Integrando Java com Bancos de Dados Relacionais

Euricelia Viana e Paulo Borba*
DI-UFPE

ABSTRACT

The development and mainteinance of modern software in organizations requires the use of Object-Oriented Programming Languages (OOPL) and different persistence mechanisms, mainly Relational Databases (RDB). In fact, some APIs have been constructed with the purpose of integrating OOPL with RDB. These APIs, however, must be properly used in order not to lose the benefits of object-orientation. This work presents a proposal for using those APIs to integrate the Java™ programming language with RDB. The proposed integration follows object-oriented principles, contributing to obtain reuse and extensibility, and minimizing the impact on efficiency.

RESUMO

O desenvolvimento e manutenção de software nas organizações aponta para a utilização de Linguagens de Programação Orientadas a Objetos (LPOO) e mecanismos de persistência diversos, principalmente, Bancos de Dados Relacionais (BDR). De fato, já existem diversas APIs (Application Programming Interfaces) construídas com o objetivo de possibilitar a integração entre LPOO e BDR. A utilização destas APIs, no entanto, deve ser feita de forma criteriosa, sob pena de perder facilmente os benefícios da orientação a objetos. Este trabalho apresenta uma proposta de como estas APIs devem ser utilizadas para integrar a linguagem de programação orientada a objetos Java™ com BDR. A integração proposta é realizada com a preocupação de obedecer aos princípios da orientação a objetos, contribuindo para a obtenção de reusabilidade e extensibilidade, e minimizando o impacto em eficiência.

1. Introdução

A tarefa de transformar um projeto Orientado a Objetos (OO) em uma implementação OO não-persistente pode ser considerada relativamente simples[1], se observado o fato de que tem sido amplamente estudada e de que já existe uma gama diversificada de métodos [3, 9, 12] e ferramentas CASE que facilitam a sua execução

Em se tratando, no entanto, de obter uma implementação OO persistente, torna-se necessário integrar LPOO com algum mecanismo de persistência, como BDR, por

* E-mail: {emv, phmb} @di.ufpe.br. Caixa Postal 7851. Recife-PE.

[1] Naturalmente, abstraindo o esforço de implementação da interface gráfica do usuário (ou GUI, *Graphical User Interface*).

exemplo. Há várias APIs [11,14] que possibilitam esta integração, mas elas devem ser utilizadas de acordo com determinadas diretrizes para não que não se perca os benefícios da orientação a objetos nem de BDR. Apesar de alguns dos métodos e ferramentas mencionados estabelecerem o mapeamento das classes da análise ou do projeto OO para as tabelas do BDR, a ausência de diretrizes bem definidas para realizar outros aspectos da integração propicia a geração de uma série de problemas que resultam da violação de princípios básicos da orientação a objetos, como encapsulamento, *information hiding* e modularização [1].

De fato, diversos ambientes de desenvolvimento, como Delphi™, JBuilder™ e Visual Café™, tornam semi-automática a integração de LPOO com BDR. No entanto, esta integração é realizada sem a preocupação de obedecer a critérios que garantam a continuidade e a reversibilidade [10] da implementação em relação ao projeto OO. Assim, obtém-se persistência a um alto custo, pois os princípios da orientação a objetos acabam não sendo completamente seguidos, uma vez que, mesmo que o código de interface com o usuário e uma parte do código de acesso a dados sejam OO, tem-se um código de negócio [13] não OO, conforme explicamos nas seções seguintes.

Os Bancos de Dados Orientados a Objetos (BDOOs) são uma alternativa para reduzir os problemas acima, pois oferecem homogeneidade ao processo de desenvolvimento, estendendo o uso do paradigma OO ao armazenamento dos dados persistentes, e facilitando a integração com as LPOOs, uma vez que ambos manipulam objetos [13]. O uso de BDOO, no entanto, não dispensa a existência de diretrizes bem definidas para realizar a integração; embora, caso estas diretrizes não existam, o nível de problemas gerados pelo programador tende a ser menor, devido à homogeneidade citada anteriormente.

Como BDOO não é, atualmente, a tecnologia de banco de dados dominante no mercado[2], justifica-se a definição de diretrizes que indiquem como fazer a integração de LPOO com BDR, de forma adequada, assegurando os benefícios das duas tecnologias e garantindo, portanto, importantes fatores de qualidade de software: reusabilidade, extensibilidade e eficiência [1]. O objetivo deste trabalho é formalizar estas diretrizes através da definição de uma arquitetura para realizar a integração de Java com BDR, utilizando a API JRB (Java Relational Binding) [11].

O restante do trabalho está organizado da seguinte forma. Inicialmente, apresentamos os princípios para a integração, ressaltando os requisitos que guiaram a definição da arquitetura. Em seguida, discutimos os problemas identificados na "Integração *ad hoc*", abordagem suportada pelos ambientes de desenvolvimento e usualmente utilizada na prática. Na Seção 4, propomos a arquitetura para a "Integração objeto-relacional", baseada na estruturação do sistema em camadas independentes de classes, discutindo como evitar os problemas da integração *ad hoc*. Apresentamos possíveis implementações para uma aplicação, utilizando, inicialmente, a "Integração *ad hoc*" e evoluindo para a "Integração Objeto-relacional". Java [8] é a linguagem de programação utilizada nos exemplos e o acesso ao banco de dados é feito através das APIs `java.sql` [14] e JRB. Nas conclusões, são citados os resultados de algumas experiências práticas com a utilização da arquitetura proposta e feitas considerações finais.

[2] Por conta do legado de investimento em sistemas desenvolvidos, pela cultura dos profissionais atuando no mercado ou até mesmo por questões de performance.

A arquitetura é apresentada gradualmente, baseando-se em soluções para os problemas que vão sendo identificados e nos princípios que norteiam a sua definição.

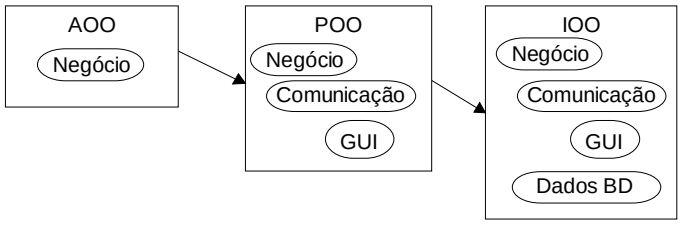
2. Princípios para a Integração

Freqüentemente, o modelo de objetos[3] produzido pela análise e projeto OO serve apenas de base para gerar as tabelas do BDR ou outra estrutura qualquer de armazenamento. Dessa forma, obtém-se uma implementação que, embora utilize uma linguagem OO, não é uma implementação OO pura, no sentido de que omite a definição de objetos de negócio. Além disso, mesmo com a definição destes objetos, a falta de diretrizes adequadas na estruturação da implementação pode resultar na presença de código de dados nos objetos de negócio, mostrando uma falta de preocupação em garantir extensibilidade e reusabilidade.

Por outro lado, a arquitetura proposta neste trabalho foi definida com base no princípio básico de integrar LPOO com BDR sem comprometer fatores de qualidade de software, obtendo os benefícios, tanto de LPOO quanto de BDR: reusabilidade, extensibilidade e eficiência. De fato, a nossa arquitetura torna possível obter uma implementação OO pura e persistente, correspondendo fielmente à análise e projeto OO.

Em particular, para dispor de objetos de negócio reusáveis, é preciso seguir outro princípio: a independência dos objetos de negócio em relação ao mecanismo utilizado para garantir a persistência destes objetos. Assim, a arquitetura proposta neste trabalho sugere a criação de uma camada de software transparente para tratar do armazenamento e manipulação dos objetos no banco de dados, fortalecendo o isolamento das classes de acesso ao mecanismo de persistência.

Além da camada de acesso a dados, o sistema é composto por outras camadas — negócio, GUI e comunicação — que evoluem com o processo de desenvolvimento, conforme ilustrado abaixo:

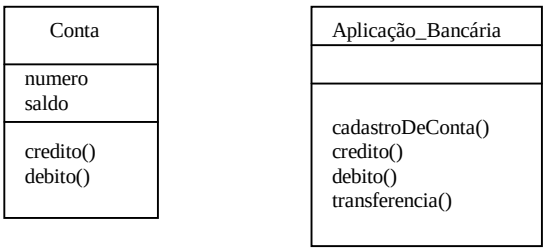


Observe que só na atividade de implementação foi acrescentada a camada “DadosBD”, que contém o código de manipulação de dados persistentes. A integração não é feita, necessariamente, sem causar impacto na camada de negócios, de forma totalmente transparente. Outro princípio seguido é minimizar o impacto causado nessa camada, ou seja, permitir que a camada DadosBD seja introduzida afetando minimamente a camada de negócio, em pontos bem determinados.

3. Integração *ad hoc*

[3] O nome do produto gerado para representar as classes do sistema que está sendo modelado depende da linguagem utilizada. No caso de UML (Unified Modeling Language) [6], usa-se “diagrama de classes”.

Para ilustrar melhor a diferença entre a "Integração *ad hoc*"— freqüentemente utilizada na prática e suportada pelos ambientes de desenvolvimento — e a integração a ser proposta, considere uma aplicação bancária extremamente simples: um banco contém contas, que são identificadas por um número e nas quais se pode creditar ou debitar dinheiro, alterando o valor disponível na conta — o saldo. O escopo do exemplo ficará restrito à classe `Conta` e à classe principal ou fachada[4], `Aplicação_Bancária`, que representa o sistema e funciona como uma classe "servidora" da camada de negócio. Estas classes são apresentadas pelos diagramas de classe[5] abaixo:



Assumindo o mapeamento trivial da classe `Conta` para a tabela `CONTA` do BDR[6]:

CONTA

numero	saldo

pode-se facilmente obter uma implementação simplista baseada na "Integração *ad hoc*", já que diante da inexistência de diretrizes para a realização da integração com o banco de dados, o código de armazenamento e acesso à tabela `CONTA` pode ser inserido diretamente na classe `Aplicação_Bancária`[7]:

```
class Aplicação_Bancária {
  void credito(int num, double val) throws SQLException {
    stmt.executeUpdate("UPDATE CONTA SET saldo=saldo+"+val+"
        WHERE numero="+num);
  }
    ⋮
}
```

onde `SQLException` é a exceção levantada pelos métodos da API `java.sql` quando ocorre algum problema na realização de operações do BDR, e `executeUpdate` é o método da API que recebe como parâmetro uma *string* contendo um comando SQL de inserção, atualização ou exclusão, para ser executado pelo BDR. No exemplo acima, a *string* é formada com a utilização do operador de concatenação de Java, representado pelo símbolo "+".

Note que algumas classes do diagrama de classes não são implementadas como classes de Java, são apenas representadas por tabelas do BDR, onde um registro da tabela contém os valores dos atributos de um objeto da classe correspondente. Assim, creditar

---

[4] A classe fachada é definida pelo padrão de projeto *Facade* [4].
[5] Os diagramas de classe apresentados são expressos em UML.
[6] O mapeamento é feito mapeando classes para tabelas do BDR, uma vez que tanto a classe quanto o conjunto-entidade do MER [7] podem ser vistos como conjuntos de objetos semelhantes.
[7] Por simplicidade, o código dos exemplos é apresentado parcialmente, ressaltando apenas as características que são examinadas no exemplo.

em uma determinada conta resulta em invocar diretamente o método que realiza a operação de atualização correspondente no registro do banco de dados.

3.1. Deficiências da Integração *ad hoc*

Observe no exemplo anterior que a tabela `CONTA` é acessada diretamente através da classe `Aplicação_Bancária`, da mesma maneira que poderia ser acessada de qualquer outra classe do sistema, revelando a inexistência de encapsulamento e *information hiding*, princípios básicos da orientação a objetos. Assim, não há nenhuma preocupação em relação à forma como é estruturada a implementação, tudo se resume a tabelas de banco de dados com atributos que podem ser alterados arbitrariamente em qualquer parte do sistema. Portanto, mudar a estrutura de uma tabela, adicionando ou removendo atributos, ou simplesmente modificando tipos de atributos existentes, causa impacto direto no código do sistema, sem localidade de efeito.

De fato, reusabilidade e extensibilidade são seriamente comprometidas pela integração *ad hoc*, pois não há uma distinção clara entre o código de negócio, que contém regras e objetos de negócio, e o código de dados [13]. O código com as restrições para se cadastrar uma conta ou a semântica das operações de crédito e débito, por exemplo, é misturado ao código que trata da recuperação e armazenamento de dados. Dessa forma, caso seja necessário adaptar o sistema para utilizar outro mecanismo de persistência, tem-se, na verdade, que desenvolver um novo sistema. O mesmo acontece se for necessário adaptar o sistema para trabalhar com outro esquema de dados. A justificativa para isto é que o código de negócio, que poderia ser reusado nas adaptações não é isolado.

Realmente, a inexistência de diretrizes claras para fazer a integração transfere para o programador toda a responsabilidade sobre como estruturar a implementação de um sistema, resultando nos problemas acima. A utilização de ambientes, como JBuilder™, por exemplo, que contém componentes que encapsulam as operacões básicas da camada de acesso a dados não resolve estes problemas, pois é preciso definir claramente como e onde serão requisitados os serviços destes componentes, reforçando a tese de que é mesmo necessário ter uma arquitetura que defina estes aspectos.

3.2. Variações da Integração *ad hoc*

O problema da falta de encapsulamento, identificado no exemplo da Seção 3, pode ser resolvido se os programadores forem disciplinados a ponto de restringir a manipulação da tabela `CONTA` a uma classe `Conta`, por exemplo. Assim, teríamos a seguinte definição da classe `Aplicação_Bancária`:

```
class Aplicação_Bancária {
  void credito(int num, double val) throws DBRuntimeException
  {
    Conta c = Conta.procura(num);
    c.credito(val);
  }
    ⋮
}
```

onde apenas se faz chamadas para que as operações (alteração de atributos) com contas sejam feitas por métodos da classe `Conta`, que tem atributos e métodos de negócio, bem como código de dados.

Esta solução exige um mecanismo que garanta a consistência entre os objetos em memória principal e as tabelas em memória secundária. Para isso, pode-se utilizar uma API de mais alto nível que `java.sql`, como a API JRB que implementa uma cache responsável por controlar a transferência dos dados dos objetos Java na *heap* para o banco de dados e vice-versa.

Temos, então, a seguinte definição da classe `Conta`, utilizando a API JRB:

```
class Conta implements PersistentObject {
  private double saldo;
  private int numero;
  void credito(double v) throws DBRuntimeException {
    saldo = saldo+v;
    this.update();
  }
    ⋮
}
```

onde `implements PersistentObject` indica que a classe `Conta` é persistente e contém os métodos de acesso a dados, declarados na interface[8] `PersistentObject`; `DBRuntimeException` é a exceção levantada pelo método `update` (equivalente ao método executeUpdate do exemplo da Seção 3) quando há algum problema na comunicação com o BDR ou na execução da atualização. `Update` é um método gerado automaticamente pelas ferramentas que acompanham JRB e serve para requisitar ao BDR que atualize o valor dos atributos das tabelas a partir do valor dos atributos dos objetos que estão na cache de JRB. Além do método `update`, são gerados métodos como `access`, `delete`, `markModify`, entre outros que servem para fazer a manipulação direta dos dados persistentes.

Além do que foi visto, é possível obter outras variações da integração *ad-hoc*, sem no entanto, resolver os problemas identificados na Seção 3.1., pois todas apresentam pelo menos alguma das deficiências que analisaremos a seguir.

3.3. Deficiências das Variações da Integração *ad hoc*

Analisando o exemplo da Seção 3.2, pode-se verificar que, em relação ao exemplo do início da Seção 3, tem-se, além do ganho em encapsulamento, um ganho em continuidade e reversibilidade do processo de desenvolvimento, pois as classes da implementação são um reflexo das classes do projeto.

Permanece, entretanto, a deficiência com relação à independência de persistência. A classe de negócio `Conta`, por exemplo, é dependente de operações de acesso banco de dados, como se pode observar na chamada ao método `update`. Assim, mesmo utilizando uma API de mais alto nível, mudar o mecanismo de persistência (de BDR para BDOO, por exemplo) implica em mudar o código de negócio, prejudicando extensibilidade. Também não se pode ter o mesmo código de negócio utilizando formas

[8] Uma interface em Java contém um conjunto de assinaturas de métodos abstratos. Para que uma classe implemente uma interface, primeiro declara a interface numa cláusula `implements` e depois provê o corpo dos métodos da interface [2].

diferentes de armazenar os objetos persistentes. De fato, a dependência entre o código de dados e o código de negócio inibe o potencial de reusabilidade das classes de negócio, fortes candidatas a integrarem uma biblioteca de classes reusáveis, pois reusar o conceito de conta bancária, neste caso, implica em reusar também a maneira de armazenar as contas de forma persistente.

4. Integração Objeto-relacional

A idéia básica da integração objeto-relacional é evitar que o código de armazenamento e recuperação de dados seja misturado ao código de operações e objetos inerentes ao negócio em questão. Para tanto, através da priorização do encapsulamento, propõe-se a separação das classes utilizadas na implementação do sistema em dois tipos[9]:

- Classes para descrição dos objetos de negócio, advindos da modelagem do domínio de problema;
- Classes para manipulação e armazenamento de dados.

A comunicação entre os dois tipos de classes é feita através de interfaces que garantem uma maior independência do código de negócio em relação à forma como efetivamente são implementadas as operações de persistência (o acesso ao banco de dados ou outro mecanismo, como arquivos, por exemplo). A principal motivação é reduzir o impacto causado por modificações no sistema, possibilitando localidade de efeito, tanto para modificações de requisitos funcionais (como a introdução de novos serviços) quanto não-funcionais (como adaptar o sistema para utilizar outro mecanismo de persistência ou melhorar a performance de consultas). Dessa forma, conseguimos resolver também os outros problemas da integração *ad hoc*.

Assim, tem-se uma implementação modular, que pode ser dividida em camadas independentes [13], onde as classes que contêm código de negócio compõem a camada de negócio; as que contêm código de dados, a camada de dados; as que contêm código de comunicação entre subsistemas, a camada de comunicação; e as que contêm código de interface com o usuário, a camada de interface.

A arquitetura para integração objeto-relacional, que especifica melhor os aspectos acima, é apresentada progressivamente, nas subseções a seguir, sendo ao final desta seção descrita graficamente de forma completa.

4.1. Classes Básicas e Classes cadastro

Na integração objeto-relacional, a implementação reflete ainda mais o projeto OO, pois na implementação, assim como no projeto OO, a classe que define um conjunto de objetos é diferente da classe que define um tipo específico de objeto. Contrastando, a classe `Conta` da Seção 3.2 mistura os conceitos de conta e cadastro de contas.

Assim, as classes de negócio são subdivididas em dois tipos: classes básicas e classes cadastro (ou histórico, log, etc.). As classes básicas representam objetos básicos de negócio. As classes cadastro representam agrupamentos destes objetos básicos, tendo como operações a inserção, busca e exclusão de elementos do cadastro, verificações ou testes de pré-condições relativos a estas manipulações e mais as operações que invocam operações típicas dos objetos de negócio como crédito e débito. Considerando o

[9] Não considerando o código da GUI e de comunicação com outros subsistemas.

exemplo da aplicação bancária, as classes básicas são `Conta, Cliente` e `Extrato,` entre outras, enquanto uma classe cadastro é `CadastroDeContas,` descrita na Seção 4.1.2.

4.1.1. Classes Básicas

Seguindo a integração objeto-relacional, a classe `Conta` é de fato extensível e reusável, pois espelha mais fielmente o resultado do projeto OO:

```
class Conta implements PersistentObject {
  private double saldo;
  private int numero;
  void credito(double v) { saldo=saldo+v; }
    ⋮
}
```

Contrastando com a classe `Conta` da Seção 3.2., o método `credito` acima contém apenas código de negócio, não invocando métodos de armazenamento de dados. Como na Seção 3.2, a classe `Conta` acima implementa `PersistentObject`, e contém os métodos gerados automaticamente pelas ferramentas de JRB, o que a torna dependente desta API[10]. Entretanto, esta porção de código de dados, dependente da API, pode ser facilmente removida, de forma automática, sem impactar a camada de negócio, pois o código de dados não é invocado pelos métodos de negócio, como ilustra a operação `credito` acima.

Já a classe `Aplicação_Bancária`, por sua vez, contém um cadastro de contas como atributo, ou seja, um objeto da classe `CadastroDeContas`, representando as contas cadastradas no banco. Esta classe apenas invoca os métodos dos cadastros que contém:

```
class Aplicação_Bancária {
  private CadastroDeContas contas;
    ⋮
  void credito(int num, double val) throws
      ContaInexistenteException, RepositorioException {
    contas.credito(num,val);
  }
}
```

onde as exceções acima são oriundas do método `credito` de `CadastroDeContas,` indicando, respectivamente, que não existe uma conta de número "`num`", e que o repositório utilizado para armazenar as contas não está disponível[11].

4.1.2. Classes Cadastro

[10]Os métodos gerados automaticamente (`update, access`, etc.) foram omitidos por simplicidade, pois já foram apresentados na Seção 3.2.

[11] Isto pode ocorrer pelas seguintes razões, dentre outras: a conexão com o mecanismo de persistência não está ativa, a transação corrente é apenas para leitura, ou a classe não é persistente.

A classe cadastro contém um atributo repositório, passado como argumento para o construtor e, cujas implementações utilizam diferentes mecanismos para armazenamento de dados. É na classe cadastro que está o código de negócio dos métodos de um cadastro. Assim, a classe `CadastroDeContas` é definida da seguinte forma:

```
class CadastroDeContas {
  private RepositorioDeContas contas;
  void credito(int num, double val) throws ContaInexistenteException,
      RepositorioException {
     Conta c = contas.procura (num);
     c.credito(val);
     contas.atualiza(c);
  }
    ⋮
}
```

onde `RepositorioDeContas` é a interface que define as operações que são implementadas pela camada de dados[12]; `ContaInexistenteException` é levantada pelo método `procura`; `RepositorioException` é uma exceção levantada pelo método `atualiza`; e o comando "`contas.atualiza(c)`" garante que o estado do objeto `Conta` seja refletido para o mecanismo de persistência.

A classe `CadastroDeContas` é responsável, também, por atividades como, testar se a conta é válida ou já existente e invocar o método da interface repositório que é responsável pela inserção da conta na estrutura de armazenamento:

```
void insere(Conta c) throws ContaJaExisteException,
    ContaInvalidaException, RepositorioException {
  if (c!=null){
    if (contas.existe(c.numero())) throw new ContaJaExisteException();
    else contas.insere(c);
  } else throw new ContaInvalidaException();
}
```

Observe que tanto no código de armazenamento ilustrado acima no método `insere`, quanto no código de recuperação de contas ilustrado abaixo no método `procura`, o serviço necessário é solicitado a um objeto de uma classe que implementa `RepositorioDeContas`:

```
Conta procura(int num) throws RepositorioException{
  Conta c = contas.procura(num);
  return c;
}
```

É para que seja possível manter a modularidade e a separação entre as camadas, que apenas as classes da camada de dados devem ser responsáveis pelos detalhes sobre como será implementada a estrutura e as operações de acesso e armazenamento, seja usando banco de dados relacional, arquivos, estruturas voláteis na memória principal, ou mesmo banco de dados OO.

De fato, para que se possa ter a classe `CadastroDeContas` independente da estrutura de armazenamento, dois componentes importantes são:

[12] Esta interface é explicada detalhadamente na Seção 4.2.

- as classes repositório, da camada de dados e;
- as interfaces entre as classes da camada de dados (classes repositório) e da camada de negócios (classes cadastro).

Estes componentes são detalhados a seguir.

4.2. Interfaces Repositório

Como veremos depois, as classes repositório possuem sempre um perfil específico: contêm os métodos de manipulação dos objetos na estrutura de armazenamento. Assim, propomos a definição de interfaces repositório estabelecendo o padrão de comunicação entre os objetos das classes cadastro e os objetos das classes repositório, através das assinaturas de métodos para inserção, atualização, consultas e exclusão nos repositórios. A utilização da interface proporciona extensibilidade, uma vez que torna explícito o fato de que, seja qual for a estrutura de armazenamento usada, a camada de negócio necessita apenas que as operações declaradas na interface sejam implementadas em uma classe repositório.

Voltando ao exemplo da aplicação bancária, note que a classe `CadastroDeContas` livra-se da tarefa de manipulação de dados, solicitando a objetos de `RepositórioDeContas` que o façam. A interface `RepositórioDeContas` é definida da seguinte forma:

```
interface RepositórioDeContas {
  void insere(Conta conta) throws RepositorioException;
  void atualiza(Conta conta) throws RepositorioException;
  void deleta(int num) throws RepositorioException;
  boolean existe (int num) throws RepositorioException;
  Conta procura(int num) throws RepositorioException;
    ⋮
}
```

O método `atualiza` faz sentido apenas em implementações persistentes, uma vez que é responsável por garantir que o estado do objeto em memória principal seja atualizado na estrutura de armazenamento sendo utilizada.

Também é importante que haja flexibilidade para que, numa fase de otimização, seja possível identificar operações críticas e implementá-las de forma diferenciada, caso isto seja realmente necessário. Além disso, é interessante disponibilizar operações que permitam realizar pesquisas seqüenciais, retornando objeto a objeto, como:

```
  void iniciabusca() throws RepositorioException;
  Conta proximoObjeto() throws RepositorioException;
  void finalizabusca() throws RepositorioException;
```

4.2.1. Eficiência

Um dos requisitos básicos que a arquitetura para integração objeto-relacional deve atender é a eficiência. Para que se possa ter uma solução de compromisso, que obedeça aos princípios de OO, mas que não comprometa a boa performance possibilitada pela utilização de BDR, a interface repositório pode ser acrescida de operações que realizem consultas mais específicas, como

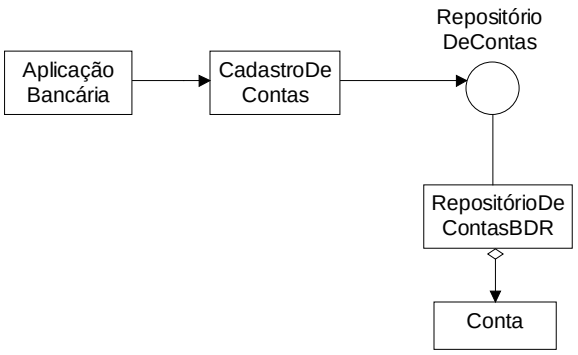
```
Conta selecionaMaiorSaldo() throws RepositorioException;
```

que retorna a conta com o maior saldo. Em particular, esta operação pode ser implementada de forma bem mais eficiente com uma consulta SQL do que com comandos em Java.

4.3. Classes Repositório

No nível mais baixo da arquitetura proposta estão as classes repositório, com o código de manipulação da estrutura de armazenamento: tabelas do BDR, neste artigo. Estas são as classes que conhecem a estrutura de armazenamento e dependem dela, de tal forma que mudanças nestas estruturas causam impacto direto nestas classes e só nelas, isolando as outras camadas do sistema. A classe `RepositórioDeContasBDR` precisa, portanto, saber que o objeto do tipo `Conta` deve ser inserido na tabela `CONTA` e que as colunas desta tabela são `numero` e `saldo`[13].

O diagrama de classes, incluindo a interface e a classe repositório é apresentado abaixo:



A classe `RepositorioDeContasBDR` é definida da seguinte maneira[14]:

```
class RepositóriodeContasBDR implements RepositorioDeContas{
  void insere(Conta c) throws RepositorioException{
    try{
      c.update();
    } catch (DBRuntimeException e) {
      throw new RepositorioException();
    }
  }
    ⋮
}
```

onde "`c.update()`" é a chamada ao método da classe `Conta` que é gerado automaticamente por JRB e serve para incluir o objeto na base, ou atualizá-lo, caso já

[13] A dependência em relação aos nomes das estruturas de armazenamento pode ser contornada através da leitura dessas informações a partir de um arquivo de configuração, e da passagem desses nomes como argumentos para o construtor da classe repositório.

[14] Por simplicidade, não incluímos nos exemplos nenhuma referência a início, confirmação e aborto de transações.

tenha sido armazenado. Na Seção 3.2., a chamada ao método update estava na própria classe Conta, tornando-a dependente do código da API. A exceção `DBRuntimeException` é levantada pelo método `update` para indicar que a operação de inserção ou atualização no BDR não pôde ser realizada com sucesso. Note que `DBRuntimeException` é capturada, sendo levantada `RepositorioException` em seu lugar. Isto é feito para que na classe cadastro, que pertence à camada de negócio não haja uma dependência em relação aos métodos de exceção da API.

No construtor desta classe estão as informações sobre a conexão com a base de dados a ser acessada:

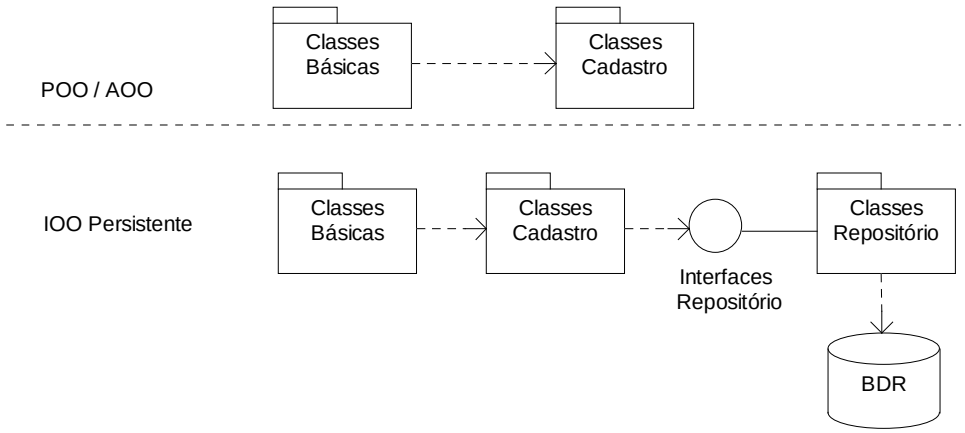
```
void RepositorioDeContasBDR(String tipoSistema, String nomeServidor,
   String login, String senha, String classeDoDriver) throws
     RepositorioException{
 Database bd = new Database(tipoSistema,nomeServidor,
     login,senha );
 try {
  if (classeDoDriver!=null) Class.forName(classeDoDriver);
  bd.connect(nomeServidor);
  bd.open(nomeServidor);
 }catch(Exception e) {
  throw new RepositorioException();
 }
}
```

onde `Database` é a classe de JRB que implementa as operações de banco de dados, como abrir (`open`), fechar, conectar (`connect`) e desconectar. Esta classe recebe alguns argumentos que podem ser lidos de um arquivo de configuração e são passados pela classe fachada (no caso, `Aplicação_Bancária`) à classe cadastro e desta à classe repositório. Destes argumentos, o `tipoSistema` indica se é usado banco de dados Oracle, Sybase ou acesso via JDBC; o `nomeServidor`, no caso de JDBC indica a URL (*Uniform Resource Locator*), que funciona como um endereço para localizar a base de dados. Caso o `tipoSistema` seja JDBC, é preciso indicar qual a classe do driver JDBC [14] que está sendo utilizado.

Da forma apresentada acima, mudanças na camada de dados, como trocar o mecanismo de persistência ou mesmo acessar mais de uma base de dados têm um impacto localizado nesta classe, principalmente no construtor.

4.4. Visão Geral da Arquitetura da Integração Objeto-relacional

Agora que os componentes da arquitetura para integração objeto-relacional já foram discutidos separadamente, é possível agrupá-los em uma descrição gráfica, utilizando a notação de pacotes de UML:



5. Conclusões

Apresentamos uma arquitetura para integração de Java com BDR. A grosso modo, esta arquitetura estrutura o sistema de maneira a obter camadas independentes de código, permitindo separar código de negócio do código de dados. A utilização da arquitetura proporciona uma série de vantagens: independência dos objetos de negócio em relação ao mecanismo de persistência e à API utilizada para possibilitar a integração, continuidade em relação ao projeto OO, obtenção de objetos reusáveis, e localidade de efeito na mudança de detalhes do esquema de dados.

A classificação das classes do sistema — em classes básicas, classes cadastro, interfaces repositório e classes repositório — e a delimitação do escopo de serviços implementados em cada uma destas classes, possibilitam o projeto e construção de métodos e ferramentas que possam, respectivamente, orientar e automatizar parte do trabalho do programador na implementação de sistemas persistentes. De fato, um método está sendo desenvolvido com o objetivo de construir sistemas de informação baseados nesta arquitetura [5].

Embora os exemplos apresentados neste trabalho, para ilustrar a arquitetura tenham sido construídos utilizando a linguagem Java, a arquitetura é bastante genérica, podendo ser facilmente aplicada a outras linguagens de programação orientadas a objetos, obtendo os benefícios discutidos aqui. Além disso, apesar do objetivo inicial ter sido integrar Java com BDR, a arquitetura proposta para realizar esta integração resolve parte dos problemas que temos para integrar Java com BDOO, arquivos indexados e outros mecanismos de persistência.

A arquitetura proposta foi colocada à prova, não só em exemplos de cunho acadêmico, mas também em aplicações práticas de domínios diferentes. Os sistemas abaixo foram desenvolvidos utilizando a arquitetura, que, na realidade, foi sendo aperfeiçoada de acordo com os problemas identificados ao longo dos desenvolvimentos:

- SGC (Sistema de Gerência de Clientes), sistema de informação para que clientes solicitem, via Internet, serviços em telefones celulares;
- VAREJO[15], sistema de automação de frente de loja de supermercados de pequeno e grande porte.

[15] O nome real não pode ser revelado por motivos comerciais, já que o sistema ainda está sendo desenvolvido.

O primeiro sistema foi implementado com BDR, enquanto o segundo utiliza o BDOO PSE™ da ObjectDesign.

Agradecimentos

A Marconi Lima e Denise Neves, pela ajuda no entendimento e na identificação dos problemas encontrados na aplicação de uma versão preliminar da arquitetura no desenvolvimento do SGC.

Referências Bibliográficas

[1] Bertrand Meyer. Object-Oriented Software Construction. Prentice-Hall, second edition, 1997.

[2] David Flanagan. Java in a Nutshell, O'Reilly & Associates, second edition, 1997.

[3] Denise Neves. Desenvolvimento de Software como um Processo Contínuo e Reversível uando BON e Java. Tese de Mestrado, Departamento de Informática, Universidade Federal de Pernambuco, Recife, Dezembro de 1997.

[4] Erich Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[5] Euricelia Viana. Um Método para Implementação Orientada a Objetos Persistente usando Java. Tese de Mestrado em desenvolvimento, Departamento de Informática, Universidade Federal de Pernambuco, Recife, 1999.

[6] Grady Booch, James Rumbaugh e Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1998.

[7] Henry Korth e Abraham Silberschatz. Sistema de Banco de Dados. Makron Books, segunda edição, 1993.

[8] James Gosling, Bill Joy e Guy Steele. The Java Language Specification. Addison-Wesley, 1996.

[9] James Rumbaugh e outros. Object-Oriented Modelling and Design. Prentice-Hall International Editions, 1991.

[10] Kim Waldén e Jean-Marc Nerson. Seamless Object-Oriented Software Archictectures: Analysis and Design of Reliable Systems. Prentice Hall, 1995.

[11] O2 Technology User Manual: Java Relational Binding. Version 2.0. July, 1997.

[12] R. Fichman and C. Kemerer. Object-oriented and conventional analysis and design methodologies: comparison and critique. IEEE Computer, vol 25, n° 10, 1992, pp. 23-29.

[13] Scott Ambler. Building Object Applications that Work. Cambridge University Press and Sigs Books, 1998.

[14] Sun Microsystems Package java.sql. http://www.javasoft.com/products/jdk/1.1/docs/api/Package-java.sql.html.