto używać pakietów. A zatem warto powrócić do zagadnień związanych z programami testującymi i umieścić je w pakiecie.

Ulepszenie programu testującego

Poniżej przedstawiono sposób utworzenia kolejnego pakietu. Zorganizowanie procedur testowych w jednym pakiecie służyć ma głównie w celu utrzymania porządku w kodzie. Na tym etapie zostaną pominięte kwestie związane z wydajnością i kilkoma innymi zagadnieniami, które omówiono wcześniej. Pakiet testujący zapewnia wygodny sposób obsługi, a jego struktura umożliwia świetną współpracę ze strukturą pakietu `book`, którego budowę przedstawiono we wcześniejszej części niniejszego rozdziału.

Poniżej zamieszczono specyfikację omawianego pakietu:

```
CREATE OR REPLACE PACKAGE test_book AS
   PROCEDURE run (verbose IN BOOLEAN DEFAULT TRUE);
   PROCEDURE add;
   PROCEDURE add_copy;
   PROCEDURE book_copy_qty;
   PROCEDURE change;
   PROCEDURE remove_copy;
   PROCEDURE weed;
END test_book;
/
```

Każda jednostka programowa głównego pakietu `book` posiada odpowiednik w pakiecie `test_book`, choć nie uwzględniono żadnych parametrów. Można wywoływać je pojedynczo lub wywołać ogólną procedurę `run`, która uruchamia wszystkie pozostałe. Posiada ona jeden opcjonalny parametr logiczny wskazujący, czy mają być prezentowane szczegółowe informacje dotyczące testów. Oprócz tego każda procedura zawiera kod służący do testowania.

Jeśli wszystko przebiega prawidłowo, przeprowadzenie testu z poziomu SQL*Plus wygląda następująco:

```
SQL> SET SERVEROUTPUT ON SIZE 1000000
SQL> execute test_book.run
Testing book package...
...add procedure, detection of NULL input: PASSED
...add procedure, book_record count: PASSED
...add procedure, book_copy record count: PASSED
...add procedure, book fetch matches insert: PASSED
...add procedure, book copy fetch matches insert: PASSED
...add procedure, detection of duplicate isbn: PASSED
...add_copy procedure, nominal case, first book: PASSED
...add_copy procedure, nominal case, second book: PASSED
...add_copy procedure, ignore duplicates: PASSED
...add_copy procedure, bad isbn detection: PASSED
...add_copy procedure, NULL isbn detection: PASSED
...add_copy procedure, NULL barcode_id detection: PASSED
...book_copy_qty function, zero count: PASSED
...book_copy_qty function, non-zero count: PASSED
...change procedure, single field test: PASSED
...change procedure, NULL barcode_id detection: PASSED
...remove_copy procedure, book count normal: PASSED
...remove_copy procedure, book copy count normal: PASSED
...remove_copy procedure, superfluous invocation: PASSED
...weed procedure, book count normal: PASSED
...weed procedure, book copy count normal: PASSED
...weed procedure, superfluous invocation: PASSED
book package: PASSED
```

Można także wyłączyć tryb szczegółowych komunikatów, podając wartość FALSE przy wywołaniu procedury `run`. Odpowiednie polecenie przedstawiono poniżej:

```
SQL> SET SERVEROUTPUT ON SIZE 1000000
SQL> execute test_book.run(verbose => FALSE)
book package: PASSED
```

Omawiany pakiet jest zbyt obszerny, aby zamieścić go w tym miejscu. Posiada on kilka udogodnień w porównaniu do wersji pierwotnej. Pełna wersja omawianego pakietu znajduje się pod adresem *ftp://ftp.helion.pl/przyklady/orplsq.zip* Odwiedzając witrynę wydawnictwa O'Reilly, Czytelnik może także zapoznać się z utPLSQL (*http://oracle.oreilly.com/utplsql*), darmową aplikacją służącą do testowania.

Celowość działań

Być może Czytelnik uważa, że w treści niniejszego rozdziału położono zbyt duży nacisk na kwestię testowania oprogramowania. Warto jednak uświadomić sobie, że programowanie jest działaniem ukierunkowanym na szczegóły — użytkownicy aplikacji są bardzo wymagający i jakiekolwiek lekceważenie potencjalnych problemów jest bardzo ryzykowną postawą (istnieje tak zwany „styl programowania ekstremalnego"[6] gloryfikujący taką postawę). A zatem program testowy należy tworzyć bardzo wcześnie. Niektórzy uważają, że program testowy powinien być tworzony jeszcze przed napisaniem głównej aplikacji. Zbyt długie zwlekanie z napisaniem programu testowego może w efekcie spowodować, że nigdy nie powstanie, a nawet jeśli powstanie, ewentualna zwłoka może uniemożliwić osiągnięcie pewnych korzyści wynikających z testowania. Myśląc o spodziewanych wynikach na wczesnym etapie tworzenia aplikacji łatwiej jest znaleźć błędy i naprawić je szybko i bez ponoszenia zbędnych kosztów.

Pisanie dobrych programów testowych nie jest zadaniem trywialnym. Dlatego wielu programistów, piszących aplikacje pod presją czasu, stwierdza, że może pominąć ten etap. Warto jednak zapamiętać następującą dewizę:

> *Jeśli nie ma czasu, aby wykonać zadanie prawidłowo za pierwszym razem, to skąd wziąć czas, aby wykonać je od nowa?*

Autorzy uważają, że aby zostać naprawdę dobrym programistą, należy tworzyć programy testowe. Taki sposób postępowania umożliwi spędzanie większej ilości czasu na pisaniu nowych programów, a mniej na usuwaniu błędów w programach starych.

Przejście na wyższy poziom

Po zdobyciu pewnego doświadczenia w wykorzystywaniu procedur, funkcji i pakietów Czytelnik z pewnością zechce podnieść poziom swojej pracy programistycznej. Poniżej znajduje się kilka porad i sugestii dotyczących tego zagadnienia.

---

[6] Więcej informacji na ten temat można znaleźć w książce poświęconej tej tematyce, której autorem jest Kent Beck lub zajrzeć na witrynę *http://www.extremeprogramming.org/*.

Nadawanie nazw plikom

Przyjęcie odpowiedniego sposobu nazywania plików, które zawierają kod źródłowy tworzonych programów, jest z pewnością bardzo ważnym zagadnieniem. Przykładowo, w przypadku plików zawierających kod PL/SQL warto, aby rozszerzenie nazwy pliku odzwierciedlało jego zawartość. W rozdziale 6. znajduje się bardziej szczegółowe omówienie tego zagadnienia. Poniżej podano jedynie kilka głównych wytycznych:

Wzorzec nazwy pliku	Zawartość
nazwa.pro	(Pojedyncza) procedura magazynowana
nazwa.fun	(Pojedyncza) funkcja magazynowana
nazwa.sql	Blok anonimowy lub skrypt zawierający wiele bloków, instrukcji SQL i (lub) poleceń SQL*Plus
nazwa.pks	Specyfikacja pakietu
nazwa.pkb	Ciało pakietu

W razie zastosowania powyższych konwencji niektóre z plików, które wymieniono w niniejszym rozdziale, nosiłyby następujące nazwy:

add_book.pro	**Procedura magazynowana add_book**
book_copy_qty.fun	**Funkcja magazynowana book_copy_qty**
test_add_book.sql	**Program testujący dla add_book**
book.pks	**Specyfikacja pakietu book**
book.pkb	**Ciało pakietu book**

Specyfikacja i ciało pakietu powinny być umieszczone w oddzielnych plikach.

Ponowne używanie kodu

Czytelnik nie powinien sądzić, że wydajność pracy programisty jest wprost proporcjonalna do liczby linii napisanego kodu. Dawno temu był to pogląd licznych teoretyków programowania. Jednak małe jest piękne — programista powinien starać się osiągać duży efekt kosztem małego wysiłku. Oto pewna historyjka:

> *Pewien Prawdziwy Programista zdołał zmieścić program dopasowywania do wzorca* (*pattern matching*) *w kilkuset kilobajtach nieużywanej pamięci sondy Voyager, której zadaniem było odszukanie i sfotografowanie nowego księżyca Jowisza*[7].

Bez względu na to, czy powyższa anegdota jest prawdziwa czy nie, każdy programista powinien starać się pisać kod zwięzły (ale nie niezrozumiały). Jednym ze sposobów na osiągnięcie tego celu jest projektowanie kodu możliwego do ponownego wykorzystania. Poniższy przykład umożliwi zrozumienie problemu.

[7] Fragment pochodzi z artykułu „Real Programmers Don't Use PASCAL", Datamation, July 1983, s. 263 – 265.

Poniżej przedstawiono działanie procedury `reporteq`.

```
Wypisz opis testu.
Porównaj wartość oczekiwaną z wartością otrzymaną.
Jeśli obydwie wartości są równe, wypisz komunikat PASSED, jeśli nie — FAILED.
```

Poniższy kod już był omawiany w niniejszym rozdziale, ale został powtórzony dla zachowania przejrzystości wykładu.

```
CREATE OR REPLACE PROCEDURE reporteq (description IN VARCHAR2,
      expected_value IN VARCHAR2, actual_value IN VARCHAR2)
AS
BEGIN
   DBMS_OUTPUT.PUT(description || ': ');

   IF expected_value = actual_value
      OR (expected_value IS NULL AND actual_value IS NULL)
   THEN
      DBMS_OUTPUT.PUT_LINE('PASSED');
   ELSE
      DBMS_OUTPUT.PUT_LINE('FAILED.  Expected ' || expected_value
         || '; got ' || actual_value);
   END IF;
END;
/
```

Program ten porównuje dwie zmienne typu `VARCHAR2`. Następna funkcja, `reporteqbool`, posiada identyczny pseudokod, ale jest przeznaczona do porównywania wartości logicznych typu `BOOLEAN`. Można by dokonać tylko drobnych modyfikacji w oryginale, jak pokazano w poniższym fragmencie kodu:

```
CREATE OR REPLACE PROCEDURE reporteqbool (description IN VARCHAR2,
      expected_value IN BOOLEAN, actual_value IN BOOLEAN) AS
BEGIN
   DBMS_OUTPUT.PUT(description || ': ');
   IF (expected_value AND actual_value)
      OR (expected_value IS NULL AND actual_value IS NULL)
   THEN
      DBMS_OUTPUT.PUT_LINE('PASSED');
   ELSE
      DBMS_OUTPUT.PUT_LINE('FAILED.');
   END IF;
END;
/
```

Nawet programista, który nie jest odpowiedzialny za oprogramowanie sondy Voyager, może zauważyć, że powyższy kod charakteryzuje się pewną nadmiarowością. W razie konieczności modyfikacji działania części testującej lub raportującej (na przykład w celu zapisania wszystkich opisów i wyników testu w tabeli bazy danych) okazałoby się, że trzeba wykonać dwa razy więcej pracy. Problematyczne też może okazać się określenie dwóch różnych miejsc, w których należy dokonać zmian. Często takie zmiany są dokonywane przez kogoś innego, niż twórca oryginalnej wersji. Osoba taka może nie być świadoma istnienia konieczności zmodyfikowania kodu w dwóch miejscach. W rezultacie powstanie niezgodność, której wykrycie będzie kwestią przypadku.

W języku PL/SQL istnieje możliwość uniknięcia takiego problemu przez zmodyfikowanie drugiej procedury w taki sposób, aby wywoływała pierwszą. Aby to umożliwić, należy przekonwertować wartości logiczne `BOOLEAN` na `VARCHAR2`. Niestety, funkcja `TO_CHAR` nie obsługuje zmiennych

typu BOOLEAN. Można jednak napisać własną funkcję dokonującą tej konwersji. Zgodnie z obowiązującymi w systemie Oracle zasadami nadawania nazw funkcjom konwersji, jej nazwą będzie booleantochar:

```
CREATE OR REPLACE FUNCTION booleantochar(is_true IN BOOLEAN)
RETURN VARCHAR2
AS
BEGIN
   IF is_true
   THEN
      RETURN 'TRUE';
   ELSIF NOT is_true
   THEN
      RETURN 'FALSE';
   ELSE
      RETURN TO_CHAR(NULL);
   END IF;
END booleantochar;
/
```

Teraz można już zmodyfikować procedurę reporteqbool w następujący sposób:

```
CREATE OR REPLACE PROCEDURE reporteqbool (description IN VARCHAR2,
      expected_value IN BOOLEAN, actual_value IN BOOLEAN)
AS
BEGIN
   reporteq(description, booleantochar(expected_value),
      booleantochar(actual_value));
END reporteqbool;
/
```

Najważniejsza kwestią, na którą należy zwrócić uwagę, jest to, że cały kod związany z testowaniem znajduje się w procedurze reporteq. A zatem w razie konieczności zmiany sposobu jej działania wystarczy tylko zmienić kod jednego programu. Nawet jeśli osoba zajmująca się konserwacją danego kodu nie będzie wiedziała, który program należy zmienić, omawiany sposób tworzenia kodu umożliwi jej szybkie zorientowanie się w sytuacji.

Nie jest to jednak ostatnia metoda ułatwiania sobie pracy. Można uprościć ją jeszcze bardziej za pomocą techniki zwanej *przeciążaniem* (*overloading*).

Uproszczenie poprzez przeciążenie

Czytelnik zapewne przypomina sobie, że we wcześniejszej części niniejszego rozdziału opisano w skrócie technikę przeciążania. Opisano utworzenie procedury reporteq, która obsługuje ciągi znakowe oraz procedury reporteqbool, wykonującej porównania na wartościach logicznych typu BOOLEAN. Prawdopodobnie potrzebna byłaby także wersja obsługująca daty — funkcja reporteqdate, i tak dalej. Zadania wykonywane przez te wszystkie programy byłyby bardzo podobne. Warto zatem byłoby utworzyć jeden program (lub przynajmniej zastosować tylko jedną nazwę programu), który pełniłby funkcje wszystkich wyżej wymienionych. Pozwoliłoby to na przeniesienie wielu obowiązków użytkownika na program użytkowy. Im mniej nazw programów trzeba zapamiętać, tym lepiej.

Technika przeciążania służy właśnie do osiągnięcia tych celów. Uogólniając, przeciążenie procedury oznacza zdefiniowanie więcej niż jednej procedury o tej samej nazwie. W ramach pakietu ma się wówczas dostęp do faktycznie czterech różnych procedur. Poniżej znajduje się przykład wykorzystania tej techniki w pakiecie zawierającym zbiór programów testowych. Poniżej przedstawiono kod przykładowej specyfikacji pakietu:

```
CREATE OR REPLACE PACKAGE tut AS
   PROCEDURE reporteq (description IN VARCHAR2,
      expected_value IN VARCHAR2, actual_value IN VARCHAR2);

   PROCEDURE reporteq (description IN VARCHAR2,
      expected_value IN NUMBER, actual_value IN NUMBER);

   PROCEDURE reporteq (description IN VARCHAR2,
      expected_value IN BOOLEAN, actual_value IN BOOLEAN);

   PROCEDURE reporteq (description IN VARCHAR2,
      expected_value IN DATE, actual_value IN DATE);

   PROCEDURE inna_procedura;
END;
/
```

Poszczególne procedury `reporteq` mają tą samą nazwę i różnią się tylko typami danych parametrów. To jest część „ukryta". Należy jeszcze zaimplementować wszystkie cztery procedury w jednym ciele pakietu. Można uwzględnić ich wzajemne wywoływanie się, co opisano we wcześniejszej części niniejszego rozdziału. Autorzy pozostawili utworzenie ciała pakietu jako ćwiczenie dla Czytelnika.

Ogromną zaletą techniki przeciążania jest to, że używając takiej procedury lub funkcji PL/SQL, samodzielnie rozpoznaje ona, którą z nich należy wywołać:

```
DECLARE
   rozmiar_buta NUMBER;
   rezultat_szukania VARCHAR(64);
BEGIN
   ...
   tut.reporteq('procedura flubber, wykrycie maksymalnego rozmiaru buta',
      expected_result => 15, actual_result => rozmiar_buta);

   tut.reporteq('procedura flubber, wyszukiwanie morsa',
      expected_result => 'Jestem morsem', actual_result => rezultat_szukania);
END;
```

Oznacza to, że każde wywołanie procedury `reporteq` zostaje dopasowane do odpowiedniej sytuacji.

Istnieje kilka sytuacji, w których metoda przeciążania nie umożliwia prawidłowego działania systemu. Błąd jednak zostanie zwrócony dopiero w czasie próby uruchomienia programu. Poniżej wymieniono podstawowe zasady, których przestrzeganie pozwoli na prawidłowe działanie mechanizmu przeciążania:

1. Przeciążane programy muszą posiadać taką samą nazwę oraz znajdować się w tym samym pakiecie[8].

[8] Można także przeciążać programy dołączone w sekcji deklaracji.

2. Przeciążane programy muszą różnić się albo liczbą parametrów, albo rodzajem typu danych parametrów (uwzględniając pozycyjne porównanie). Przykładowo, w przypadku mechanizmu przeciążania system bazy danych, który wykonuje program, nie może stwierdzić różnicy pomiędzy typami `NUMBER` a `INTEGER`, ale będzie to możliwe w przypadku typów `NUMBER` i `VARCHAR2`, gdyż należą one do różnych rodzajów typów danych.
3. Procedura może być przeciążona funkcją, nawet jeśli powyższe dwa warunki nie są spełnione.
4. Przeciążane funkcje muszą się różnić czymś więcej, niż tylko typem danych wartości zwracanej.

W przypadku niespełnienia któregoś z wymienionych warunków zazwyczaj pojawia się błąd wykonania: *PLS-00307: too many declarations of '<nazwa podprogramu>' match this call*.

Dalsza droga

W niniejszym rozdziale przedstawiono różne porady dotyczące programowania obronnego. Ten styl pracy uwzględnia możliwość występowania najgorszych sytuacji. Prawidłowo napisany program powinien radzić sobie z błędnymi danymi wejściowymi nie dając jednocześnie błędnych danych wyjściowych.

Istnieje co najmniej kilka możliwości uniknięcia „syndromu błędów na wyjściu". Niektóre z nich zostały omówione w trakcie omawiania sposobu tworzenia pakietu, który obsługuje i chroni dane o książkach przechowywane w bazie danych. Podsumowując:

- Należy zawsze pamiętać o tym, że zmienne lub parametry PL/SQL mogą mieć wartość NULL, co jest szczególnie ważne w przypadku konstruowania instrukcji `IF-THEN`.
- Należy wykorzystywać „obwoluty tabel" i rozwijać w sobie dyscyplinę potrzebną do używania tej techniki.
- Podczas deklarowania parametrów programów magazynowanych należy nadawać wartości domyślne wszędzie tam, gdzie jest to uzasadnione.
- Ogólnie rzecz biorąc, należy korzystać raczej z notacji wymieniającej niż pozycyjnej, szczególnie w sytuacji, gdy daje to dodatkowe informacje, które należy uwzględnić.
- Należy unikać powtarzania tego samego kodu, w przeciwnym razie przyszłe jego modyfikacje będą bardziej narażone na powstawanie błędów.
- Należy organizować kod w postaci pakietów, zamiast używania wielu pojedynczych procedur i funkcji.
- Należy obsługiwać wyjątki tam, gdzie jest to uzasadnione oraz przechwytywać je w sytuacji, gdy istnieje możliwość zaistnienia problemów, których tworzony program nie będzie rozwiązywał samodzielnie.
- Należy korzystać z mechanizmu przeciążania w celu zmniejszenia złożoności aplikacji, upraszczając tym samym przyszły jej rozwój.
- Należy tworzyć pewne programy testujące dla każdej z tworzonych jednostek programowych.

W kolejnym rozdziale zostanie opisane rozszerzenie tworzonej aplikacji „na zewnątrz", w kierunku użytkownika końcowego poprzez utworzenie interfejsu użytkownika, służącego do obsługi niektórych z cech.