

MAC0439 - Laboratório de Bancos de Dados

Aula 20 - Parte 2

Padrões Arquiteturais para o Acesso a BDs

9 de novembro de 2016
Prof^a Kelly Rosa Braghetto

Primórdios das Arquiteturas em Camadas

- ◆ Anos 90 – primeiros sistemas “em camadas”
 - ◆ Apenas 2 camadas: cliente – servidor
 - ◆ Cliente = interface com o usuário
 - ◆ Servidor = BD relacional
- ◆ Ferramentas para o desenvolvimento dos clientes: Visual Basic, PowerBuilder, Delphi
- ◆ Facilitavam a construção de aplicações de manipulação intensiva de dados
- ◆ Permitiam que controles fossem arrastados para uma área de desenho da interface, e que depois fossem conectados a elementos do BD

Problema da Arquitetura Cliente-Servidor

- ◆ Onde embutir a lógica do domínio?
 - ▶ Regras de negócio, validações, cálculos, etc.
- ◆ Geralmente, ficavam no código do cliente
 - ▶ Lógica era embutida nas telas da interface
 - ▶ Precisava ser replicada em diferentes telas → manutenção difícil

Problema da Arquitetura Cliente-Servidor

- ◆ Alternativa: embutir lógica do negócio no servidor → no BD, por meio de *stored procedures*
- ◆ Essa estratégia não é muito bem aceita devido as características das linguagens para *stored procedures*
 - ▶ São mais pobres que linguagens de programação convencionais
 - ▶ Não são padronizadas; são específicas para um SGBD e impedem que o BD possa ser “portado” a um baixo custo

Solução: Arquitetura em 3-Camadas

- ◆ Contexto: disseminação das LPOOs e o crescimento da Web
- ◆ Componentes:
 - ▶ **Camada de apresentação:** para as interfaces com o usuário
 - ▶ **Camada de domínio:** para a lógica do negócio
 - ▶ **Camada de fonte de dados** (*data source*)

Um Parênteses sobre Objetos...

Em linguagens de programação OO:

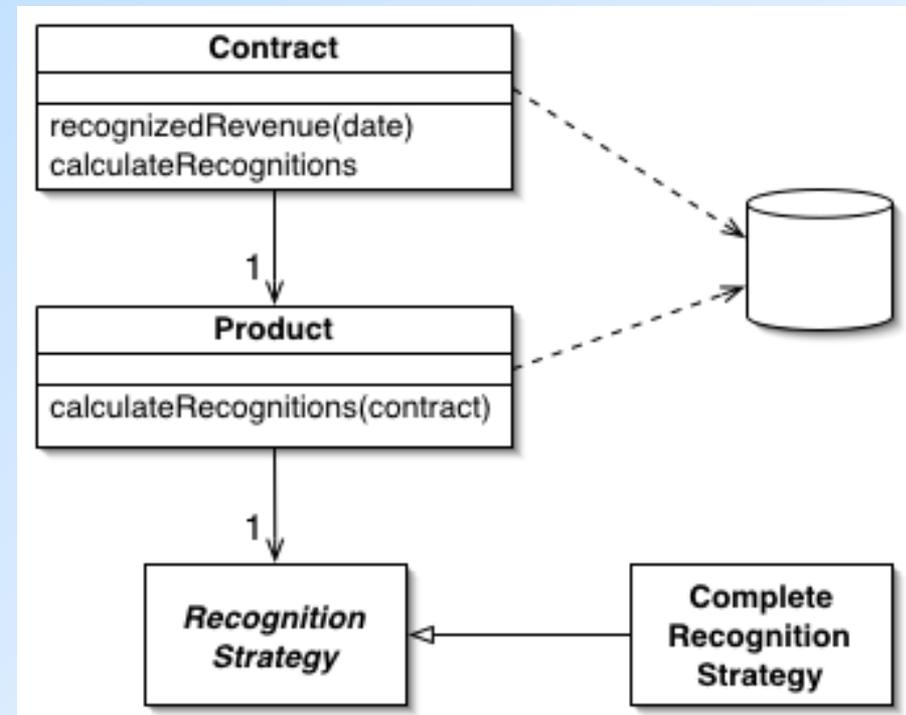
- ◆ **Objeto** é uma instância de uma classe
 - ▶ Objetos têm estados (= valores para seus atributos)
 - ▶ Objetos têm comportamentos (= métodos)
- ◆ **Modelo de objetos** é uma coleção de todas as definições de classes de uma aplicação
- ◆ As classes podem representar:
 - ▶ Elementos de interface do usuário
 - ▶ Recursos do sistema
 - ▶ Eventos da aplicação
 - ▶ **Abstrações dos conceitos de negócio**

Objetos de Negócio

- ◆ Exemplos de objetos que abstraem conceitos de negócio:
 - ▶ Num sistema de processamento de pedidos:
 - Cliente, Pedido e Produto
 - ▶ Numa aplicação financeira
 - Cliente, Conta, Crédito, Débito
- ◆ Esses objetos modelam o domínio do negócio onde a aplicação específica irá operar → são chamados de **Modelo de Domínio**

Padrão Modelo de Domínio

- ◆ Um objeto do modelo de domínio incorpora tanto dados quanto comportamento
- ◆ Um modelo de domínio cria uma rede de objetos interconectados, onde cada objeto representa algum indivíduo significativo
 - ▶ Que pode ser tão grande quanto uma corporação ou tão pequeno quanto uma linha em um formulário de pedido



Persistência de Objetos

- ◆ Os objetos do modelo de domínio representam os principais estados e comportamentos da aplicação
- ◆ Geralmente, esses objetos:
 - ▶ São compartilhados por vários usuários simultaneamente
 - ▶ São armazenados e recuperados entre as execuções da aplicação
- ◆ Persistência de objetos - capacidade desses objetos de “sobreviverem” além do tempo de execução da aplicação
- ◆ Objetos podem ser persistidos em diferentes tipos de fontes de dados (*data sources*)
- ◆ **Obs.:** a persistência não é exclusividade dos objetos de domínio (mas é um requisito mais frequente para eles)

Arquiteturas para a Comunicação com as Fontes de Dados

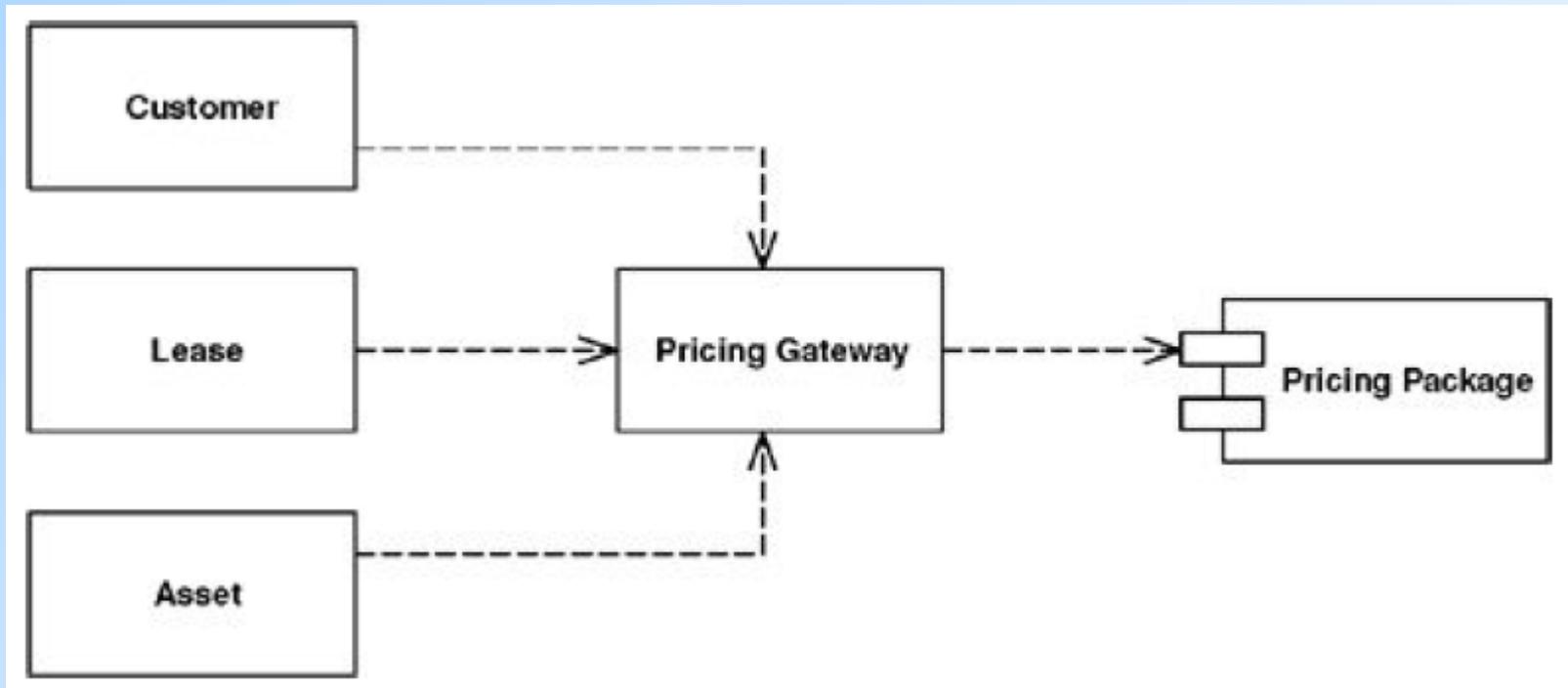
- ◆ Um dos principais papéis da camada de fonte de dados é estabelecer a comunicação entre o sistema e o BD
- ◆ Existem diferentes padrões arquiteturais no que se refere à forma como a lógica do domínio “conversa” com o BD
- ◆ A escolha de uma arquitetura para a comunicação com o BD é uma etapa crítica no projeto de um sistema, já que é algo muito complicado de se refatorar depois

Arquiteturas para a Comunicação com as Fontes de Dados

- ◆ Recomendado: separar código em SQL do código da lógica do negócio, colocando-os em classes diferentes
 - ▶ Vantagem dessa separação: código SQL concentrado em um só lugar evita replicação, facilita a manutenção, etc.
- ◆ Organização das classes pode se basear na estrutura do BD
 - ▶ Ex.: uma classe por tabela → *gateway* para a tabela

Gateway

- ◆ É um objeto que encapsula o acesso a um recurso ou sistema externo



Tipos de *Gateways* para Acesso ao BD

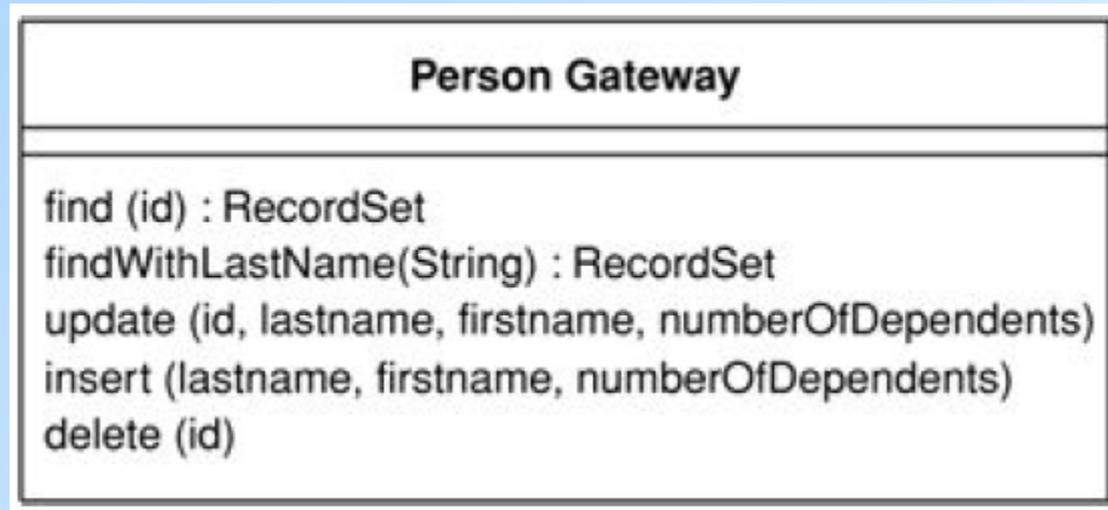
- ◆ **Gateway de linha de dados** (*row data gateway*)
 - um objeto que atua como um *gateway* para uma única tupla em uma fonte de dados
 - ▶ Existe uma instância por tupla
- ◆ **Gateway de tabela de dados** (*table data gateway*) - um objeto que atua como *gateway* para uma relação (tabela ou visão) do BD
 - ▶ Uma única instância (do tipo conjunto de registros) manipula todas as tuplas da relação
 - ▶ **Conjunto de registros** (*dataset*)- estrutura genérica de dados para a representação de tabelas e linhas (que imita a estrutura de um BD)

Gateway de Linha de Dados

Person Gateway
lastname firstname numberOfDependents
insert update delete <u>find (id)</u> <u>findForCompany(companyID)</u>

- ◆ Um *gateway* de linha de dados é um objeto com uma estrutura semelhante a tupla armazenada no BD, mas que pode ser acessada por meio dos mecanismos regulares de uma linguagem de programação
- ◆ Todos os detalhes do acesso a fonte de dados ficam escondidos atrás dessa interface

Gateway de Tabela de Dados



- ◆ Um gateway de tabela de dados armazena todo o SQL para acessar uma única relação: seleção, inserção, atualização e remoção
 - ▶ Outros códigos chamam seus métodos para todas as interações com o banco de dados

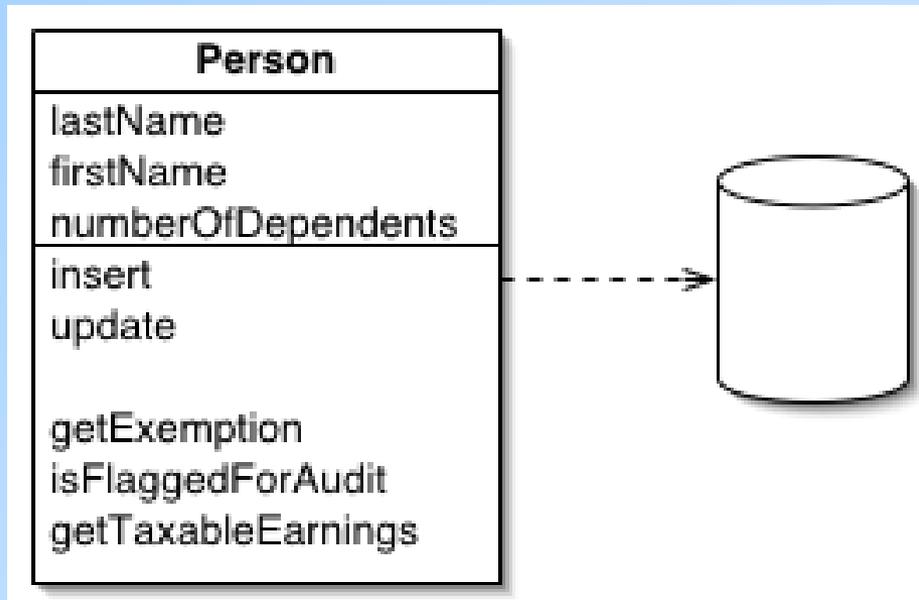
Outros Padrões de Acesso/Mapeamento

- ◆ Em aplicações que usam modelo de domínio, outras opções de mapeamento podem ser mais apropriadas
 - ▶ Dependendo do modelo de domínio e da estrutura da fonte de dados, os *gateways* podem ter complexidade **demais** ou **de menos**
- ◆ Exemplos de padrões de mapeamento usados com modelo de domínio:
 - ▶ ***Active Record***
 - ▶ ***Data Mapper***

Active Record

- ◆ Em aplicações com modelo de domínio simples:
 - ▶ Estrutura do modelo de domínio se assemelha bastante à do BD, com uma classe do domínio por tabela do BD
 - ▶ Objetos do domínio possuem lógica de negócio de complexidade moderada
 - ▶ Nesse contexto, é viável que cada objeto do domínio se ocupe pelo carregamento e pelo salvamento de dados no BD → esse padrão de arquitetura é chamado de **Registro Ativo (Active Record)**

Active Record



- ◆ Um objeto que encapsula uma tupla de uma relação do BD, encapsula o acesso ao BD e adiciona lógica de domínio aos dados
- ◆ Um objeto carrega tanto dados quanto comportamento
- ◆ A maior parte dos dados é persistente e precisa ser armazenada em um BD

O Padrão *Active Record* “na Prática”

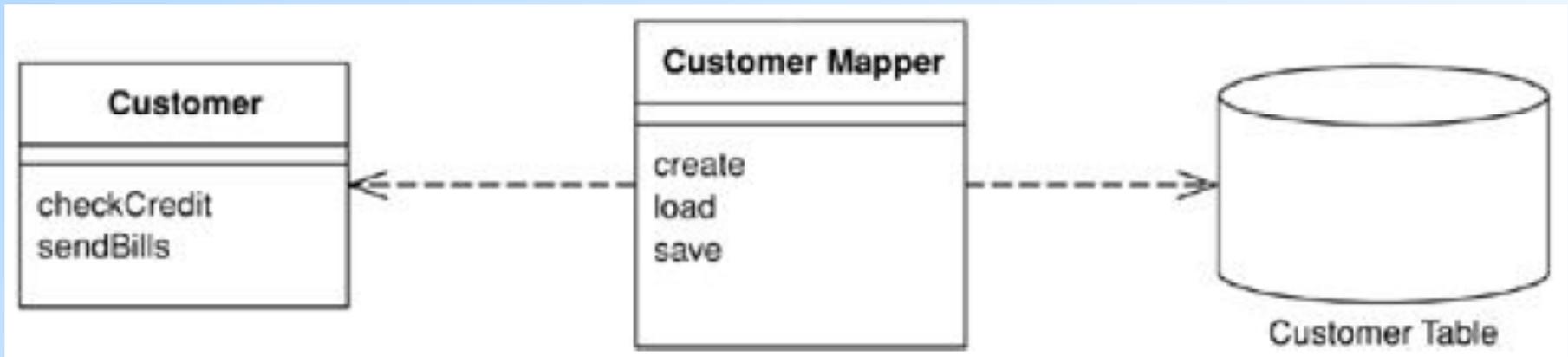
- ◆ ActiveRecord (Ruby)
- ◆ PHP ActiveRecord (PHP)
- ◆ ActiveJDBC (Java)
- ◆ DBIC (Perl)
- ◆ Objective-C, Python, ...

Mas o *Active Record* Não É Suficiente Quando...

- ◆ A lógica de domínio é mais complicada → modelo de domínio complexo
 - ◆ Não há um mapeamento 1-para-1 entre as classes do modelo e as tabelas do BD
 - ◆ Existe a necessidade de se testar a lógica de negócio sem que o acesso ao BD seja feito o tempo todo
- Nesses casos, nem mesmo a indireção do padrão Gateway é suficiente para lidar com a complexidade
- ◆ Alternativa: padrão ***Data Mapper***

Data Mapper

- ◆ É uma camada de mapeadores que transferem dados entre objetos de domínio e um BD, mantendo-os independentes entre si
- ◆ Essa é a arquitetura de mapeamento mais complicada, mas que garante isolamento entre as duas camadas
 - ▶ Tanto o modelo de domínio quanto o de BD podem variar sem que um afete o outro



O Padrão *Data Mapper* “na Prática”

- ◆ MyBatis, para Java
- ◆ SQLAlchemy, para Python
- ◆ Ruby Object Mapper (ROM), para Ruby
- ◆ Doctrine, para PHP
- ◆ ...

Mapeamento Objeto - Relacional

- ◆ A maioria dos projetos de desenvolvimento de software usa:
 - ▶ Uma linguagem OO (como, Java, C#, Ruby, etc.)
 - ▶ Um BD relacional para armazenar dados
- ◆ Problema: **incompatibilidade conceitual** (*impedance mismatch*) entre objetos e relações
- ◆ Solução em potencial: uso de **Bancos de Dados de Objetos** (antes chamados de BDs Orientados a Objetos)
 - ▶ Entretanto, não há implementações de SGBDOOs usados em grande escala
 - ▶ Mas muitos SGBDs se auto-denominam Objeto-Relacionais: PostgreSQL, Oracle, ...

Implementando os Padrões “na Raça”, em Java

- ◆ Classes do domínio → implementados como *Java Beans*
- ◆ *Java Bean* - classe em Java que possui um construtor sem argumentos e métodos *get* e *set* para o acesso aos atributos
- ◆ Exemplo:

```
create table contatos (  
  id serial,  
  nome VARCHAR(255),  
  email VARCHAR(255),  
  endereco VARCHAR(255),  
  dataNascimento DATE,  
  primary key (id)  
);
```

```
public class Contato {  
    private Long id;  
    private String nome;  
    private String email;  
    private String endereco;  
    private Calendar dataNascimento;  
  
    // métodos get e set para id, nome, email, ...  
    public String getNome() {  
        return this.nome;    }  
    public void setNome(String novo) {  
        this.nome = novo;    }  
    public String getEmail() {  
        return this.email;    }  
    public void setEmail(String novo) {  
        this.email = novo;    }  
    ...  
}
```

Um Padrão para Controlar as Conexões com o BD: Fábrica de Conexões

- ◆ Uma fábrica de conexões se ocupa da abertura de uma conexão com o BD sempre que for necessário. Exemplo em Java + JDBC:

```
public class FabricaDeConexao {
    public Connection obterConexao() {
        try {
            return DriverManager.getConnection(
                "jdbc:postgresql://data.ime.usp.br:23001/bd_teste",
                "usuario", "senha");
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

- ◆ Sempre que uma nova conexão for necessária:

```
Connection con = new FabricaDeConexao().obterConexao();
```

Fechando uma Conexão

```
public static void main(String[] args) throws SQLException {
    Connection con = null;
    try {
        con = new FabricaDeConexao().obterConexao();
        // faz algumas operações sobre a conexão
        // que podem gerar exceções (SQLException)
    } catch (SQLException e) {
        System.out.println(e);
    } finally {
        con.close(); // A conexão será fechada mesmo
    } // em caso de exceção no bloco "try"
}
```

Fechando uma Conexão com Java ≥ 7

- ◆ É possível se beneficiar da estrutura *try-with-resource*
<http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>
- ◆ Nesse tipo de “try”, é possível declarar e inicializar objetos que implementam *java.lang.AutoCloseable*
 - ◆ Ao término do “try”, o *close* do objeto será invocado automaticamente

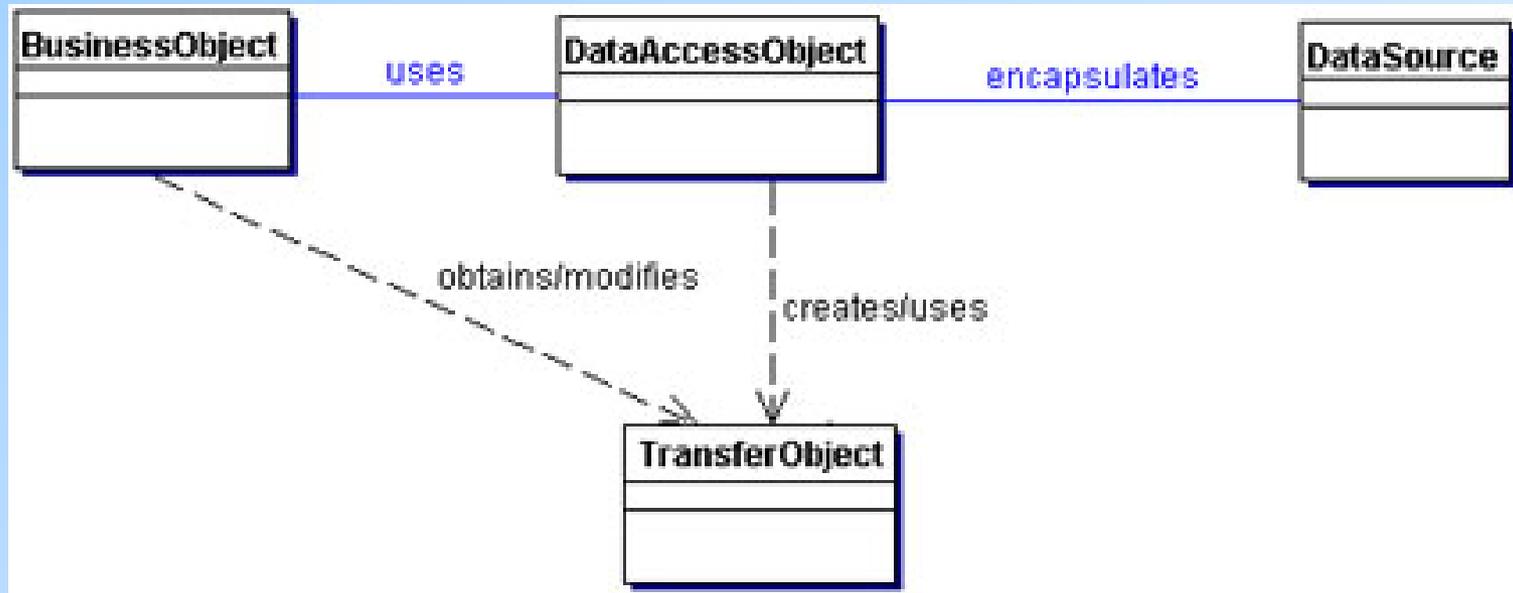
◆ Exemplo:

```
try (Connection con = new FabricaDeConexao().obterConexao())
{
    // faz um monte de operações sobre a conexão
    // que podem gerar exceções
catch(SQLException e) {
    System.out.println(e);
}
```

Padrão DAO

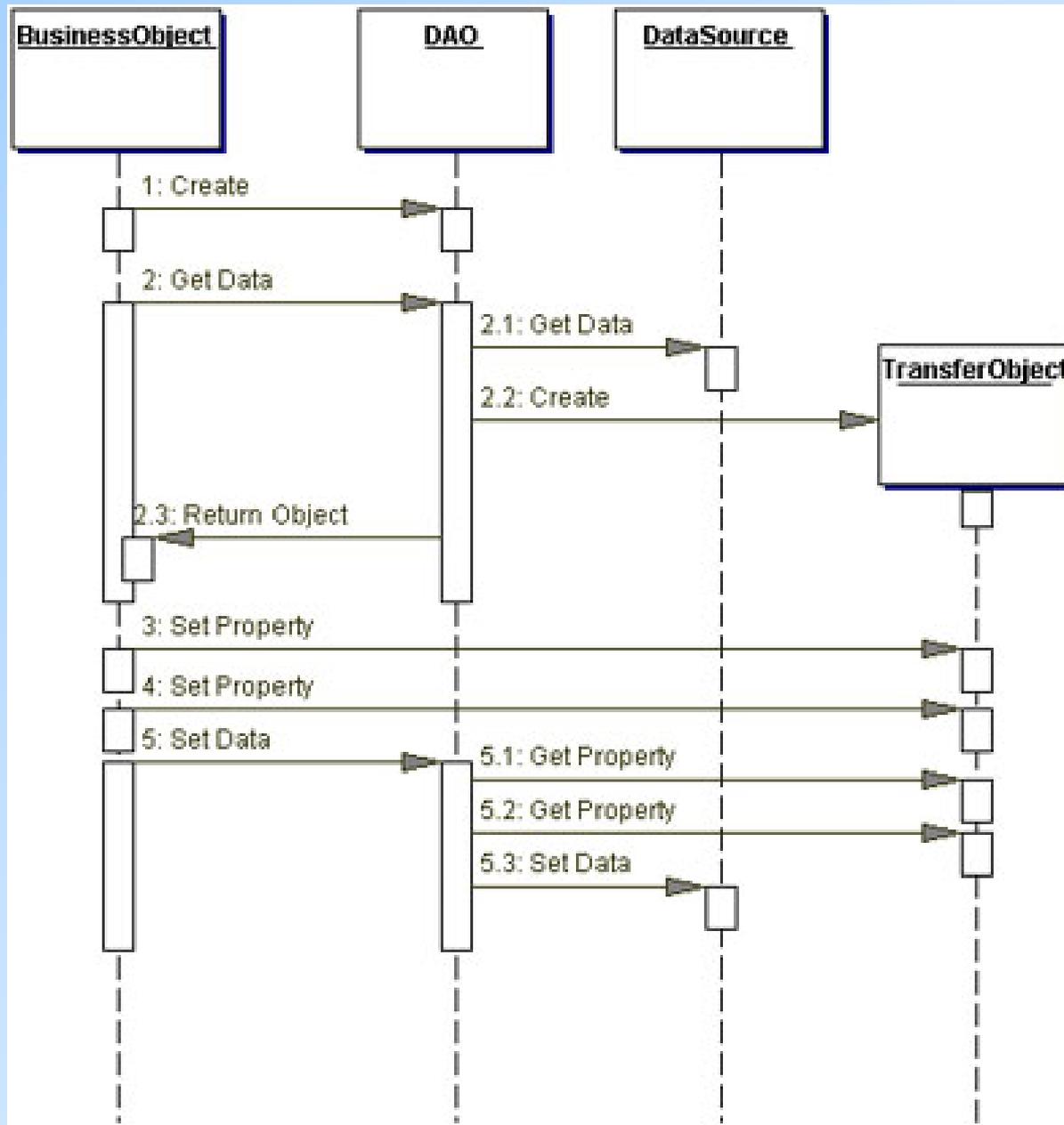
- ◆ **Data Access Object (DAO)** – objeto de acesso aos dados
- ◆ Muito usado em aplicações Java
 - ▶ Origem: “Core J2EE Patterns”
- ◆ “O DAO abstrai e encapsula todos os acessos a uma fonte de dados. Ele gerencia a conexão com a fonte de dados para obter e armazenar dados”
- ◆ Provê uma interface abstrata para algum tipo de mecanismo de persistência
 - ▶ SGBDRs, SGBOO, XML, etc.
 - ▶ Provê operações sobre os dados sem expor detalhes de implementação da fonte de dados
- ◆ Equivale ao padrão **Data Mapper!**

DAO - Diagrama de Classes

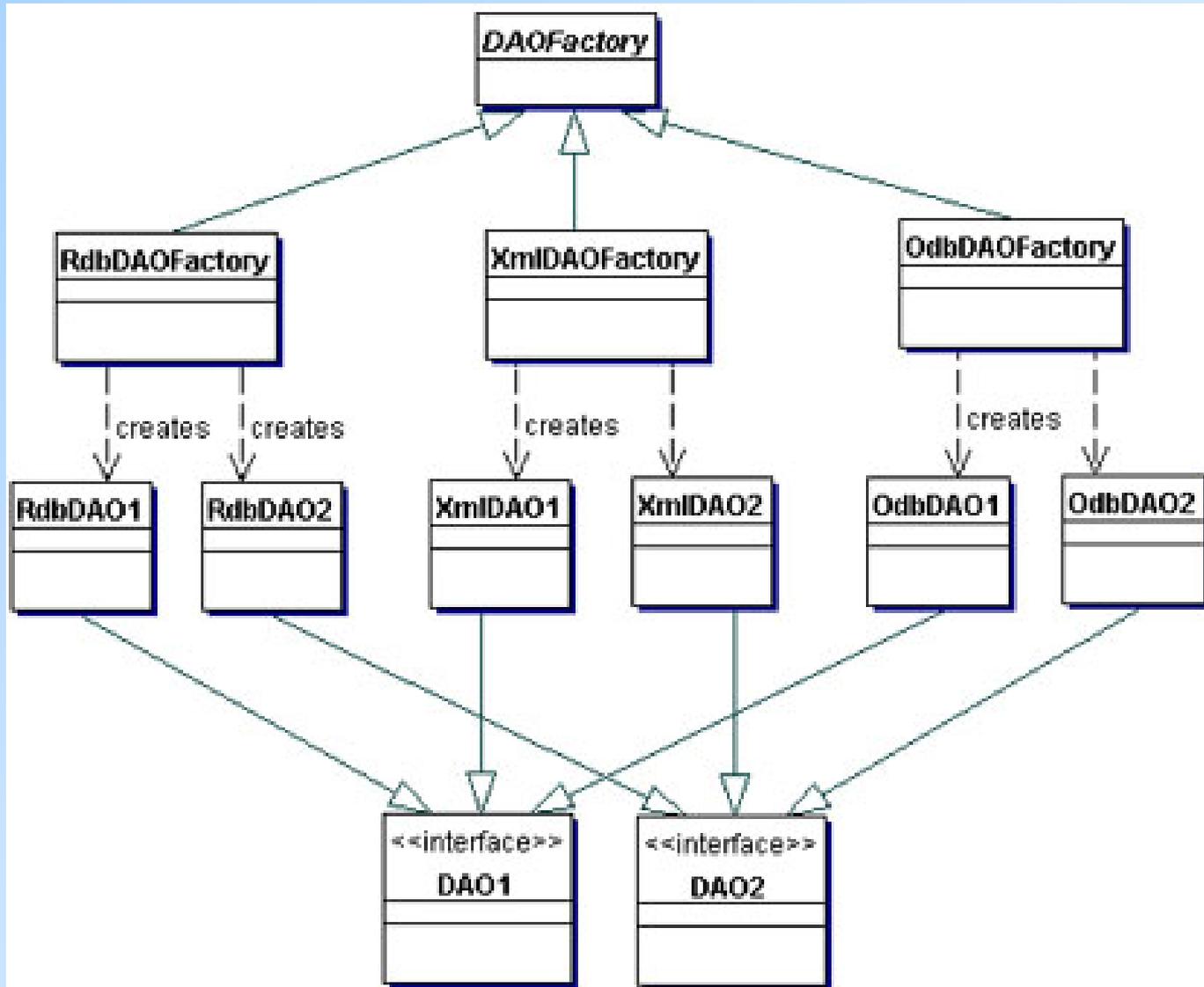


- ◆ **BusinessObject** - objeto “cliente”, que requisita o acesso aos dados
- ◆ **DataAccessObject** - abstrai a implementação do acesso aos dados para o BusinessObject
- ◆ **DataSource** - representa a implementação de uma fonte de dados
- ◆ **TransferObject** - usado para “carregar” dados (obtidos da fonte de dados) para o BusinessObject

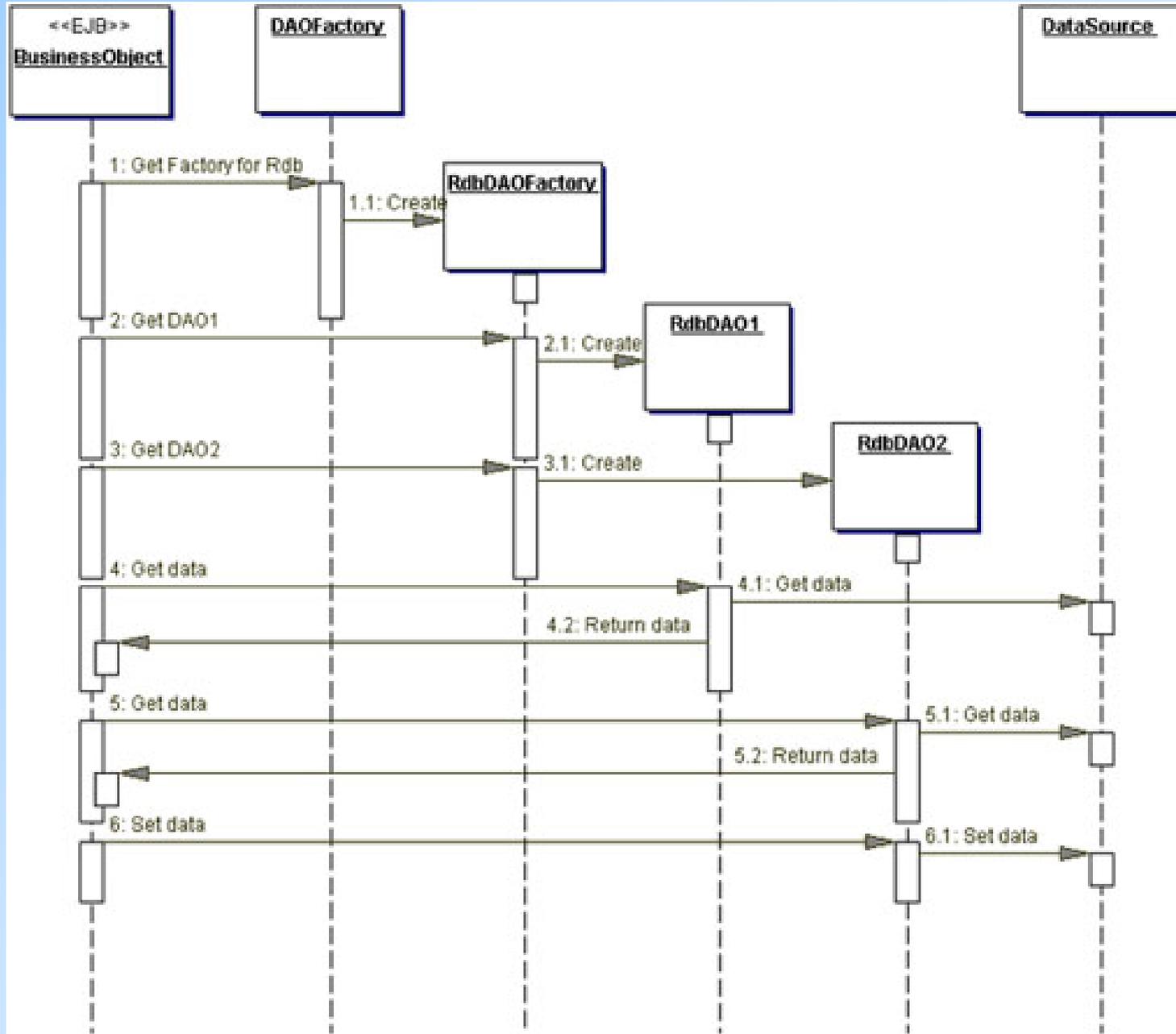
DAO - Diagrama de Sequência



Fábrica de DAOs



Fábrica de DAO - Diagrama de Sequência



Exemplo de uma Implementação Simples do DAO em Java

- ◆ Exemplo1, disponível no Paca
 - ▶ Adaptado do exemplo dado em:

<http://www.caelum.com.br/apostila-java-web/bancos-de-dados-e-jdbc/>

Exemplo de Implementação do *Data Mapper* em Java

- ◆ Exemplo2, disponível no Paca
 - ▶ Baseado nos exemplos contidos no livro “*Patterns of Enterprise Architecture*”
 - ▶ Adaptado do código em:
<https://github.com/asakichy/PofEAA>
- ◆ Além do *Data Mapper*, o código exemplifica a implementação dos padrões ***Identity Map*** e ***Finder***

“Lição de Casa”

- ◆ Ler (no site ou no livro do Martin Fowler) sobre os padrões:
 - ▶ Unit of Work
 - ▶ Identity Map
 - ▶ Lazy Load

Referências Bibliográficas

- ◆ Livro: “Patterns of Enterprise Application Architecture”, de Martin Fowler
 - ◆ Capítulo 3 - “Data Source Architectural Patterns” está disponível online:
<http://www.informit.com/articles/article.aspx?p=1398618>
 - ◆ Resumo dos padrões:
<http://martinfowler.com/eaCatalog/>
- ◆ “Core J2EE Patterns - Data Access Object”
<http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>
- ◆ Apostila da Caelum - “Banco de Dados e JDBC”
<http://www.caelum.com.br/apostila-java-web/bancos-de-dados-e-jdbc/>
- ◆ Slides do prof. João Eduardo Ferreira sobre persistência:
<http://www.ime.usp.br/~jef/persistencia.pdf>