Persistência com JDBC e JPA

Aula 2

Marcos Alberto Lopes da Silva

(malopes21@gmail.com)

Pós-Java 2017-2018!!

Sumário

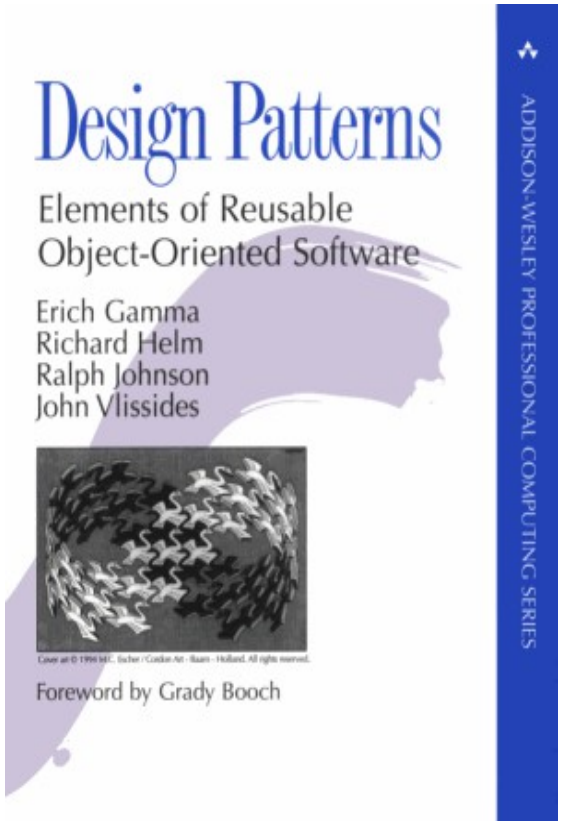
- Persistência de Objetos - MOR
- DAO – Data Access Object
- Pool de conexões;
- Persistência automática com Jpa;

Siglas e acrônimos

- POO = Programação Orientada a Objetos;
- SGBDR = Sistema de gerenciamento de bancos de dados relacionais;
- DAO = Data Acess Object;
- JDBC = Java Database Connectivity;
- JPA = Java Persistence API;
- ORM = object-relational mapping ou mapeamento objeto-relacional;
- JSR = Java Specification Request;

Persistência de objetos

- Popularização do paradigma POO nos últimos 15~20 anos



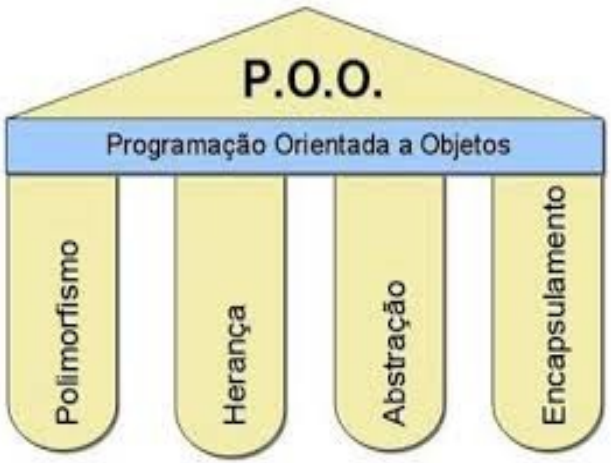
Persistência de objetos

- Porém...

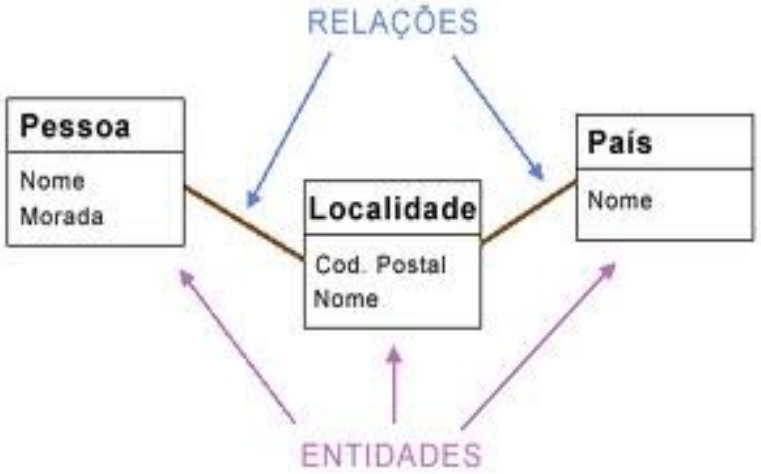
- Compatibilidade entre POO e SGBDR?

Persistência de objetos

- Objetos vs Tabelas;
- Herança vs Joins;
- Encapsulamento vs Projeção

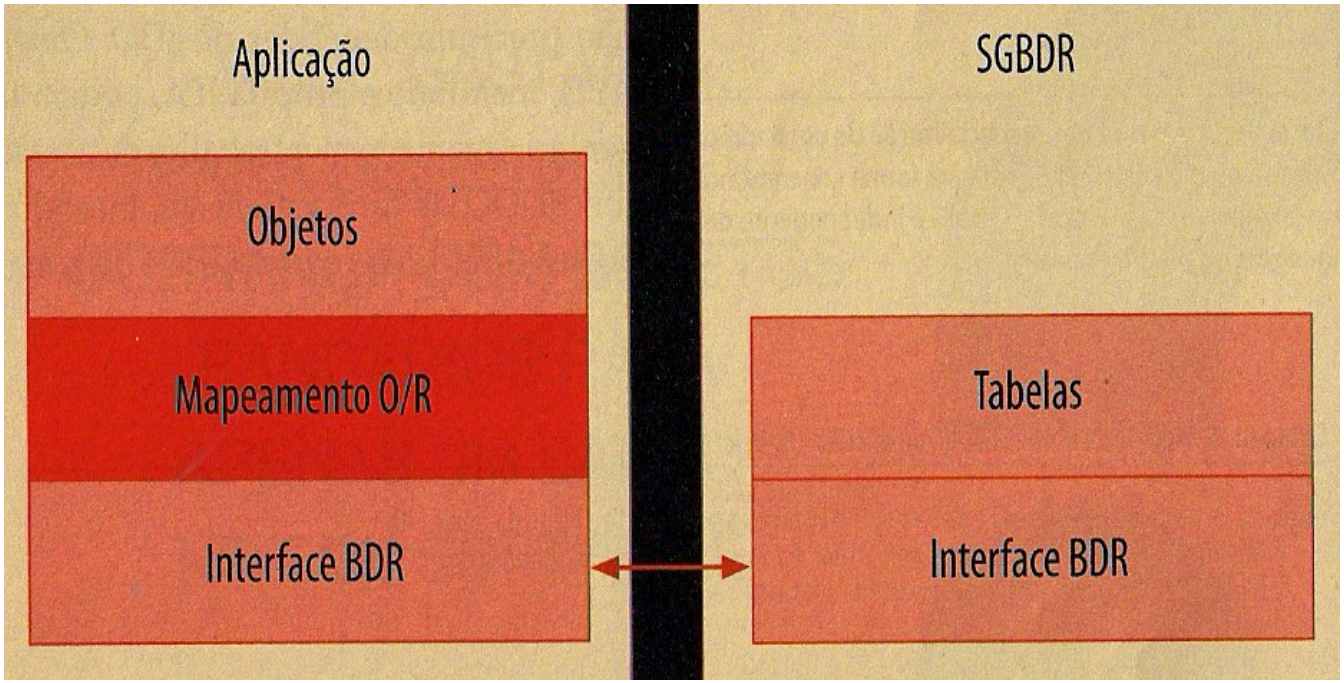


VS

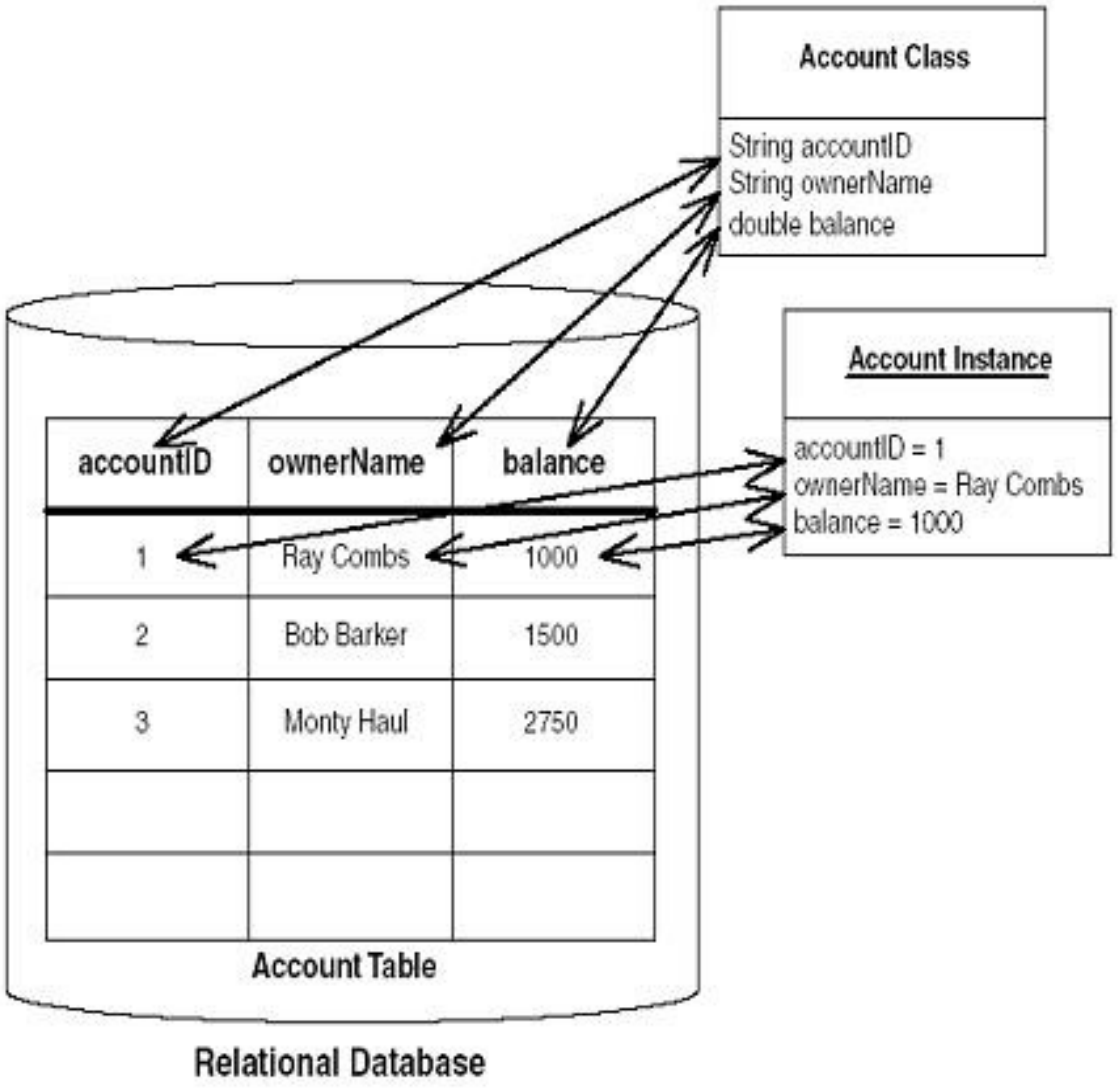


Persistência de objetos

- BDR – interface nativa do SGBDR (Sql, estruturas de dados, protocolos de rede otimizados para executar consultas sql e transferir dados tabulares



Persistência de objetos



Persistência de objetos

- Opções:

  1) JDBC – mapeamento objeto-relacional escrito à mão; padrão DAO/Data Mapper, Active Record, etc.

  2) Hibernate/EclipseLink/ibatis – mapeamento objeto-relacional automático;

  4) JPA sobre Hibernate/EclipseLink/ibatis – mapeamento objeto-relacional automático;

  3) Bancos OO – sem nenhum mapeamento

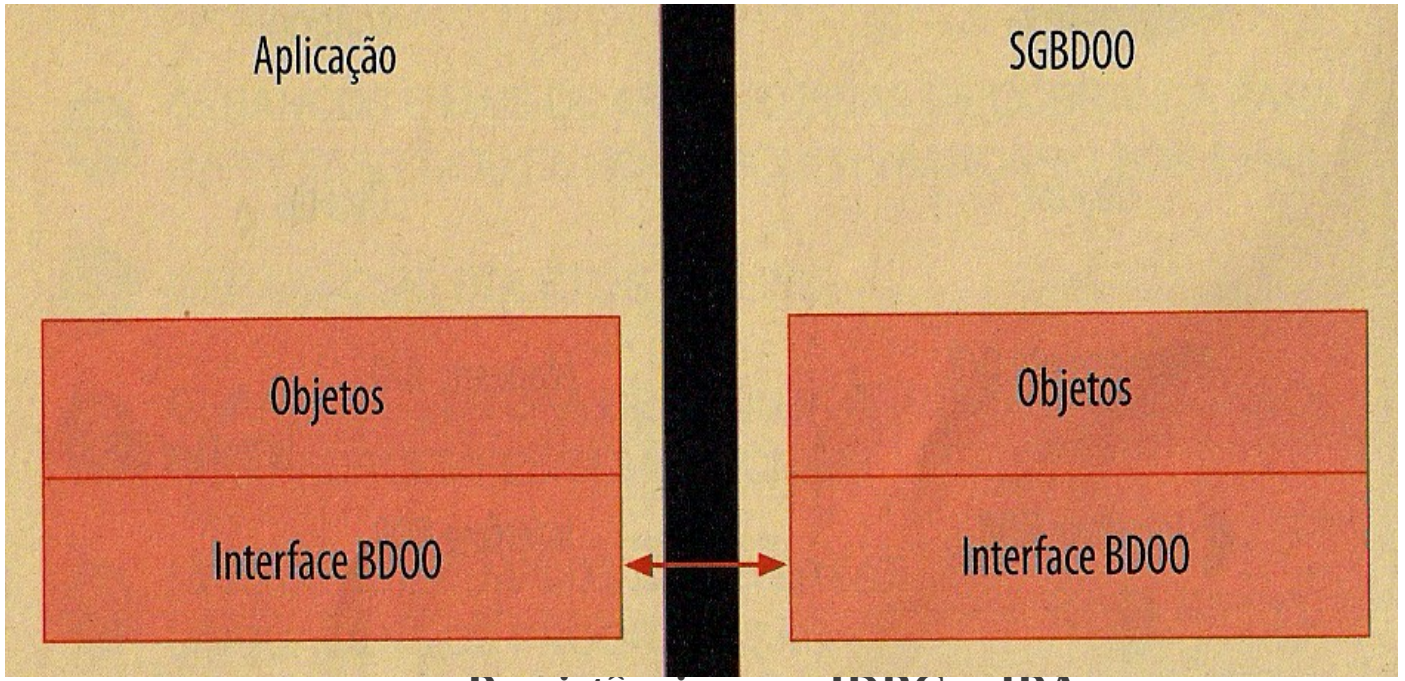
Persistência de objetos

- Questões de projeto:

- Uma grande ”unificação OO” é melhor que misturar paradigmas?
- Todos os Bds são estritamente relacionais?
- Ferramentas ORM acessam dados de forma mais eficiente e produtiva?

Bancos OO

- Armazenam dados em hierarquias de classes;
- Linguagem OO para consultas ;
- Desempenho geralmente satisfatório



Porém...

- Falta de padronização;
- E o legado relacional .... ???

ORMs automáticos

- Fazem a conversão entre objetos e tabelas (JDO, EJB/CMP,Hibernate, etc..);
- Produtividade;
- Padronização

Java  Persistence

Porém...

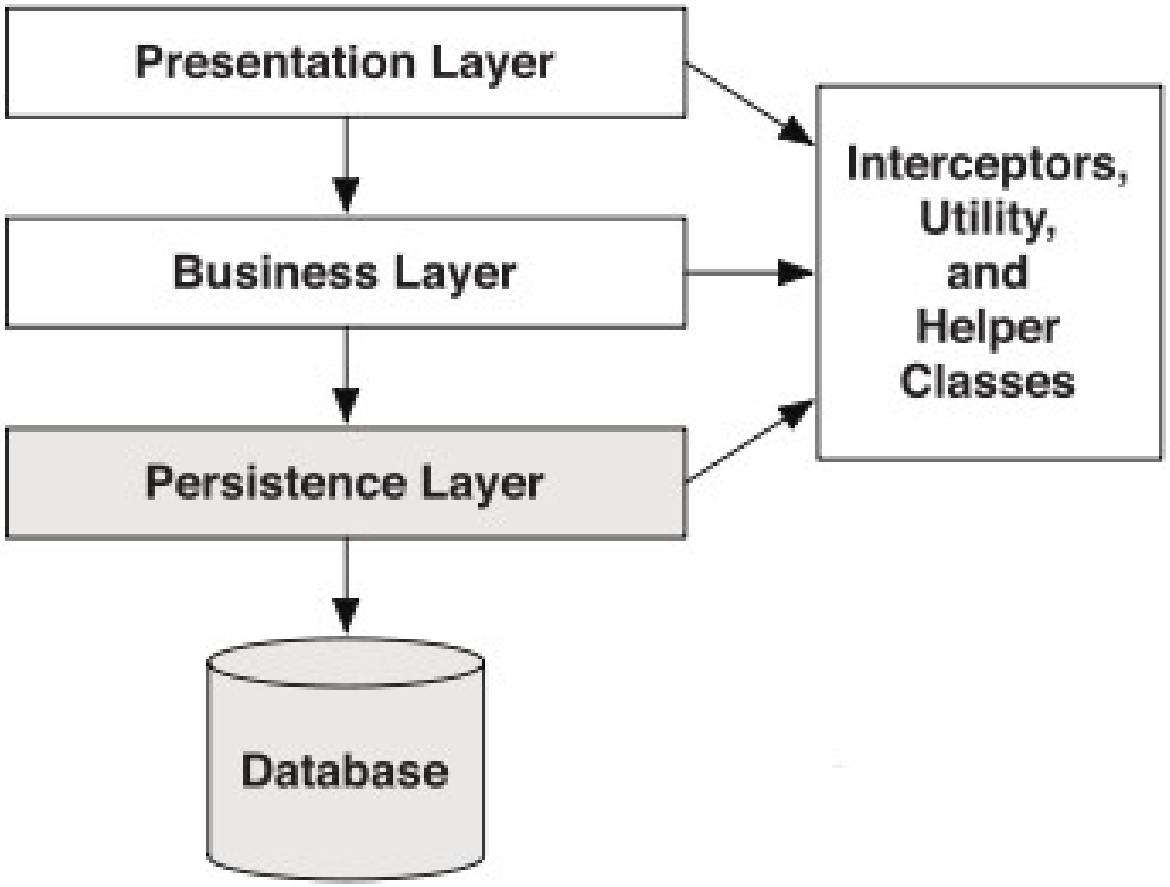
- Necessidade de domínio de 2 paradigmas (POO e relacional);
- Curva de aprendizado para conhecer um sofisticado mecanismo de ORM, cheio de opções e estratégias;
- Eficiência ligeiramente inferior (é discutível)

Design Patterns

- Um padrão de projeto descreve uma solução bem conhecida e testada para uma dada situação recorrente durante o desenvolvimento de sistemas.

- Existem diversos padrões documentados e catalogados. Normalmente esses padrões são classificados (categorizados) pela camada onde se aplica o padrão

Arquitetura em camadas



Design Patterns – camada dados

- Particularmente para acesso a dados tem-se 2 padrões bastante usados no desenvolvimento de sistemas corporativos (catálogo de Martin Fowler):

- - Active Record
  http://www.martinfowler.com/eaaCatalog/activeRecord.html

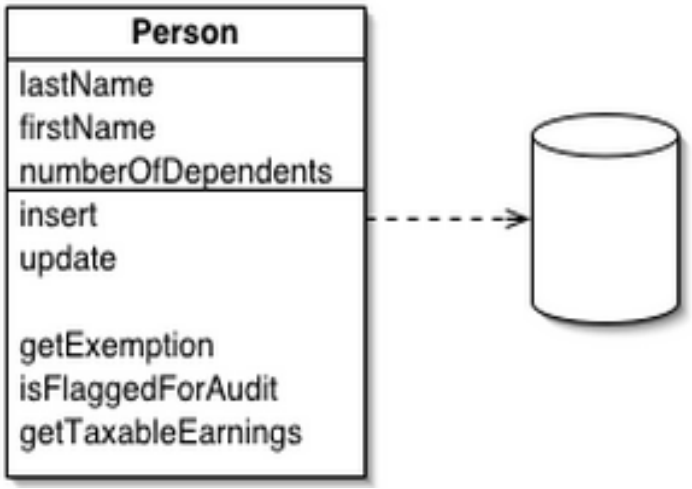
- - Data Mapper (DAO – Data Access Object)
  http://www.martinfowler.com/eaaCatalog/dataMapper.html

Active Record

Active Record

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

For a full description see P of EAA page **160**

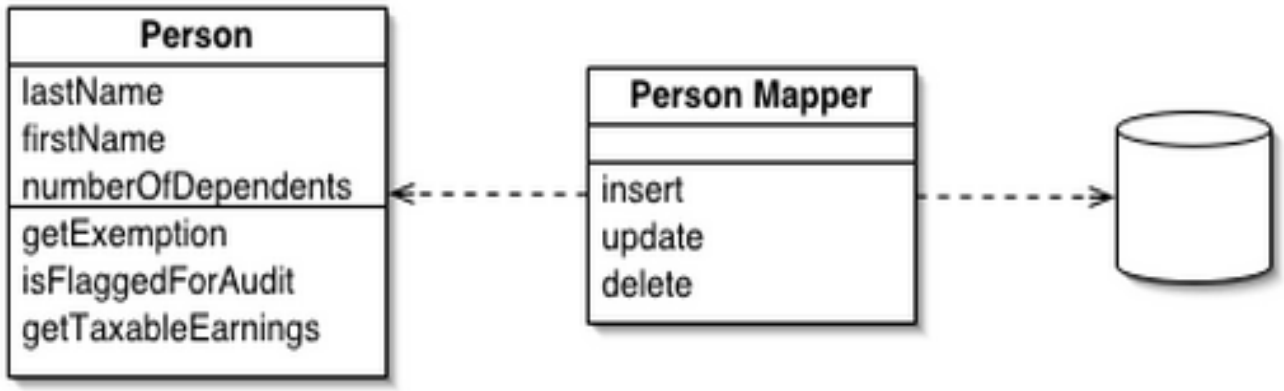


Data Mapper | DAO – Data Access Object

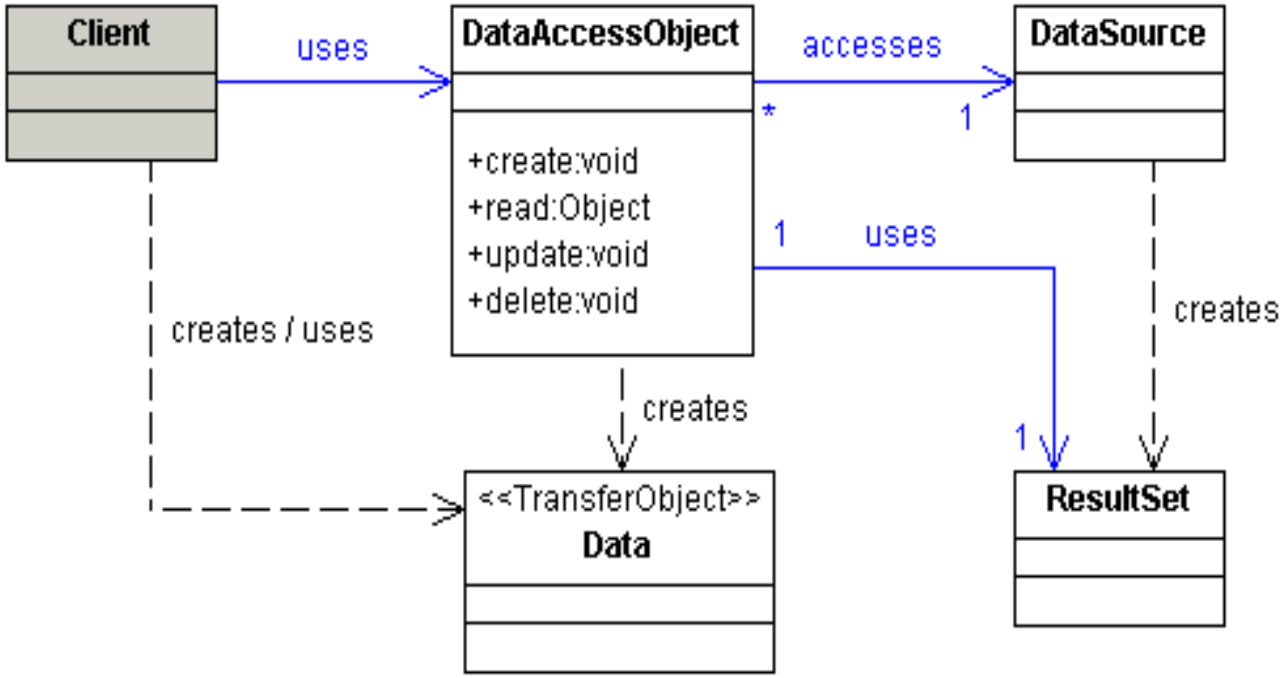
Data Mapper

A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself.

For a full description see P of EAA page **165**

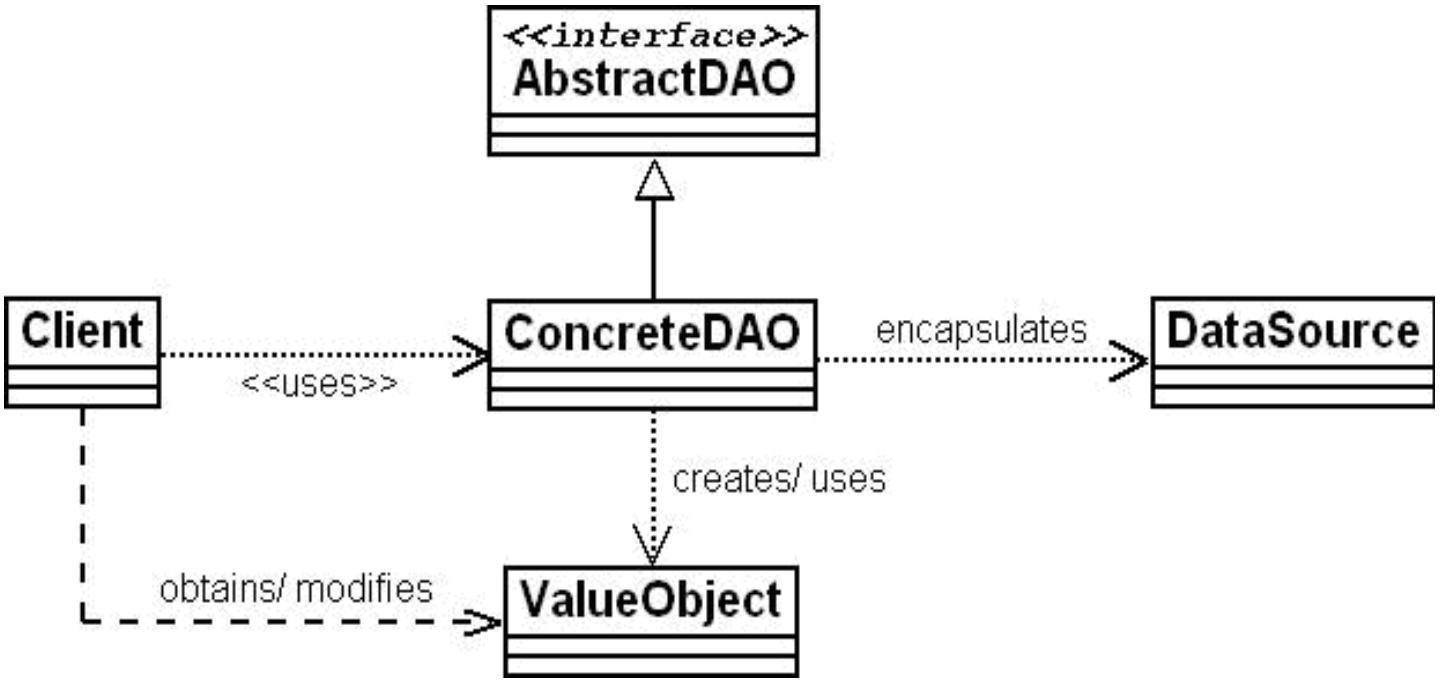


Data Mapper | DAO – Data Access Object



http://www.corej2eepatterns.com/DataAccessObject.htm

Abstract DAO



Código – Tabela x Classe

```sql
create table Cliente (
    codigo int not null,
    nome varchar(100) not null,
    tipo int,
    estado varchar(20),
    primary key (codigo)
)
```

```java
public class Cliente implements Serializable {
    private static final long serialVersionUID = 1L;
    private Integer codigo;
    private String nome;
    private Integer tipo;
    private String estado;

    //gets sets equals hashCode ...
}
```

Código – DAO, AbstractDAO

```
public interface DAO<T, I> {
    T insert(T t) throws SQLException;
    boolean remove(I i) throws SQLException;
    boolean update(T t) throws SQLException;
    T getById(I i) throws SQLException;
    List<T> findAll() throws SQLException;
}

public abstract class AbstractDAO<T, I> implements DAO<T, I>{
    protected Connection conexao;
    public Connection getConexao() {
        return conexao;
    }
}
```

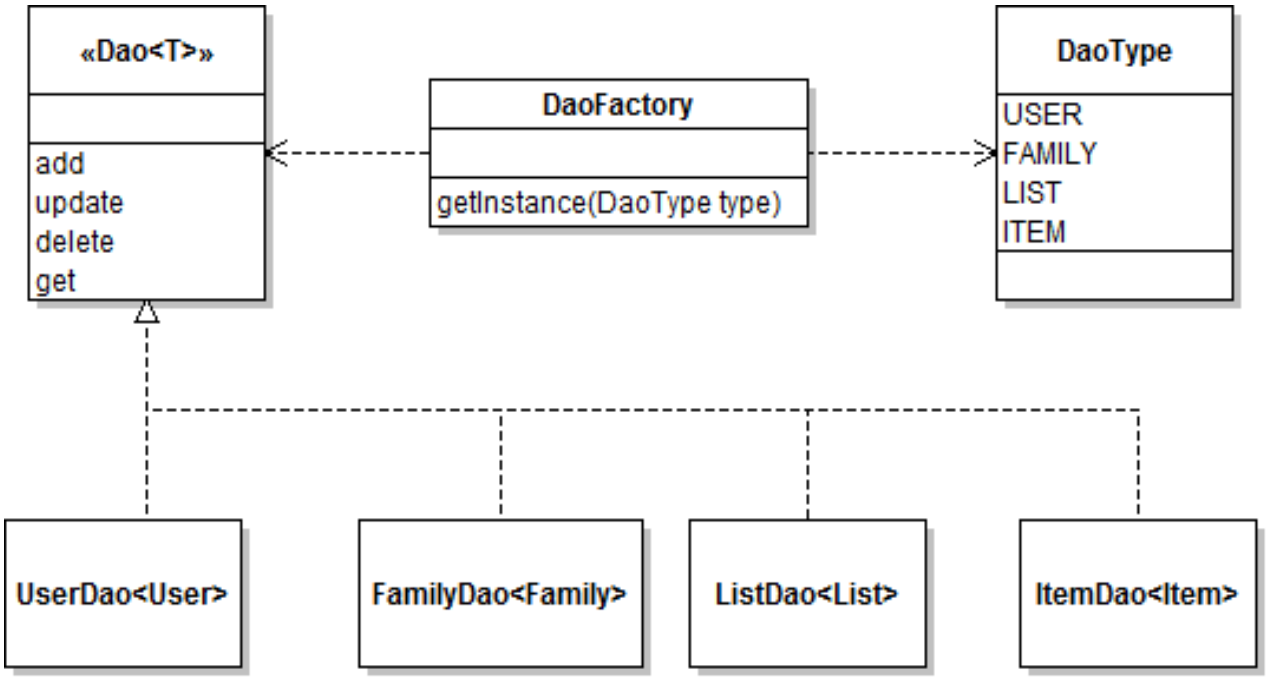
Código - ClienteDAO

```
public class ClienteDAO extends AbstractDAO<Cliente, Integer>{
    public ClienteDAO(Connection conexao){
        this.conexao = conexao;
    }
    @Override
    public Cliente insert(Cliente t) throws SQLException {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    @Override
    public boolean remove(Integer i) throws SQLException {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    @Override
    public boolean update(Cliente t) throws SQLException {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    @Override
    public Cliente getById(Integer i) throws SQLException {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    @Override
    public List<Cliente> findAll() throws SQLException {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

Código – ClienteDAO, update

```
@Override
public boolean update(Cliente cliente) throws SQLException {
    PreparedStatement statement = null;
    String sql = "Update Cliente set nome = ?,tipo = ?,"
                + "estado = ? where codpro = ?";
    try {
        statement = conexao.prepareStatement(sql);
        statement.setString(1, cliente.getNome());
        statement.setInt(2, cliente.getTipo());
        statement.setString(3, cliente.getEstado());
        statement.setInt(4, cliente.getCodigo());
        boolean retorno = statement.execute();
        return retorno;
    } finally {
        try {
            if (statement != null) {
                statement.close();
            }
        } catch (Exception e2) {
            System.err.println("Impossivel fechar comando " + e2);
        }
    }
}
```

DAOFactory


«Dao<T>»
add
update
delete
get
DaoFactory
getInstance(DaoType type)
DaoType
USER
FAMILY
LIST
ITEM
UserDao<User>
FamilyDao<Family>
ListDao<List>
ItemDao<Item>

Código - DAOFactory

```
public class DAOFactory {

    public static DAO<Cliente, Integer> getClienteDAO(Connection conexao)
        throws SQLException {

         return new ClienteDAO(conexao);

    }


    public static DAO<Produto, Integer> getProdutoDAO(Connection conexao)
        throws SQLException {

         return new ProdutoDAO(conexao);

    }

}
```

Código - ConnectionFactory

```
public class ConnectionFactory {

    private static final String driver = "org.apache.derby.jdbc.ClientDriver";
    private static final String url = "jdbc:derby://localhost:1527/posjava-aula02";
    private static final String user = "app";
    private static final String password = "app";

    private static final Logger
       logger = Logger.getLogger(ConnectionFactory.class.getName());

    public static Connection getConnection() {
        try {
            Class.forName(driver);
            Connection con = DriverManager.getConnection(url, user, password);
            con.setAutoCommit(false);
            return con;
        } catch(ClassNotFoundException | SQLException ex) {
            throw new RuntimeException("Erro: banco "+url+". "+ex);
        }
    }

    public static void closeConnection(Connection con, Statement stmt, ResultSet rs){
        try { //...
        } catch(Exception ex) {
            logger.log(Level.SEVERE, "Erro: banco "+url+". {0}", ex);
        }
    }
}
```

Código – ClienteDAOTest - insert

```
@Test
public void insertCliente() {
    Connection conexao = null;
    Cliente clienteSalvo = null;
    try {
        conexao = ConnectionFactory.getConnection();
        DAO<Cliente, Integer> clienteDAO =
                  DAOFactory.getClienteDAO(conexao);
        Cliente clienteNovo =
                  new Cliente(10, "Fulano de Tal", 2, "MG");
        clienteSalvo = clienteDAO.insert(clienteNovo);
        conexao.commit();
    } catch (Exception e) {
        System.err.println(e);
        try {
            if(conexao != null) conexao.rollback();
        }catch(Exception ex) {}
        Assert.fail("não consegui inserir Cliente");
    } finally {
        ConnectionFactory.closeConnection(conexao, null, null);
    }
    assertTrue(clienteSalvo != null);
}
```

Exercício 1

a) Construir um projeto Java standalone e demonstrar funcionando os códigos acima;

b) Implementar os demais métodos de ClienteDAO e implementar os demais métodos de ClienteDAOTest.

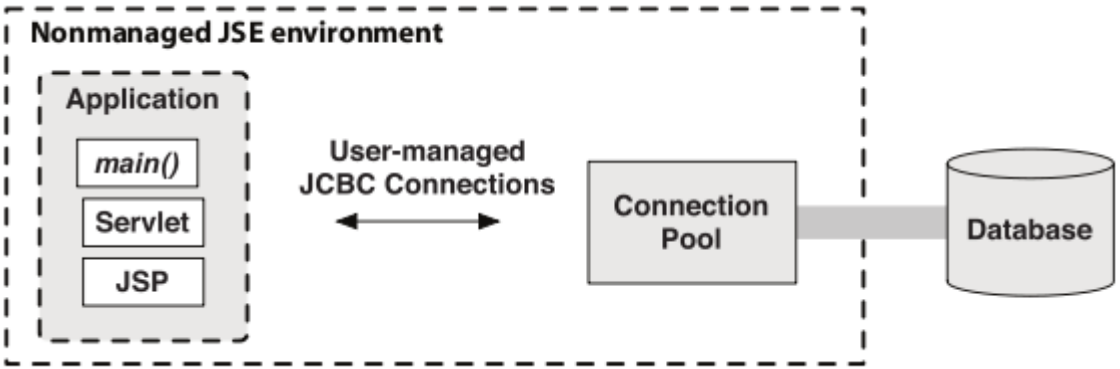
c) Implentar as classes ProdutoDAO e ProdutoDAOTest completas. Produto(codigo:int, descricao:varchar, preco:decimal)

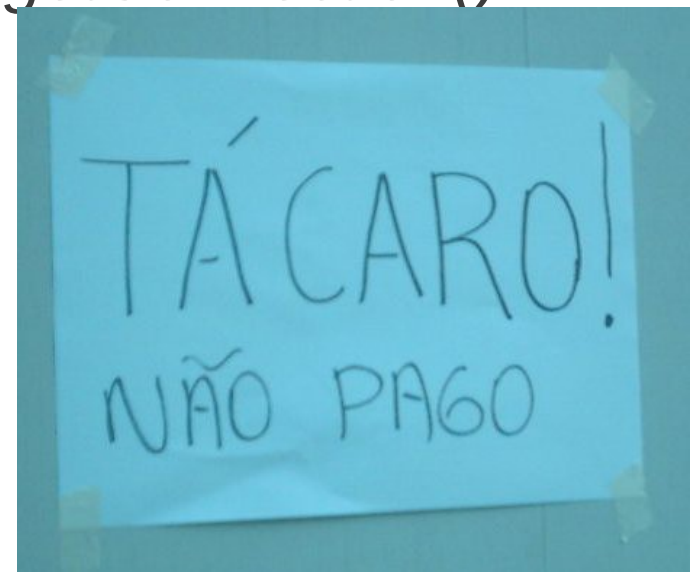
Turbinando a persistência

- Recursos que são utilizados em produção

Pools de conexões

- Pooling tenta reusar os mesmos objetos em vários lugares onde forem necessários;
- Evita criações desnecessárias de objetos;
- O uso é justificado em casos onde a construção de objetos é cara! *DriverManager.getConnection()*





Pools JDBC

- C3p0 http://www.mchange.com/projects/c3p0/index.html ;
- Jakarta DBCP http://commons.apache.org/dbcp/ ;
- Drivers JDBC com seus próprios pools

  No caso do driver Mysql,
  MysqlConnectionPoolDataSource

- Classe *Pool**ConnectionFactory.java***

Código – PoolConnectionFactory - c3p0

```
public class PoolConnectionFactory {
    private static final String driver = "org.apache.derby.jdbc.ClientDriver";
    private static final String url = "jdbc:derby://localhost:1527/posjava-aula02";
    private static final String user = "app";
    private static final String password = "app";

    private static final Logger
         logger = Logger.getLogger(PoolConnectionFactory.class.getName());
    private static ComboPooledDataSource cpds = null;

    public static Connection getConnection() {
        try {
            if (cpds == null) {
                cpds = new ComboPooledDataSource();
                cpds.setDriverClass(driver); //loads the jdbc driver
                cpds.setJdbcUrl(url);
                cpds.setUser(user);
                cpds.setPassword(password);
            }
            Connection con = cpds.getConnection();
            con.setAutoCommit(false);
            return con;
        } catch (Exception ex) {
            throw new RuntimeException("Erro: banco " + url + ". " + ex);
        }
    }
}
```

Exercício 2

- Implementar o pool de conexões do slide anterior e testar para confirmar o funcionamento
- Verificar os métodos de configuração da classe ComboPooledDataSource
- Tente montar uma classe de teste para verificar a melhoria de desempenho para obter 100 conexões usando o pool e sem usar o pool. Anote o resultado!

Exercício 3 - Para próxima aula!

- Criar um banco de dados com **no mínimo** 4 tabelas e que tenham relacionamentos entre si; usar algum domínio específico, por exemplo (acadêmico, agronegócio, locadora, financeiro, bancario, etc)
- Criar classes de DAO para realizar operações com estas tabelas;
- Criar um caso de teste para cada classe DAO para realizar testes;
- Opcional: criar pelo menos um form. Swing para acessar as operações CRUD de uma entidade.

Ferramentas ORM

- Persistência automatizada e transparente dos objetos de uma aplicação qualquer para o SGBDR;
- API CRUD;
- Linguagem de consultas;
- Especificação de metadados;
- Funções de otimização

Ferramentas ORM

- Produtividade (tarefas rotineiras);
- Manutenibilidade (LOC, refactoring, lógica ao invés de encanamento);
- Performance (explora otimizações da api jdbc);
- Independência de fornecedor (portabilidade de SGBDR)

Disparidade de paradigma

- Dados tabulares vs Objetos;
- Problemas estruturais (granulosidade, subtipos, identidade, associações);
- Problemas dinâmicos (navegação de dados, custo da disparidade)

Exemplo disparidade

```
public class User {
    private String username;
    private String name;
    private String address;
    private Set billingDetails;

    // Accessor methods (getter/setter), business methods, etc.
    ...
}
public class BillingDetails {
    private String accountNumber;
    private String accountName;
    private String accountType;
    private User user;

    // Accessor methods (getter/setter), business methods, etc.
    ...
}
```

Exemplo disparidade

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    NAME varchar(50) not null,
    ADDRESS varchar(100)
)
create table BILLING_DETAILS (
    ACCOUNT_NUMBER varchar(10) not null primary key,
    ACCOUNT_NAME varchar(50) not null,
    ACCOUNT_TYPE varchar(2) not null,
    USERNAME varchar(15) foreign key references user
)
```

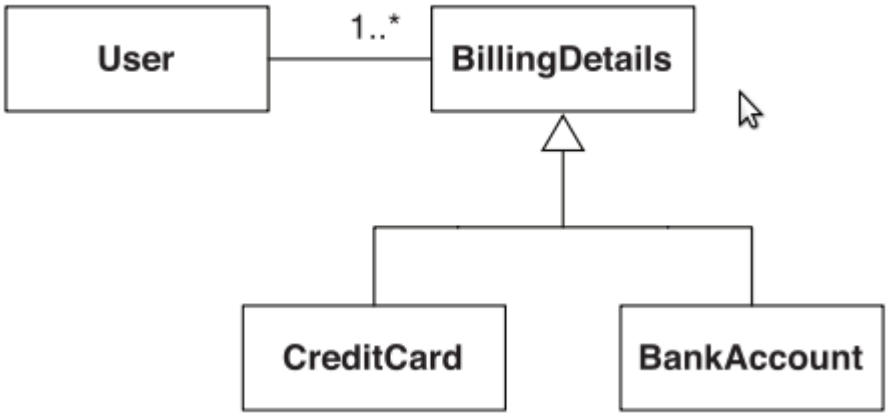
Granulosidade

- Crio uma tabela para Address no banco?

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    NAME varchar(50) not null,
    ADDRESS_STREET varchar(50),
    ADDRESS_CITY varchar(15),


     ADDRESS_ZIPCODE varchar(5),
     ADDRESS_COUNTRY varchar(15)
 )
```

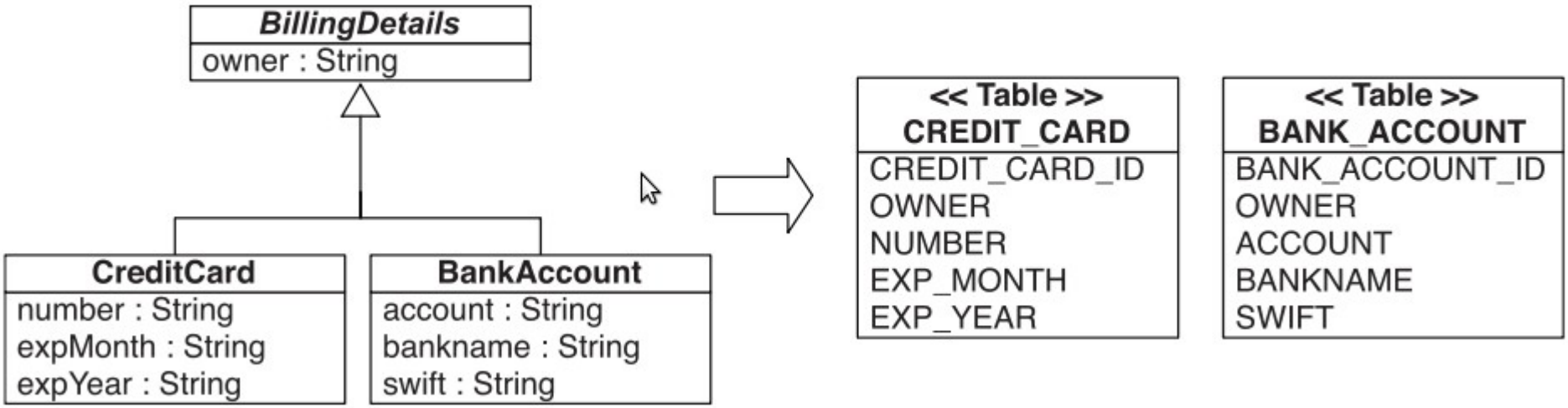
Subtipos



- Tabelas com herança de tipo?
- E o polimorfismo? User não pode referenciar, em tempo de execução, uma instância de qualquer subtipo de BillingDatains?

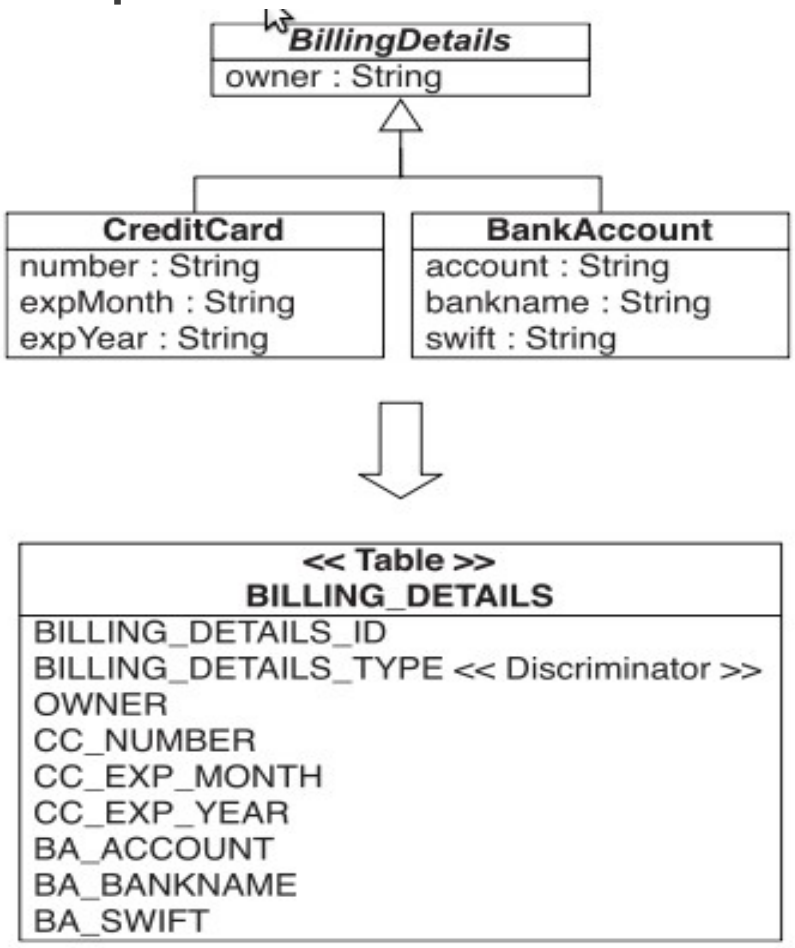
Subtipos

- Tabela por classe concreta



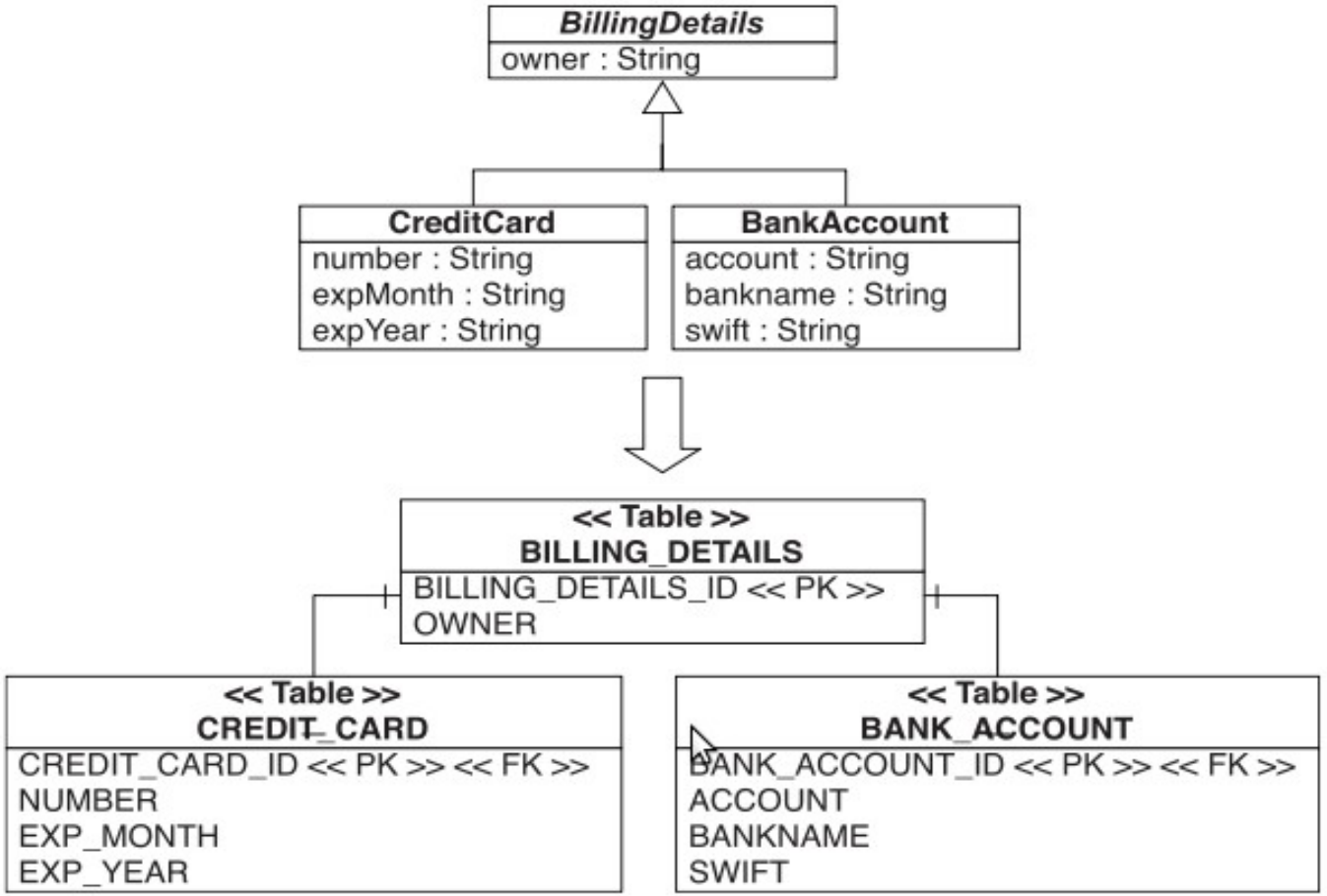
Subtipos

- Tabela por hierarquia de classe



Subtipos

- Tabela por classe



Identidade

- Como checar se dois objetos são idênticos?
- Equals() e == vs chave primária;
- Chaves artificiais (auto incremento)

```
create table USERS (
    USER_ID bigint not null primary key,
    USERNAME varchar(15) not null unique,
    NAME varchar(50) not null,
    ...
)
create table BILLING_DETAILS (
    BILLING_DETAILS_ID bigint not null primary key,
    ACCOUNT_NUMBER VARCHAR(10) not null unique,
    ACCOUNT_NAME VARCHAR(50) not null,
    ACCOUNT_TYPE VARCHAR(2) not null,
    USER_ID bigint foreign key references USER
)
```

Associação

- Referências a objetos vs chaves estrangeiras;
- Chaves estrangeiras não são bidirecionais;
- Navegação não faz sentido em dados relacionais;
- Multiplicidade muitos para muitos não existem em modelos relacionais

```
public class User {
    private Set billingDetails;
    ...
}
public class BillingDetails {
    private User user;
    ...
}
```

```
create table USER_BILLING_DETAILS (
    USER_ID bigint foreign key references USERS,
    BILLING_DETAILS_ID bigint foreign key references BILLING_DETAILS,
    PRIMARY KEY (USER_ID, BILLING_DETAILS_ID)
)
```

Navegação de dados

- Andar em redes de objetos (produto.getCategoria().getCodigo());
- Número de tabelas da junção = profundidade da rede de objetos;

```
select * from USERS u where u.USER_ID = 123
```

```
select *
    from USERS u
    left outer join BILLING_DETAILS bd on bd.USER_ID = u.USER_ID
    where u.USER_ID = 123
```

- Problema das **n + 1 seleções**

Custo da disparidade

- 30% do código escrito em Java é para tratar JDBC e criar a ponte manualmente da disparidade O/R;
- DER ou Diagrama de classes?
- Não existe uma transformação elegante esperando para ser descoberta;

Hibernate

- Framework ORM mais popular do mundo;
- Automatiza tarefas de código repetitivas;
- Gerencia dados persistentes em Java;

Eclipse Link

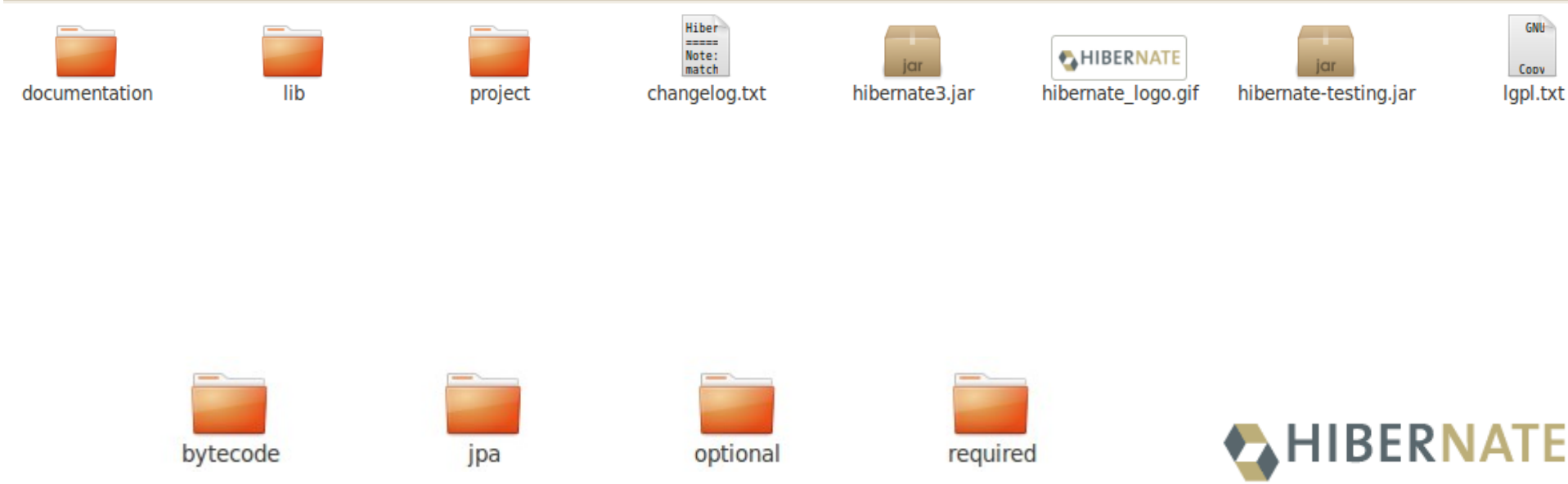
- Framework ORM da fundação eclipse;
- Implementação com foco para JPA, XML e WS;
- Gerencia dados persistentes em Java;

JPA

- Especificação JSR 220,que surgiu com o Java EE 5;
- Motor JPA independente (Hibernate, EclipseLink, iBatis, etc..);
- Permite a execução fora de um contêiner Java EE
- javax.persistence.* => dentro da especificação;
- org.hibernate.* => hibernate nativo

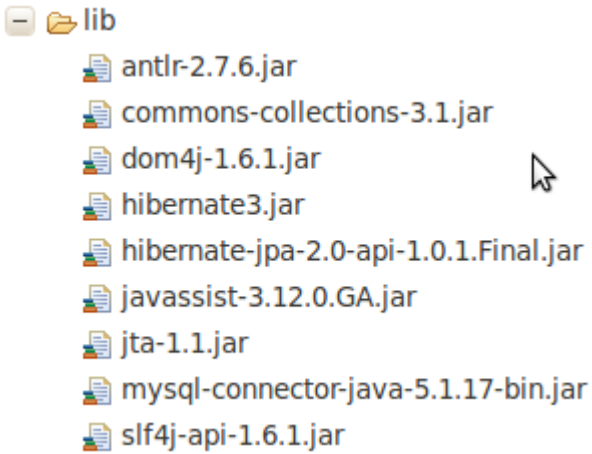
Startup com Hibernate & JPA

- http://www.hibernate.org/



Startup com Hibernate & JPA

- Colocar os jars no classpath do projeto



Mapeamento de entidades

- @Entity – anotação que declara a classe como algo que será persistido no banco;
- @Table – faz o mapeamento da tabela. O owner do banco pode ser mapeado com schema

```
@Entity
@Table(name="tbFornecedor")
public class Fornecedor implements Serializable {
```

Anotações

- Existem desde o Java 5;
- Substituem excessos de configurações XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name='br.edu.unitri.model.Produto' table='produtos'>
        <id name='codigo' type='integer' column='produtoid'>
            <generator class='native' />
        </id>
        <property column='produtoNome' type='string' name='nome' />
        <property column='preco' type='double' name='preco' />
        <property column='unidadesEmEstoque' type='integer'
            name='estoque' />
        <property column='imagem' type='string' name='imagem' />
        <many-to-one column='categoriaid'
            foreign-key='fk_produto_categoria_id'
            class='br.edu.unitri.model.Categoria' not-null='true'
            cascade='save-update' name='categoria' />
    </class>
</hibernate-mapping>
```

```java
@Entity
@Table(name="produtos")
public class Produto implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name="produtoID",nullable=false)
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer codigo;

    @Column(name="ProdutoNome",nullable=false)
    private String nome;

    @JoinColumn(name="CategoriaID",referencedColumnName="categoriaID")
    @ManyToOne
    private Categoria categoria;

    @Column(name="preco")
    private Double preco;

    @Column(name="UnidadesEmEstoque")
    private Integer estoque;

    @Column(name="Imagem")
    private String imagem;
```

Mapeamento de entidades

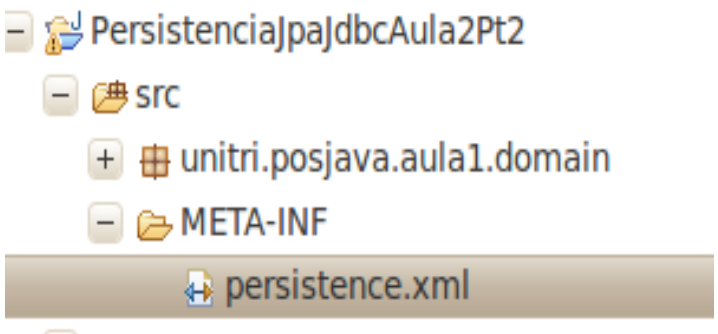
- @Id – qual atributo será a chave primária da classe;
- @GeneratedValue – indica o valor do atributo que compõe uma chave primária deve ser gerado pelo banco no momento da inserção
- @Column – mapeia nomes das colunas e seus metadados

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "CODFRN")
private Integer codigo;

@Column(name= "NOMFRN",length=255,nullable=false)
private String nome;
```

Arquivo de configuração

- persistence.xml configura informações como banco de dados, pools de conexões, cache
- Precisa estar presente dentro do diretório META-INF;
- META-INF precisa estar no classpath do projeto



Arquivo de configuração

- Persistence unit define o nome da unidade de persistência e o tipo de transação que o Hibernate utilizará. RESOURCE_LOCAL é utilizado para Java SE

```
<persistence-unit name="AcaoPU" transaction-type="RESOURCE_LOCAL">
```

- Provider diz qual é o motor JPA que será utilizado

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
```

Arquivo de configuração

- hbm2ddl.auto define que operação será feita na base de dados quando a aplicação iniciar

  none – não faz nada;

  create-drop – dropa e cria a base de dados automaticamente;

  update – invocará alter table a cada novo atributo ou relacionamento

- show_sql – mostra as consultas sql que o hibernate criará

```
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
<property name="hibernate.hbm2ddl.auto" value="none" />
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true" />
```

EntityManager

- EntityManager é uma unidade de acesso aos dados
- EntityManagerFactory é uma fábrica de EntityManager baseada no persistence unit criado (persistence.xml)

Crud

- Crud com Pessoa (id:int, nome:varchar)

- https://sites.google.com/site/malopes21/pos-java

Exercício 4

- Construir um projeto Java/JPA (no NetBeans é mais simples para iniciar) com acesso a um BD relacional (JavaDB, Mysql, etc) com uma tabela Fornecedor (codigo:int, nome:varchar, cnpj:varchar, endereco:varchar)
- Criar uma classe de teste FornecedorJPATest para testar as operações CRUD para Fornecedor.

Seção Dúvidas

- Não existem perguntas idiotas;
- Somos uma equipe, não existe competição

Seção Dúvidas

- Dúvidas, enviem emails!!!!!

  malopes21@gmail.com

Referências bibliográficas

[1] Bauer, Christian e King, Gavin – Java persistence com Hibernate. Rio de Janeiro, Ed. Ciência Moderna, 2007;

[2] ”Programando com Pools” - Java Magazine 57;