

Mapeamento Objeto-Relacional

Data Access Object e Outras
Soluções

Gleydson Lima
gleydson@j2eebrasil.com.br



Metas

- ◆ Escrever uma arquitetura de mapeamento objeto relacional
- ◆ Evitar redundância de codificação dos comandos SQL
 - Diminuir dependência da mudança do banco
- ◆ Estudar o padrão DAO (Data Access Object)
- ◆ Analisar custo x benefício
- ◆ Implementação

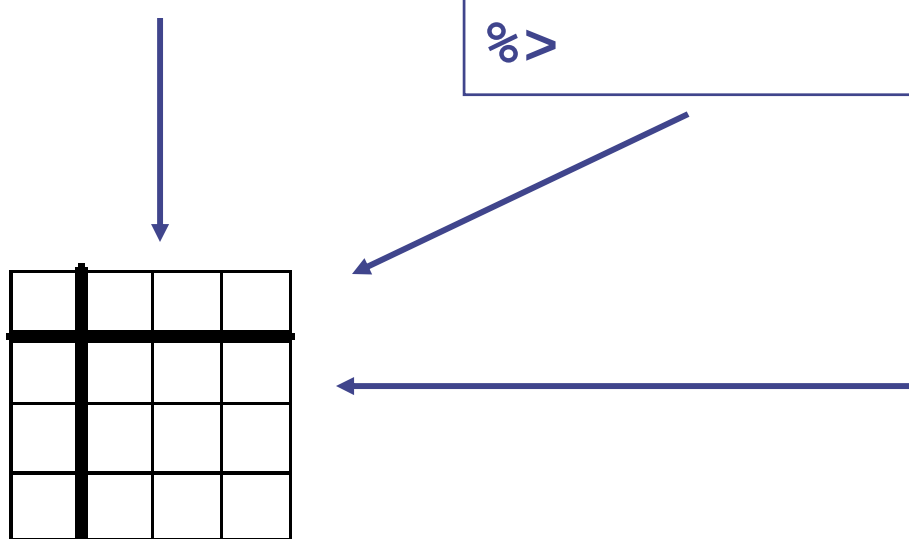


Solução tradicional

```
<%  
SELECT NOME FROM  
USUARIO WHERE ID =  
20  
%>
```

```
<%  
SELECT NOME,  
ENDereco, TELEFONE  
FROM USUARIO WHERE  
toUpper(LOGIN) =  
'GLEYDSON'  
%>
```

```
<%  
SELECT TIPO_USUARIO  
FROM USUARIO WHERE  
LOGIN = 12  
%>
```



**Tabela
Usuário**



Solução tradicional

◆ Vantagens

- Performance: Apenas os parâmetros necessários são recuperados

◆ Desvantagens

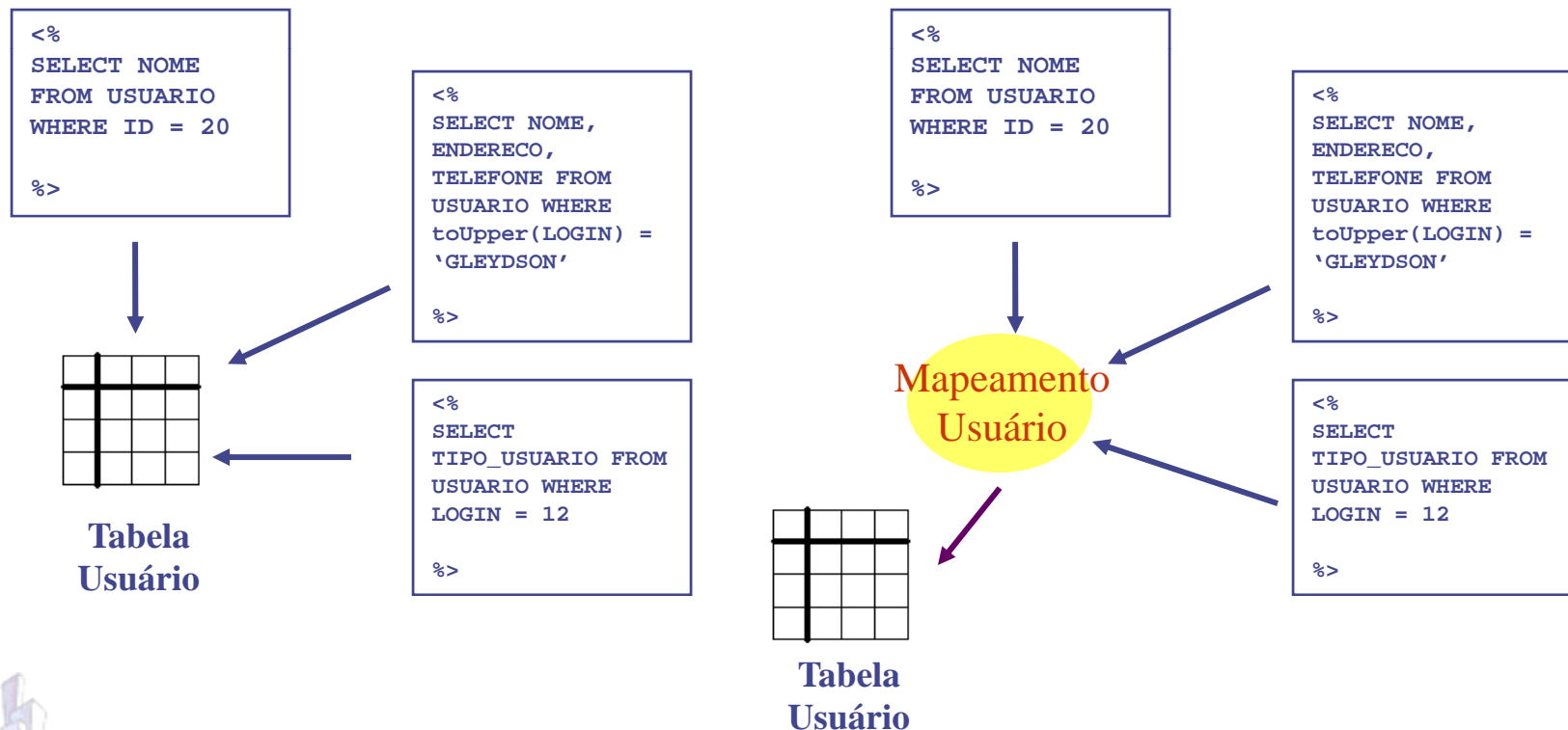
- Muitos pontos de dependências: Qualquer modificação no banco de dados implica em diversas modificações no código
 - ◆ Em grandes sistemas isso pode ser explosivo!!
- Replicação de código
- Mais rápido, APARENTEMENTE



Solução tradicional

◆ Eliminação de Dependências

- A eliminação das dependências normalmente se refere ao uso de fachadas (Facade)



Primeira Solução

◆ Código de Mapeamento na classe de domínio

```
public class Cliente {  
    private int id;  
    private String nome;  
    private long cpf;  
    // sets e gets  
  
    public void insert() {  
        // Código JDBC da Inserção  
    }  
    public void update() { // Atualização }  
    public void delete() { // Remoção }  
    public static Cliente findByPrimaryKey(int id) { }  
    public static Cliente findByCpf(long cpf) { }  
    public static Collection findByNome(String nome)  
    public void setConnection(Connection con);  
}
```



Primeira Solução (Uso)

```
public class ControleInsercao {  
    public void executar() {  
        // Abre conexão com banco de dados  
        Connection con = ....;  
        Cliente c = new Cliente();  
        c.setNome(nome informado);  
        c.setCpf(cpf informado);  
        c.setConnection(con);  
        c.insert();  
    }  
}
```

O código ao lado é usado para incluir um cliente na base dados



Primeira – Métodos de Busca

◆ Primeira Solução (métodos de busca)

```
public class Cliente {  
    public static Collection findByName(String nome) {  
        Statement st = con.createStatement();  
        ResultSet rs = st.executeQuery("SELECT *  
FROM CLIENTE WHERE NOME LIKE ' " + nome + "%'");  
        ArrayList result = new ArrayList();  
  
        while ( rs.next() ) {  
            Cliente c = new Cliente();  
            c.setId(rs.getInt("ID"));  
            c.setNome(rs.getString("NOME"));  
            c.setCpf(rs.getLong("CPF"));  
            result.add(c);  
        }  
        return result;  
    }  
}
```



Primeira Solução

- Vantagens

- ◆ Diminui a dependência com o banco de dados
- ◆ Solução Orientada a Objetos

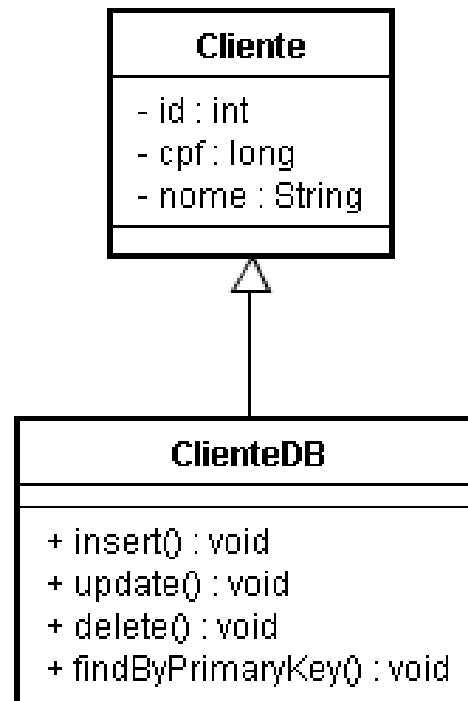
- Desvantagens

- ◆ Vinculação entre Classe de Domínio e Classe de Mapeamento
- ◆ Caso eu queira reutilizar a classe Cliente em outro ambiente (Ex: Celular)?
 - Levarei junto todo o código de mapeamento



Segunda Solução

- Classe de Mapeamento que herda a classe de domínio



Segunda Solução

```
Cliente c = new Cliente();  
c.setCpf(23232323);  
c.setNome("Marcos Valério");
```

Neste ponto não é possível persistir o objeto da classe Cliente, é OBRIGATÓRIO o uso da classe filha de persistência

```
ClienteDB c = new ClienteDB();  
c.setCpf(23232323);  
c.setNome("Marcos Valério");  
c.setConnection(con);  
c.insert();
```

Note que onde é esperado Cliente, qualquer objeto da classe ClienteDB é aceito. Ou seja, um ClienteDB é um Cliente.



Segunda Solução

◆ Vantagens

- Diminui acoplamento entre classe de domínio e classe de mapeamento

◆ Desvantagens

- Não permite que classes de domínios sejam mapeadas se o filho não for instanciado.
- Não permite persistir uma classe de domínio a qualquer momento desejado.

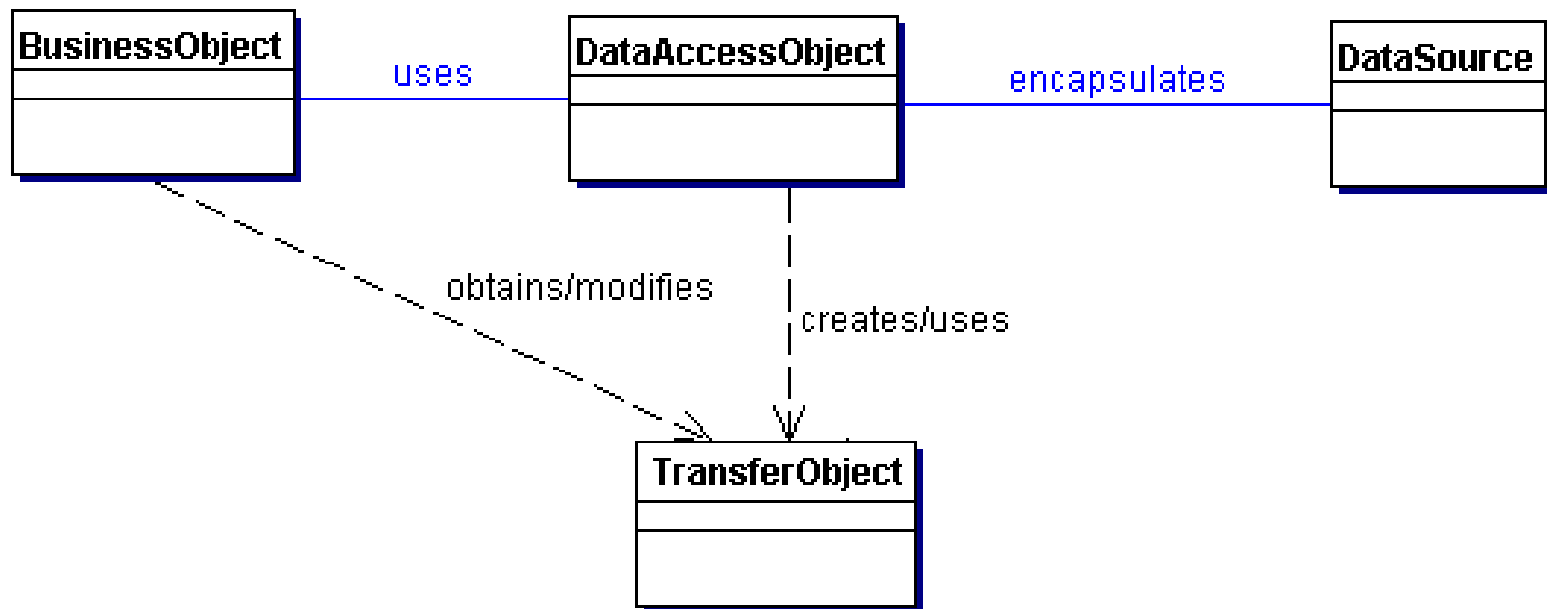


Data Access Object

- ◆ Padrão de Projeto utilizado para mapeamento objeto relacional
- ◆ Tornar o código de persistência mais organizado, reusável e desacoplado da lógica de negócio
- ◆ O objetivo, assim como as soluções anteriores, é tornar o desenvolvimento de código JDBC mais organizado.
- ◆ É considerado um J2EE Pattern.



Data Access Object (DAO)

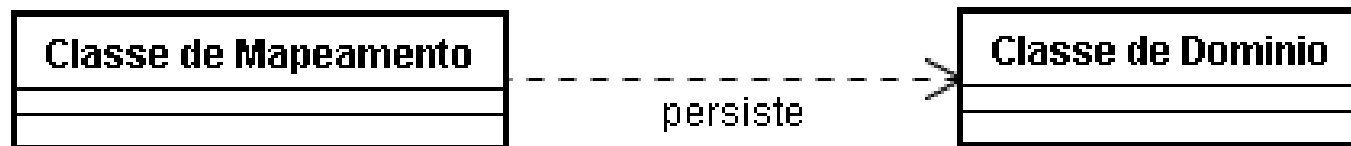


- ◆ **BusinessObject**: Classe que contém a lógica de negócio
- ◆ **DataAccessObject**: Classes de mapeamentos
- ◆ **DataSource**: Representa a fonte dos dados
- ◆ **TransferObject**: Objeto de Domínio (Ex: Cliente)

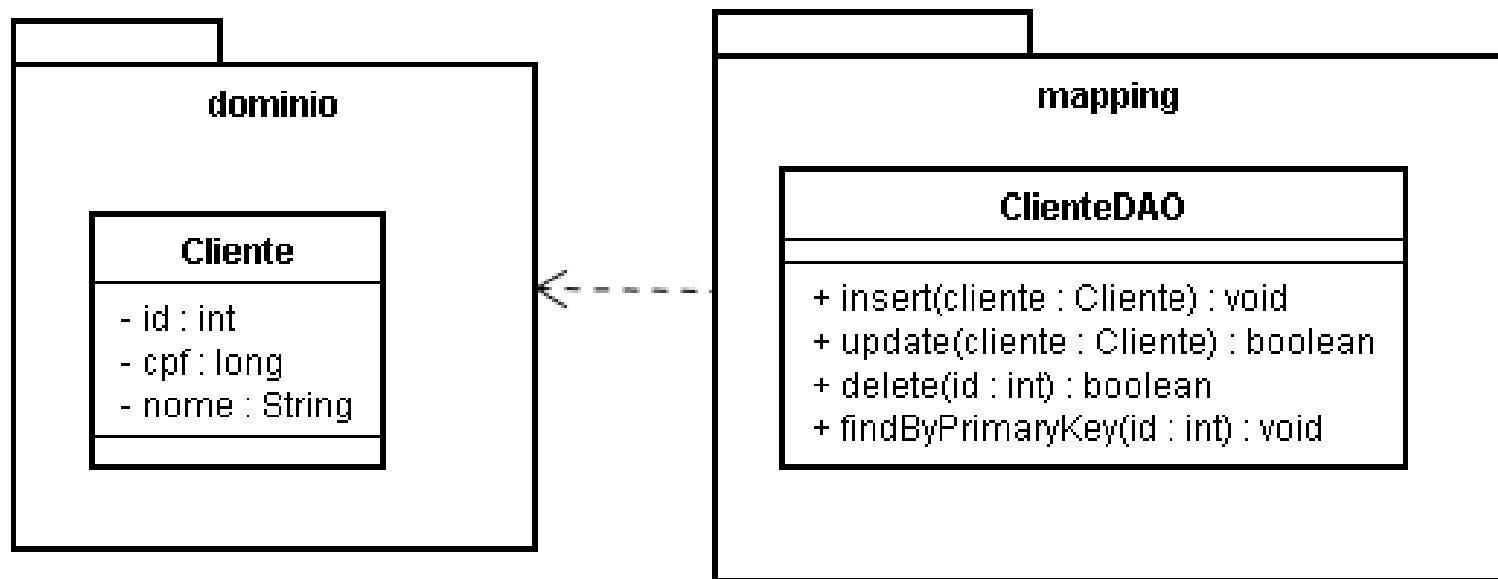


DAO

- ◆ **Separação entre classe de domínio e classe de mapeamento**
 - Solução de baixo acoplamento



DAO



DAO

◆ Vantagens

- Não há dependência da classe de domínio com a classe de mapeamento, o contrário que é verdadeiro
- Redução da dependência do banco de dados para um único ponto

◆ Desvantagens

- Ainda continuamos tendo que escrever o código de mapeamento básico (inserir, atualizar, remover, buscar pela chave primária)
- Não há um mecanismo para possibilitar o mapeamento com mais de um mecanismo de armazenamento



DAO - Exemplo

```
public class Cliente {  
    private int id;  
    private long cpf;  
    private String nome;  
  
    // gets e sets  
}  
  
public class ClienteDAO {  
    private Connection con;  
  
    public void setConnection(Connection con) {  
        this.con = con;  
    }  
  
    public void insert(Cliente c) throws SQLException {  
        PreparedStatement st = con.prepareStatement("INSERT  
INTO CLIENTE (CPF,NOME) values (?,?)");  
        st.setLong(1, c.getCpf());  
        st.setString(2,c.getNome());  
        st.executeUpdate();  
        st.close();  
    }  
}
```



DAO - Exemplo

```
public boolean update(Cliente c) throws SQLException {  
  
    PreparedStatement st = con.prepareStatement("UPDATE CLIENTE  
SET CPF = ?, NOME = ? WHERE ID = ?");  
    st.setLong(1, c.getCpf());  
    st.setString(2, c.getNome());  
    st.setInt(3, c.getId());  
    int count = st.executeUpdate();  
    st.close();  
  
    return count == 1;  
}
```

```
public boolean delete(int id) throws SQLException {  
  
    PreparedStatement st = con.prepareStatement("DELETE FROM  
CLIENTE WHERE ID = ?");  
    st.setInt(1, id);  
    int count = st.executeUpdate();  
    st.close();  
  
    return count == 1;  
}
```



DAO - Exemplo

```
public Cliente findByPrimaryKey(int id) throws SQLException {  
  
    PreparedStatement st = con.prepareStatement("SELECT CPF,NOME  
FROM CLIENTE WHERE ID = ?");  
    st.setId(1, c.getId());  
    Cliente c = null;  
  
    ResultSet rs = st.executeQuery();  
    if ( rs.next() ) {  
        c = new Cliente();  
        c.setId(id);  
        c.setCpf(rs.getLong("CPF"));  
        c.setNome(rs.getNome("NOME"));  
    }  
    return c;  
}
```



DAO - Uso

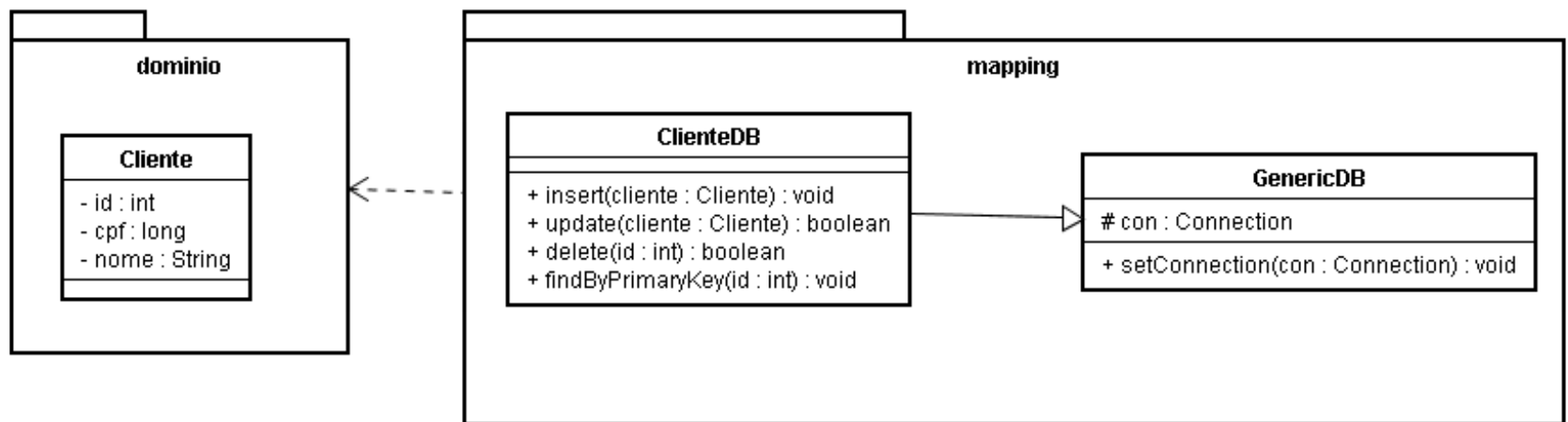
```
Cliente c = new Cliente();  
c.Cpf(1232323423);  
c.setNome("Lula Bobinho");  
  
ClienteDAO cDAO = new ClienteDAO();  
cDAO.setConnection(con);  
cDAO.inserir(c);
```

- ◆ Um objeto da classe ClienteDAO pode persistir qualquer objeto da Classe Cliente
- ◆ Não há acoplamento como na solução anterior
- ◆ Código limpo, fácil e organizado.
- ◆ Normalmente, uma VO (Value Object) por tabela é usado

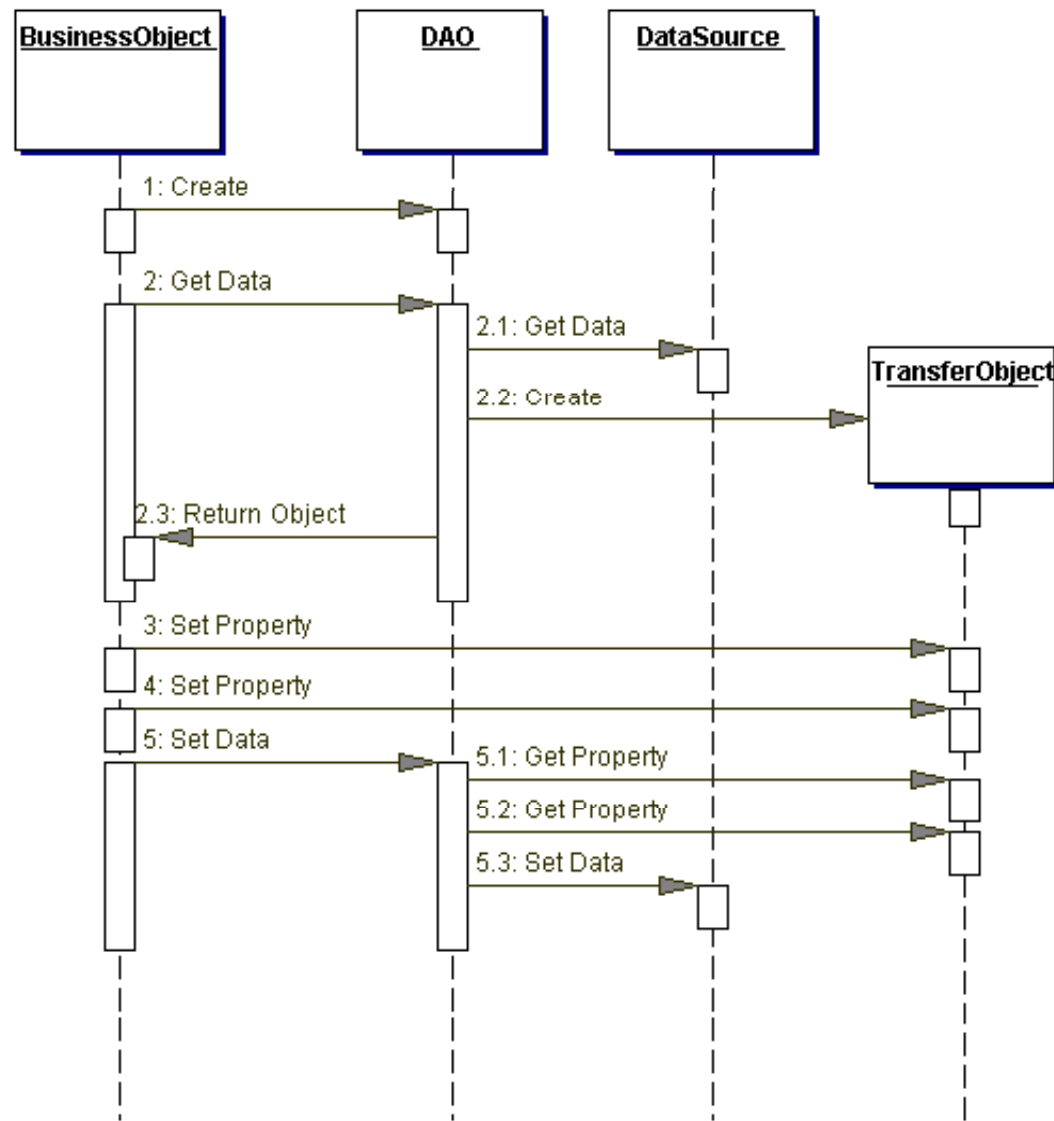


DAO - Melhorias

- ◆ Classes de mapeamento herdam uma classe geral que compartilha o comportamento de todos os mapeamentos



DAO - Seqüência



Custo x Benefício

- ◆ A construção de códigos de mapeamento torna a aplicação bem mais organizada
- ◆ Não representa mais tempo de programação
- ◆ Pode ser aplicado em qualquer linguagem orientada a objetos



Exercício e Dúvidas

◆ Implementar o DAO e mapeamento de Cliente

- Dois métodos de busca:
 - ◆ Cliente por CPF
 - ◆ Cliente por Nome (com Like)

?

