Conceitos de OOP em PL/SQL

Este artigo visa apresentar e comparar a implementação dos conceitos de programação orientada à objetos em Java e PL/SQL. Abordarei os benefícios e as principais deficiências na implementação em PL/SQL e algumas das soluções de contorno.

Tomei por base os conceitos de OOP em Java expostos no belíssimo artigo "OOPS Concepts in Java – OOPS Concepts Example" de autoria de Pankaj Kumar. Portanto, usarei a mesma estrutura e quase a mesma sua ordem de apresentação.

Os conceitos serão apresentados na primeira parte do artigo e na segunda parte farei a exposição comentada da implementação destes conceitos.

CONCEITUAÇÃO

Abstração

É o de ocultar os detalhes internos de algum artefato de código. Um exemplo disso é a ocultação da implementação de algum método de quem vai utilizá-lo. No PL/SQL essa técnica pode ser obtida mesmo sem usar OOP através de pacotes (packages) onde na especificação dos pacotes existe apenas a assinatura das funções e procedimentos. Para atuar com OOP é preciso usar os Tipos (TYPES).

Encapsulamento

É a técnica usada para implementar a abstração em OOP. É usada para definir os níveis de acesso às propriedades e métodos implementados pelo objeto. Em Java são utilizados os modificadores de acesso (private, public e protected).

Em PL/SQL não existem modificadores de acesso. Todas as propriedades e métodos definidas na especificação do Tipo (TYPE) são públicas. Não há como impedir o acesso direto à elas. Contudo, os métodos que estejam definidos no Corpo do Tipo (TYPE BODY) e que não estejam definidos no Tipo (TYPE) são privados. Não existe a figura do acesso protegido (protected).

Portanto, vemos aqui uma das primeiras deficiências da implementação de OOP do PL/SQL. Qualquer peça de código pode alterar as propriedades do objeto definidas no Tipo (TYPE), ao invés de utilizar métodos de acesso (getters/setters).

Herança

É o conceito pelo qual um objeto é baseado em um objeto preexistente. Herança é um mecanismo de reuso de código. O objeto preexistente é chamado de Superclasse e o método derivado é chamado de subclasse.

Herança permite a criação de estruturas hierárquicas semelhantes às usadas na classificação taxonômica (taxonomia) da biologia ou da geologia.

Em java usamos a palavra chave "extends" (extende) e em PL/SQL usamos a palavra "UNDER".

A herança define que o relacionamento ente objeto da subclasse e da superclasse é o de pertinência. Ou seja, um objeto da subclasse pertence ao conjunto de objetos da superclasse. Assim, um objeto da subclasse é um tipo de objeto da superclasse.

Exemplos:

1) INVERTEBRADO é um tipo de ANIMAL. Isso indica que todas as características presentes em ANIMAL estarão presentes em INVERTEBRADO.
2) COMPLEXO, INTEIRO e REAL são tipos de NÚMERO. Todas as operações possíveis para NÚMERO será possíveis para COMPLEXO, REAL e INTEIRO. Contudo, existem operações em números complexos que não são possíveis para números inteiros ($\sqrt{-1}$).

Polimorfismo

É o conceito pelo qual um objeto comporta-se de forma diferente dependendo da situação ou contexto. Existem duas formas de polimorfismo – polimorfismos em tempo de compilação (também chamado de sobrecarga) e em tempo de execução.

O polimorfismo em tempo de compilação pode ser obtido em PL/SQL sem o uso de OOP através de múltiplas assinaturas de procedimento e funções nos pacotes (PACKAGE). Pode ser obtido também utilizando OOP com Tipos (TYPE).

Já o polimorfismo em tempo de execução é atendido plenamente pela implementação de tipos (TYPE) em PL/SQL.

Para entender melhor o conceito de polimorfismo, é necessário ter compreendido o conceito de Herança.

No polimorfismo um objeto definido a partir da superclasse que receba um exemplar (instância) da subclasse, se comportará como se fosse da subclasse. Na implementação isso ficará mais simples de entender.

Associação

É o conceito pelo qual se define o relacionamento entre objetos. A associação também define a multiplicidade/cardinalidade entre os objetos. Por exemplo, dado que Professor e Aluno são duas classes de objetos, fica evidente que pode existir uma relação entre um determinado professor e vários alunos. De igual forma, um determinado aluno pode estar relacionado a diferentes professores. Logo, foi estabelecida inequivocamente que existe uma associação entre Professores e Alunos, e que está relação se dá na ordem de “n para m” onde “n” e “m” são números naturais (incluindo o zero).

Agregação

É um tipo especial de associação na qual os objetos possuem o seu próprio ciclo de vida, mas existe um tipo de “propriedade” ou “contenção”. Um objeto possui uma relação de “tem um” com o outro objeto. Por exemplo, dado que existam dois objetos, CARRO e MOTOR, fica claro que um carro ”tem um” (ou “contém um”) motor. Fica claro, que você pode eliminar o carro manter o motor e vice-versa.

Composição

É um tipo mais restritivo de agregação na qual um dos objetos possui ciclo de vida atrelado ao outro. Por exemplo, dado que existam dois objetos, HOTEL e QUARTO, fica claro que um quarto não deva mais existir se a hotel deixar de existir. Contudo o contrário não é verdade, um hotel pode existir, por exemplo, se um dos seus quartos for demolido ou que receba outra finalidade (transformá-lo em um escritório por exemplo).

Existem estudos e estudiosos que defendem o uso prioritário de Composição em detrimento da Herança. Veja aqui.

IMPLEMENTAÇÃO

Modelo de Classes

Considere o seguinte cenário. Existe uma superclasse chamada “ARQUIVO” que tem por objetivo manipular arquivos em texto através do PL/SQL usando os conceitos de OOP. Evidentemente é possível fazer isso usando o pacote UTL_TYPE usando o paradigma procedural.

Esta classe terá os seguintes métodos “abre”, “le”, “grava” e “fecha”, que respectivamente abrem o arquivo, leem uma linha do arquivo, gravam uma linha no arquivo e fecham o arquivo.

Esta superclasse possui duas subclasses “LEITURA” e “ESCRITA”, que como os seus nomes explicitam, uma permite trabalhar com arquivos apenas para leitura e outra para gravação. Caso uma leitura seja feita em um objeto de ESCRITA deve ser emitido um erro e vice-versa.

Código Fonte

Os códigos aqui exibidos estarão disponíveis no meu repositório público no GITHUB onde poderão ser baixados. Pretendo colocar um projeto Java equivalente.

Detalhes da implementação da especificação do tipo.

Os dois tipos de comandos básicos para você escrever OOP em PL/SQL são “CREATE TYPE” e o “CREATE TYPE BODY”. Eles são equivalentes em estrutura aos comandos “CREATE PROCEDURE” e “CREATE PROCEDURE BODY”. Nas views USER_SOURCE e USER_OBJECTS eles podem ser encontrados com TYPE IN (”TYPE”,”TYPE BODY”).

Vamos criar a parte pública da classe usando comando CREATE OR REPLACE TYPE:

```
CREATE OR REPLACE TYPE C_ARQ AS OBJECT
```

Observações:

1) *Vou apresentar o código em parte. O comando acima é parte do comando para criar a classe. Baixe e analise o código por inteiro.*
2) *Diferente do Java, não há uma convenção para definir o nome da classe, métodos e propriedades. Por isso, para diferenciar, eu coloco o “C_” como prefixo nos nome das minhas classes.*
3) *No Java o comando equivalente seria:*
   public class Arq {

Quando você cria uma classe (type), você deve definir a(s) sua(s) propriedade(s) e seu(s) método(s). Ao menos uma propriedade tem que existir. Isso já é uma diferença com o Java onde é possível criar classes sem propriedades. A outra, como já dito anteriormente, é que todas propriedades e métodos são públicos. A definição das propriedades vem logo após a criação da classe:

```
(id         INTEGER,
 dataType   INTEGER,
 nmDir      VARCHAR2(250),     -- PASTA OU DIRETORIO DO ARQUIVO
 nmArq      VARCHAR2(250),     -- NOME DO ARQUIVO (INCLUINDO EXTENSAO)
```

```
 tpArq      VARCHAR2(001),     -- TIPO DO ARQUIVO: R->LEITURA,
                               --                  W->GRAVACAO
 nuTamReg   INTEGER,           -- TAMANHO MAXIMO DO REGISTRO
 nuLin      INTEGER,           -- NUMERO DA LINHA A SER LIDA OU GRAVADA
 deLin      VARCHAR2(32767),   -- LINHA A SER LIDA OU GRAVADA
```

Você vai perceber que não existem (como dito antes) modificadores de acesso (private, public e protected). Assim todas as propriedades são públicas.

Observações:

1) *As propriedades estão com comentários para explicitar os seus objetivos.*
2) *As propriedades id e dataType são usadas para serem associadas ao manipulador de arquivo do pacote UTL_FILE.*
3) *Não é permitido declarar propriedade com alguns tipos compostos (arrays, utl_file etc).*
4) *No Java o comando equivalente seria:*
   public Integer nuTamReg;

Logo após você deve declarar os construtores. Caso você não forneça nenhum construtor, o PL/SQL vai lhe prover um que terá como parâmetros de entrada todas as propriedades declaradas.

Um construtor é uma função membro (MEMBER) que retorna a classe (RETURN SELF AS RESULT) No nosso casos teremos mais de um construtor:

```
--------------------------------------------------------------------
-- CONSTRUTOR SEM NENHUMA PROPRIEDADE
--------------------------------------------------------------------
CONSTRUCTOR FUNCTION C_ARQ
RETURN SELF AS RESULT,

--------------------------------------------------------------------
-- CONSTRUTOR COM TODAS AS PROPRIEDADES
--------------------------------------------------------------------
CONSTRUCTOR FUNCTION C_ARQ
(nmDir     VARCHAR2,
 nmArq     VARCHAR2,
 tpArq     VARCHAR2,
 nuTamReg  INTEGER)
RETURN SELF AS RESULT,

--------------------------------------------------------------------
-- DEFINE UM ARQUIVO COM TAMANHO DE 2000 BYTES
--------------------------------------------------------------------
CONSTRUCTOR FUNCTION C_ARQ
(nmDir     VARCHAR2,
 nmArq     VARCHAR2,
 tpArq     VARCHAR2)
RETURN SELF AS RESULT,

--------------------------------------------------------------------
-- DEFINE UM ARQUIVO DE LEITURA COM TAMANHO DE 2000 BYTES
--------------------------------------------------------------------
CONSTRUCTOR FUNCTION C_ARQ
(nmDir     VARCHAR2,
 nmArq     VARCHAR2)
RETURN SELF AS RESULT,
```

Observações:

1) *Tal qual o Java o construtor deve ter o mesmo nome que a classe.*
2) *No Java o comando equivalente seria:*
   public Arq() {

Depois é preciso definir os métodos.

```
MEMBER PROCEDURE abre,

MEMBER PROCEDURE le
(s_linha OUT VARCHAR2),

MEMBER PROCEDURE le,

MEMBER PROCEDURE grava
(e_linha IN VARCHAR2),

MEMBER PROCEDURE grava,

MEMBER PROCEDURE fecha
```

Observações:

1) *Caso você deseje que o método não seja sobrescrito nas subclasses você pode defini-lo com FINAL. Basta colocar a palavra reserva FINAL antes de MEMBER.*
2) *No Java o comando equivalente seria:*
   public String le() {

Por fim, você precisar encerrar a criação da sua classe (tipo).

```
) NOT FINAL;
```

Observações:

1) *Indica que a classe poderá possuir subclasses. Caso você deseje que a classe fosse a última da sua hierarquia, basta tirar o NOT.*
2) *Observe que ao final da declaração das propriedades e dos métodos foi utilizada a vírgula ao invés do ponto e vírgula.*
3) *Ao final do último método a vírgula foi omitida.*
4) *No Java basta colocar o fecha chaves (}).*

Detalhes da implementação do corpo do tipo.

Vamos criar a parte privada da classe com a implementação dos seus métodos usando comando CREATE OR REPLACE TYPE:

```
CREATE OR REPLACE TYPE BODY C_ARQN AS
```

Logo após passamos a criação dos métodos construtores. Vou expor apenas um deles, pois a exposição de todos seria repetitiva enfadonha

```
CONSTRUCTOR FUNCTION C_ARQ
(nmDir     VARCHAR2,
 nmArq     VARCHAR2,
 tpArq     VARCHAR2)
```

```
RETURN SELF AS RESULT
AS
BEGIN
    SELF.nmDir := nmDir;
    SELF.nmArq := nmArq;
    SELF.tpArq := tpArq;
    SELF.nuTamReg := 2000;
    SELF.nuLin := 0;
    RETURN;
END;
```

Este método recebe três parâmetros que foram usados para setar propriedades homônimas. As propriedades não fornecidas foram setadas com valores default.

As propriedades da classe foram prefixadas com a palavra chave “SELF” que é semelhante ao “this” do Java.

Ao final é emitido um comando RETURN que devolve a instância criada.

Diferente especificação, vista no item anterior, os comandos são sempre terminados com ponto e vírgula.

Em Java, o construtor equivalente seria:

```
      public Arq (String nmDir, String nmArq, String tpArq) {
            this.nmDir = nmDir;
            this.nmArq = nmArq;
            this.tpArq = tpArq;
            this.nuTamArq = 2000;
            this.nuLin = 0;
      }
```

Depois disso, passamos a inclusão dos métodos não construtores. Novamente, farei a apresentação de apenas um método, pois o nível de similaridade é amplo.

```
MEMBER PROCEDURE abre
IS
 ARQUIVO     SYS.UTL_FILE.FILE_TYPE;
 v_deMsgErro VARCHAR2(250);
BEGIN
   ARQUIVO := SYS.UTL_FILE.FOPEN(SELF.nmDir, SELF.nmArq,
                                 SELF.tpArq, SELF.nuTamReg);
   SELF.id := ARQUIVO.id;
   SELF.dataType := ARQUIVO.dataType;
   SELF.nuLin := 0;
EXCEPTION
   WHEN SYS.UTL_FILE.INVALID_PATH THEN
      v_deMsgErro := 'Caminho (' || SELF.nmDir
                  || ') ou Nome do arquivo ('
                  || SELF.nmArq || ') invalido';
      RAISE_APPLICATION_ERROR(-20305,v_deMsgErro);
END;
```

O objetivo e a implementação do método são muito simples. Ele tem por objetivo abrir e associar um manipulador ao arquivo. Caso seja impossível abrir o arquivo é lançada uma exceção. Utilizamos para abrir o arquivo a função FOPEN do pacote UTL_FILE.

Além disso, é digno de menção o "casting" realizado para pegar o *id* e o *datatype* do UTL_FILE e associá-los às propriedades homônimas da nossa classe. Elas serão utilizadas sempre que formos utilizar o UTL_FILE para fazer as operações em direção ao arquivo. Veja, por exemplo, um trecho de código do método "le".

```
…
/ARQUIVO    SYS.UTL_FILE.FILE_TYPE;
BEGIN
   ARQUIVO.id := SELF.id;
   ARQUIVO.dataType := SELF.dataType;
…
```

Como vocês podem perceber, fora o uso de algumas poucas palavras reservadas diferentes, a criação de TYPE é muito semelhante à criação de um PACKAGE.

Seguiremos apresentando os demais conceitos de herança e polimorfismo, que onde o processo começa a ficar interessante.

Implementando Herança na especificação da classe.

O processo começa com a definição da classe (TYPE) com o uso da palavra chave UNDER. Depois passamos a explicitação das propriedades da subclasse, para depois definirmos os métodos, inclusive construtores, que precisaram ser sobrescritos e dos métodos adicionais se necessário.

```
CREATE OR REPLACE TYPE C_ARQ_LEITURA UNDER C_ARQ (
-----------------------------------------------------------------------
-- ARQUIVOS DE LEITURA APENAS
-----------------------------------------------------------------------
-- NÃO DEFINI NENHUMA PROPRIEDADE ADICIONAL
-----------------------------------------------------------------------
CONSTRUCTOR FUNCTION C_ARQ_LEITURA
RETURN SELF AS RESULT,

CONSTRUCTOR FUNCTION C_ARQ_LEITURA
(nmDir     VARCHAR2,
 nmArq     VARCHAR2,
 tpArq     VARCHAR2,
 nuTamReg  INTEGER)
RETURN SELF AS RESULT,

CONSTRUCTOR FUNCTION C_ARQ_LEITURA
(nmDir     VARCHAR2,
 nmArq     VARCHAR2,
 tpArq     VARCHAR2)
RETURN SELF AS RESULT,

CONSTRUCTOR FUNCTION C_ARQ_LEITURA
(nmDir     VARCHAR2,
 nmArq     VARCHAR2)
RETURN SELF AS RESULT,

OVERRIDING MEMBER PROCEDURE grava
(e_linha IN VARCHAR2),
```

```
OVERRIDING MEMBER PROCEDURE grava
) FINAL
```

Criei novos construtores apenas para impedir que o arquivo de leitura seja instanciado com a propriedade *tpArq* associada ao valor “W” (write) que é de gravação. Caso isso aconteça será lançada uma exceção.

Sobrescrevi (override) os dois métodos *grava* pelo mesmo motivo. Caso seja utilizados será emitida uma exceção.

Por fim, defini que esta classe é FINAL impedindo que ela possa ter subclasses.

O grande problema quando você usa herança no PL/SQL é que uma vez que a classe possua subclasses, você só conseguirá alterar a especificação (TYPE), por exemplo, para adicionar uma nova propriedade se apagar todas as classes dependentes. Contudo, a alteração da implementação (TYPE BODY) é livre. Isso é sacal e limitante, mas não um empecilho.

Implementando Herança no corpo da classe.

A implementação é muito simples. Novamente para não ser repetitivo, vou exibir apenas um construtor e um método.

```
…
CONSTRUCTOR FUNCTION C_ARQ_LEITURA
(nmDir     VARCHAR2,
nmArq     VARCHAR2,
tpArq     VARCHAR2,
nuTamReg  INTEGER)
RETURN SELF AS RESULT
AS
BEGIN
  IF tpArq <> 'R' THEN
     RAISE BIB_EXCEPTIONS.INVALID_WRITE_OPERATION;
  END IF;
  SELF.nmDir := nmDir;
  SELF.nmArq := nmArq;
  SELF.tpArq := tpArq;
  SELF.nuTamReg := nuTamReg;
  SELF.nuLin := 0;
  RETURN;
END;
…
OVERRIDING MEMBER PROCEDURE grava
(e_linha IN VARCHAR2)
IS
BEGIN
   RAISE BIB_EXCEPTIONS.INVALID_WRITE_OPERATION;
END;
…
END;
```

Sem novidades. Apenas o lançamento da exceção é apresentado como diferença da implementação da superclasse.

Utilizando as nossas classes criadas

Podemos instanciar nossas classes em qualquer componente do PL/SQL. Sejam funções, procedimentos, pacotes, gatilhos e até mesmo em outras classes.

Basicamente o processo é muito simples e conta com as três etapas abaixo:

1) Declaração da instância
2) Construção da Instância
3) In.vocação do Método.

Para declarar uma nova instância, a sintaxe é muito similar à declaração de qualquer variável. Tem que estar em um espaço que seja permitido declarar uma variável. Eu gosto de prefixar as minhas variáveis que apontam para uma classe com "O_".

```
O_ARQR C_ARQ_LEITURA;
```

Para usar um construtor e criar uma instância, basta atribuir à esta variável o resultado da invocação de um construtor.

```
   O_ARQR := C_ARQ_LEITURA(nmDir => '/home/teste',
                           nmArq => 'TESTE.TXT',
                           tpArq => 'W');
```

Utilizando polimorfismo

Imagine que você possua um procedimento que receba como parâmetro uma superclasse, você pode passar para ela instâncias de qualquer das suas subclasses, bem como da própria superclasse, que os métodos serão polimorficamente invocados.

Veja que o procedimento abaixo espera receber como parâmetro duas instâncias da classe C_ARQ. A declaração "IN OUT" indica que o parâmetro é modificável e o "NOCOPY" indica que a passagem de parâmetros é por referência.

```
PROCEDURE TST_01_ABERTURA
(m_arqR  IN OUT NOCOPY C_ARQ,
 m_arqW  IN OUT NOCOPY C_ARQ)
IS
BEGIN
   /*
    * TESTANDO A ABERTURA DE UM ARQUIVO DE LEITURA COMO GRAVAÇÃO
    * É ESPERADO QUE EMITA A EXCEÇÃO INVALID_WRITE_OPERATION
    */
   BEGIN
      m_arqR := C_ARQ_LEITURA(nmDir => K_CAMINHO,
                              nmArq => K_NOME_ARQUIVO
                                    || '_01.TXT',
                              tpArq => 'W');

      DBMS_OUTPUT.PUT_LINE('01.1: NUNCA É PARA '
                           || 'CHEGAR NESTE PONTO');
      m_arqR.abre;
      m_arqR.fecha;
   EXCEPTION
      WHEN BIB_EXCEPTIONS.INVALID_WRITE_OPERATION THEN
```

```
         DBMS_OUTPUT.PUT_LINE('ERRO ESPERADO: ABERTURA MAL'
                        || ' SUCEDIDA DE ARQUIVO DE LEITURA'
                        || ' COMO ESCRITA');
   END;

   /*
    * TESTANDO A ABERTURA DE UM ARQUIVO DE ESCRITA COMO LEITURA
    * É ESPERADO QUE EMITA A EXCEÇÃO INVALID_READ_OPERATION
    */
   BEGIN
      m_arqW := C_ARQ_ESCRITA(nmDir => K_CAMINHO,
                              nmArq => K_NOME_ARQUIVO
                                    || '_02.TXT',
                              tpArq => 'R');

      DBMS_OUTPUT.PUT_LINE('01.2: NUNCA É PARA CHEGAR '
                     || 'NESTE PONTO');
      m_arqW.abre;
      m_arqW.fecha;
   EXCEPTION
      WHEN BIB_EXCEPTIONS.INVALID_READ_OPERATION THEN
         DBMS_OUTPUT.PUT_LINE('ERRO ESPERADO: ABERTURA '
                    || 'MAL SUCEDIDA DE ARQUIVO DE ESCRITA '
                    || 'COMO LEITURA');
   END;
END;
```

O objetivo do procedimento é demonstrar o polimorfismo. Primeiro as instâncias são recebidas em variáveis da sua superclasse. Os métodos invocados são os da subclasse. Se uma operação de escrita é feita sobre uma instância de leitura é emitida a exceção (pelo método correspondente) e vice-versa.

Se a operação é corretamente invocada nenhum problema é exibido.

Utilizando Associações

Imagine que você possua uma classe C_RELATORIO que tem como propriedade um arquivo para ESCRITA (no qual você gravará o seu relatório).

Você poderá facilmente fazer a invocação dos métodos “abre”, “grava” e “fecha”. Poderá fazer o controle da emissão salto de página e do cabeçalho, utilizando outras propriedades inteiras como NUMERO_LINHA e NUMERO_PAGINA.

Uma dificuldade nas associações é a criação de listas. O uso de Arrays é proibido para propriedades. Então você tem que simular isso usando métodos e parâmetros de métodos.

Conclusão

O uso de OOP é possível no PL/SQL com alguns cuidados, limitações e soluções de contorno.

Os cuidados estão ligados ao problema do encapsulamento, pois todas as propriedades são públicas. Então a manipulação direta das propriedades, principalmente para alteração deve ser evitado.

As limitações e soluções de contorno estão principalmente na impossibilidade de uso listas como propriedades. Com criatividade você consegue isso usando variáveis públicas e métodos que as manipulem. Mas isso também é uma violação ao encapsulamento.

Na minha experiência de que tem usado OOP em PL/SQL por mais de quinze anos. Eu faço uso apenas para manipulação de alguns objetos com endereços, telefones, CPF, CNPJ e arquivos. A passagem de objetos complexos como parâmetros é facilitada pela OOP.

Outra dificuldade é que a evolução dos mecanismo de OOP em PL/SQL são muito lentas.

Na verdade os programas ficam no meio do caminho entre a programação convencional (estruturada) e a OOP. Porém, na minha análise, se mostra útil.