



**Universidade de Brasília**

Departamento de Ciência da Computação



# Bancos de Dados

CIC0097



**Prof. Pedro Garcia Freitas**

<https://pedrogarcia.gitlab.io/>

[pedro.garcia@unb.br](mailto:pedro.garcia@unb.br)

Universidade de Brasília  
Instituto de Ciências Exatas  
Departamento de Ciências da Computação



Este conjunto de slides não deve ser utilizado ou republicado sem a expressa permissão do autor.

This set of slides should not be used or republished without the author's express permission.

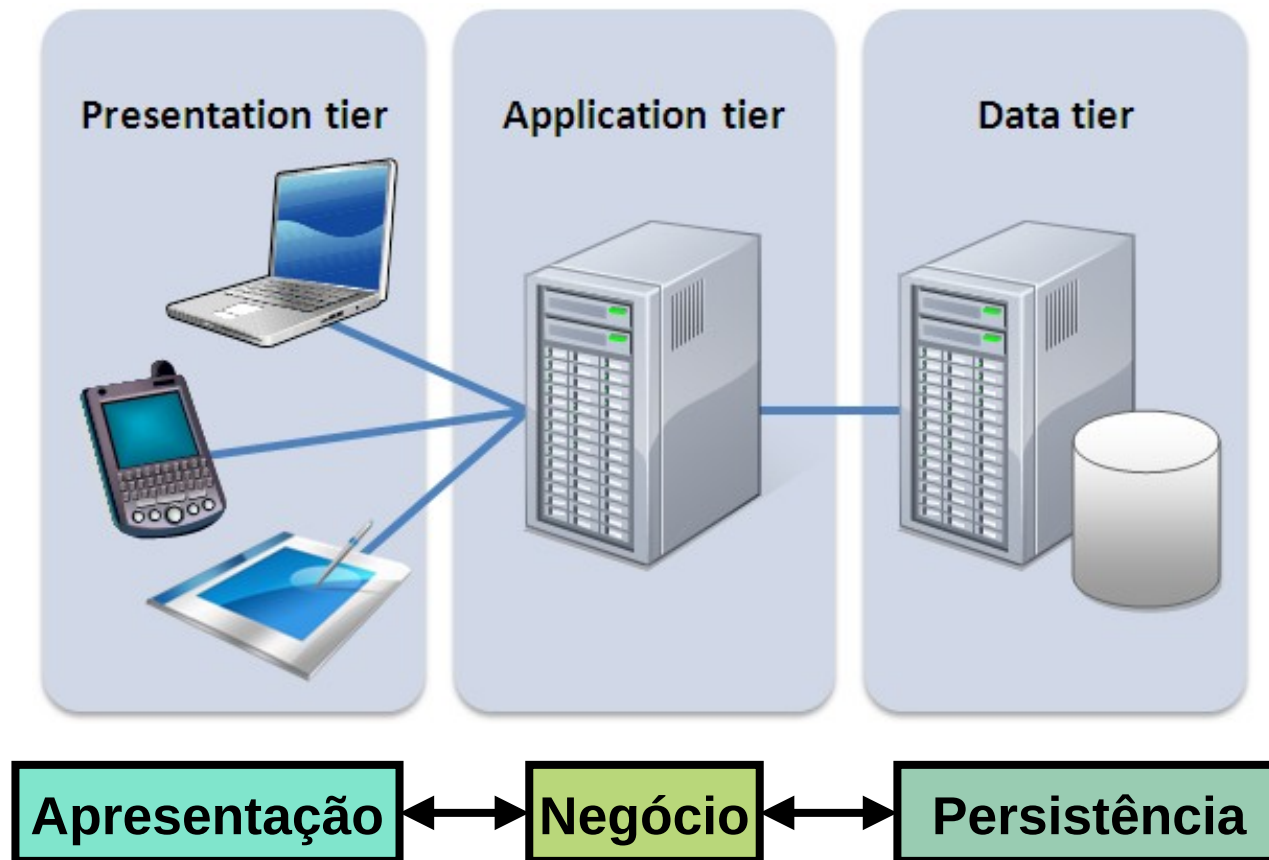


# **Módulo 17**

## **Camadas de Software e Arquitetura para Desenvolvimento**

**CIC0097/2023.1  
T1/T2**

# 1. Arquitetura para Desenvolvimento



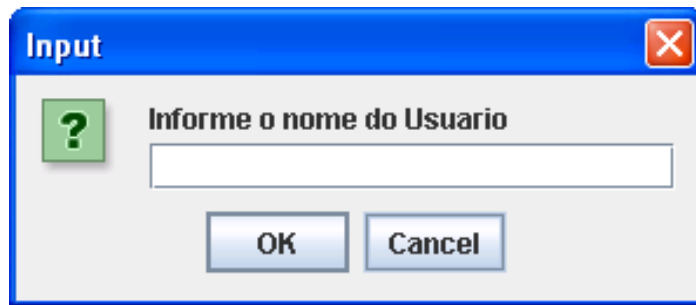
# 1. Arquitetura para Desenvolvimento



**Arquitetura em três camadas  
(*three tier architecture*) é  
o padrão de desenvolvimento  
para aplicativo !!!**



# 1. Arquitetura para Desenvolvimento



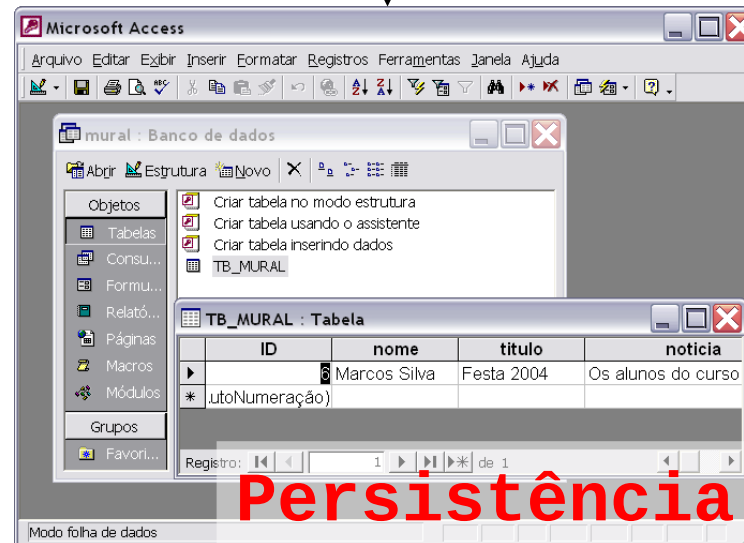
Input

Informe o nome do Usuario

OK Cancel

**Apresentação**

**Negócio/Aplicação**



Microsoft Access

mural : Banco de dados

Objetos

- Tabelas
- Consu...
- Formu...
- Relató...
- Páginas
- Macros
- Módulos
- Grupos
- Favori...

Novo

- Criar tabela no modo estrutura
- Criar tabela usando o assistente
- Criar tabela inserindo dados
- TB\_MURAL

TB\_MURAL : Tabela

ID	nome	titulo	noticia
1	Marcos Silva	Festa 2004	Os alunos do curso
* (autoNumeração)			

Registro: 1 de 1

Modo folha de dados

**Persistência**



# 1. Arquitetura para Desenvolvimento

A arquitetura de três camadas (tiers) é uma arquitetura de aplicativo de software estabelecida que organiza aplicativos em três camadas de computação física e lógica:

- a camada de apresentação ou a interface com o usuário;
- a camada do aplicativo, na qual os dados são processados;
- e a camada de dados, na qual os dados associados ao aplicativo são armazenados e gerenciados.



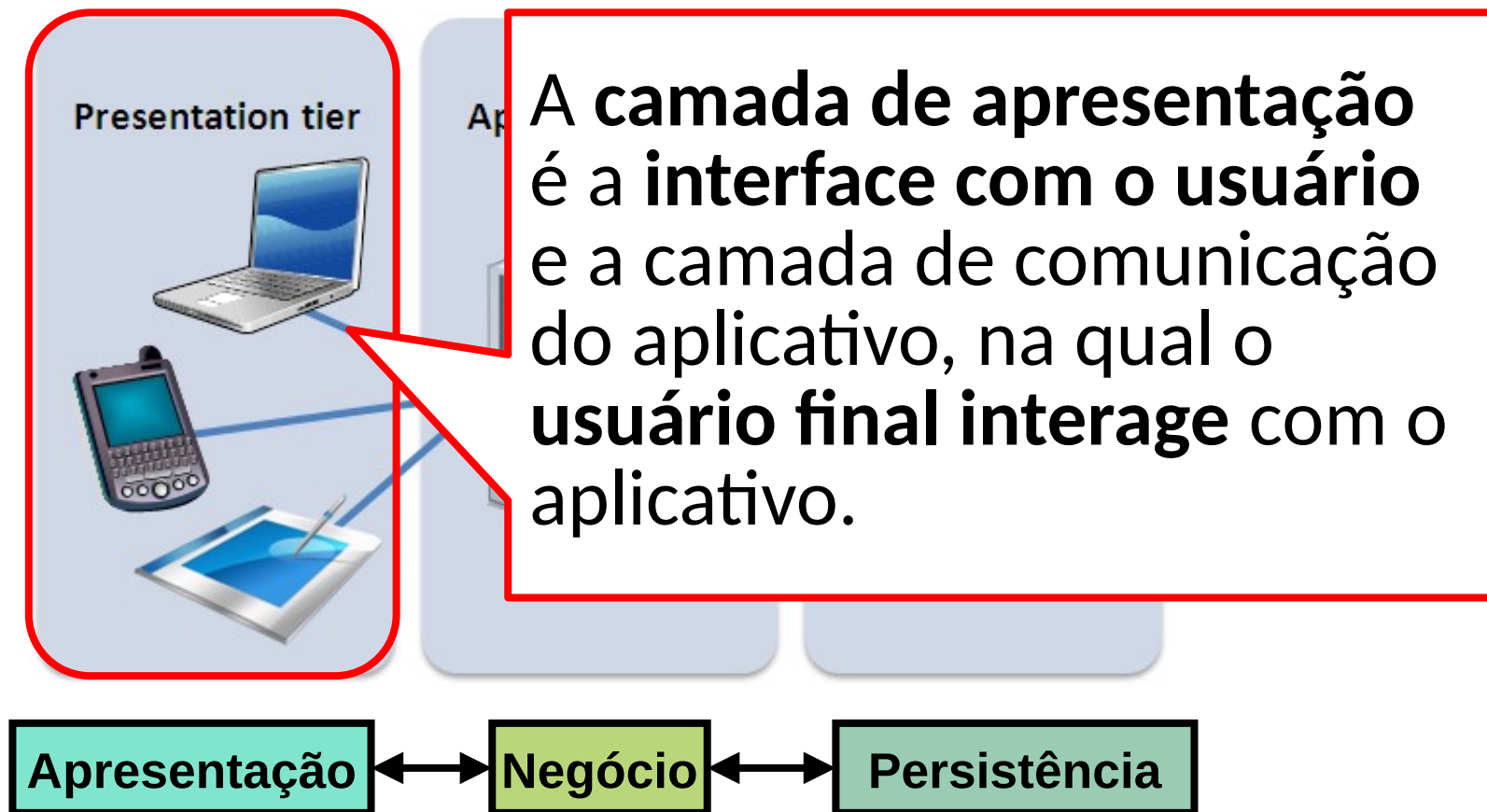
# 1. Arquitetura para Desenvolvimento

O principal **benefício** da arquitetura de três camadas é que devido ao fato de cada camada executar sua própria infraestrutura, cada camada pode ser **desenvolvida simultaneamente** por uma equipe de desenvolvimento separada e pode ser atualizada ou **ajustada conforme necessário** sem impactar as outras camadas.

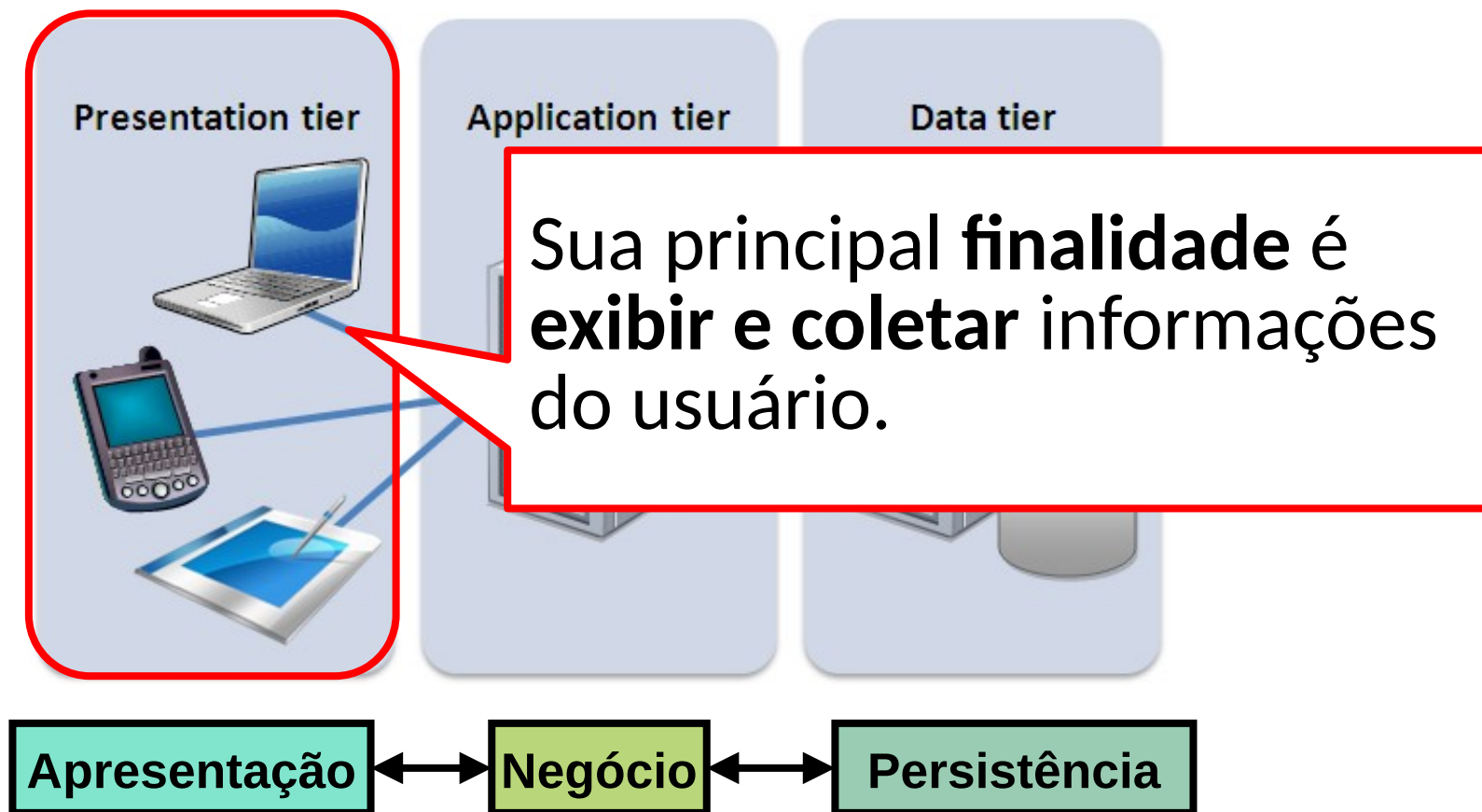




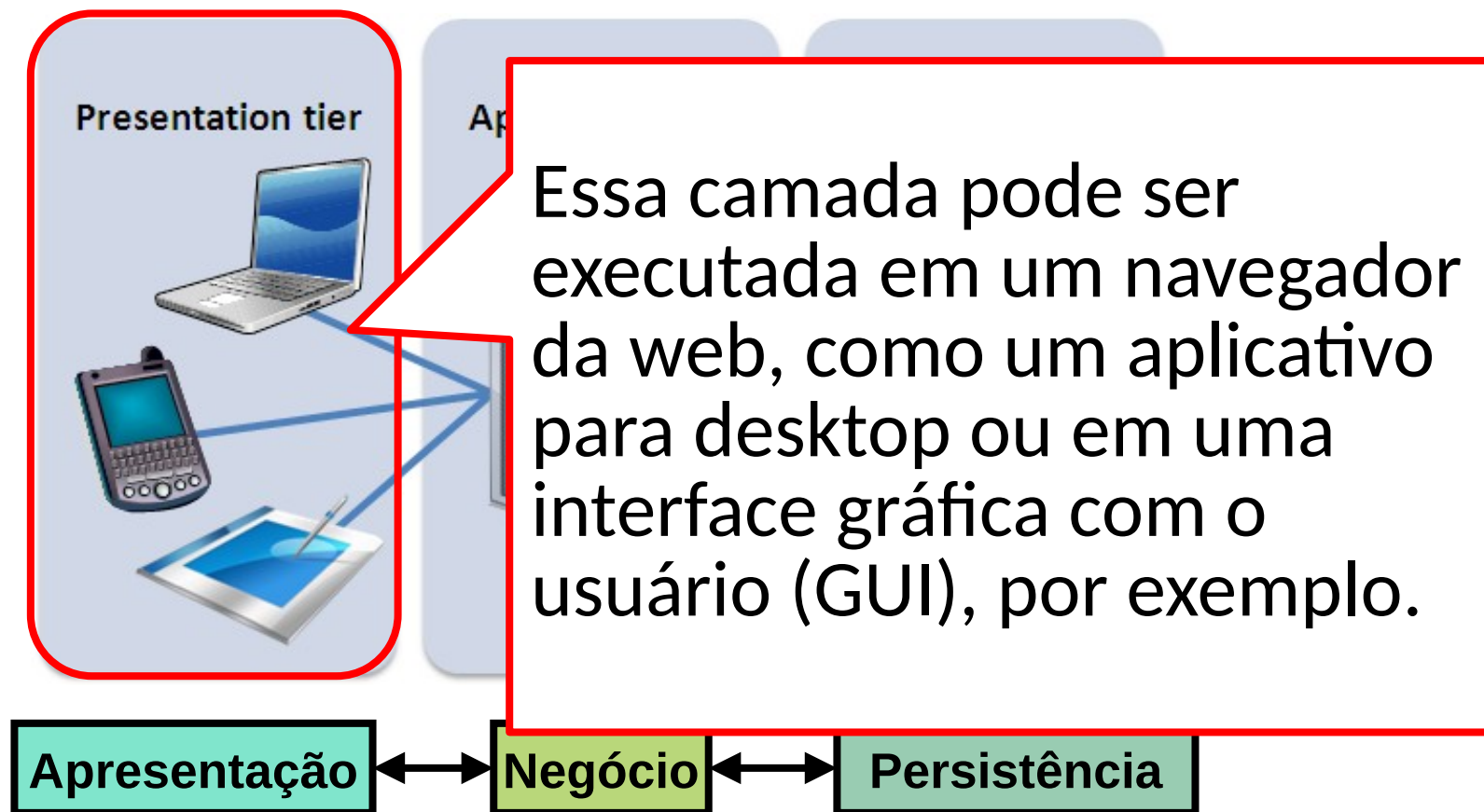
# 1. Arquitetura para Desenvolvimento



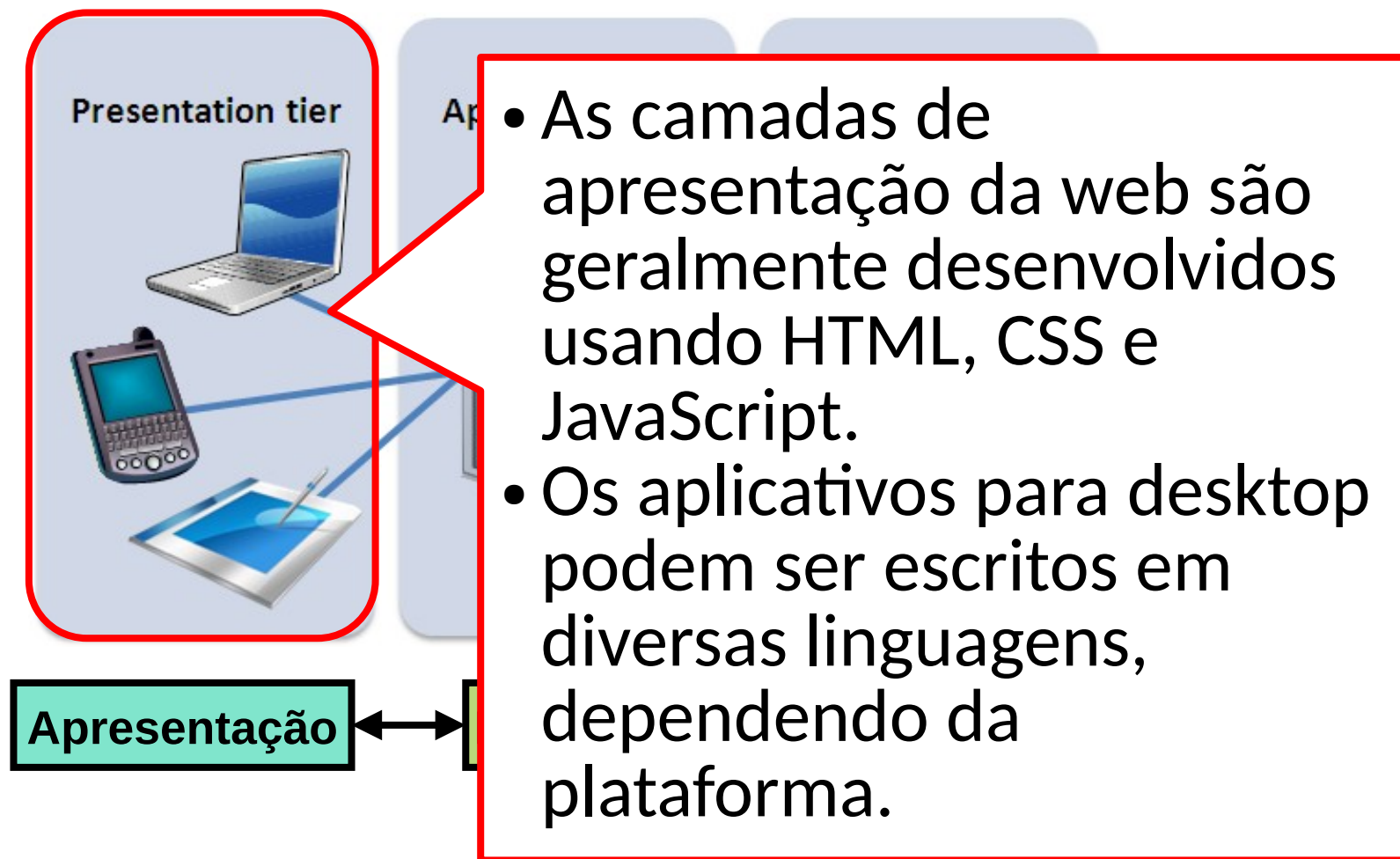
# 1. Arquitetura para Desenvolvimento



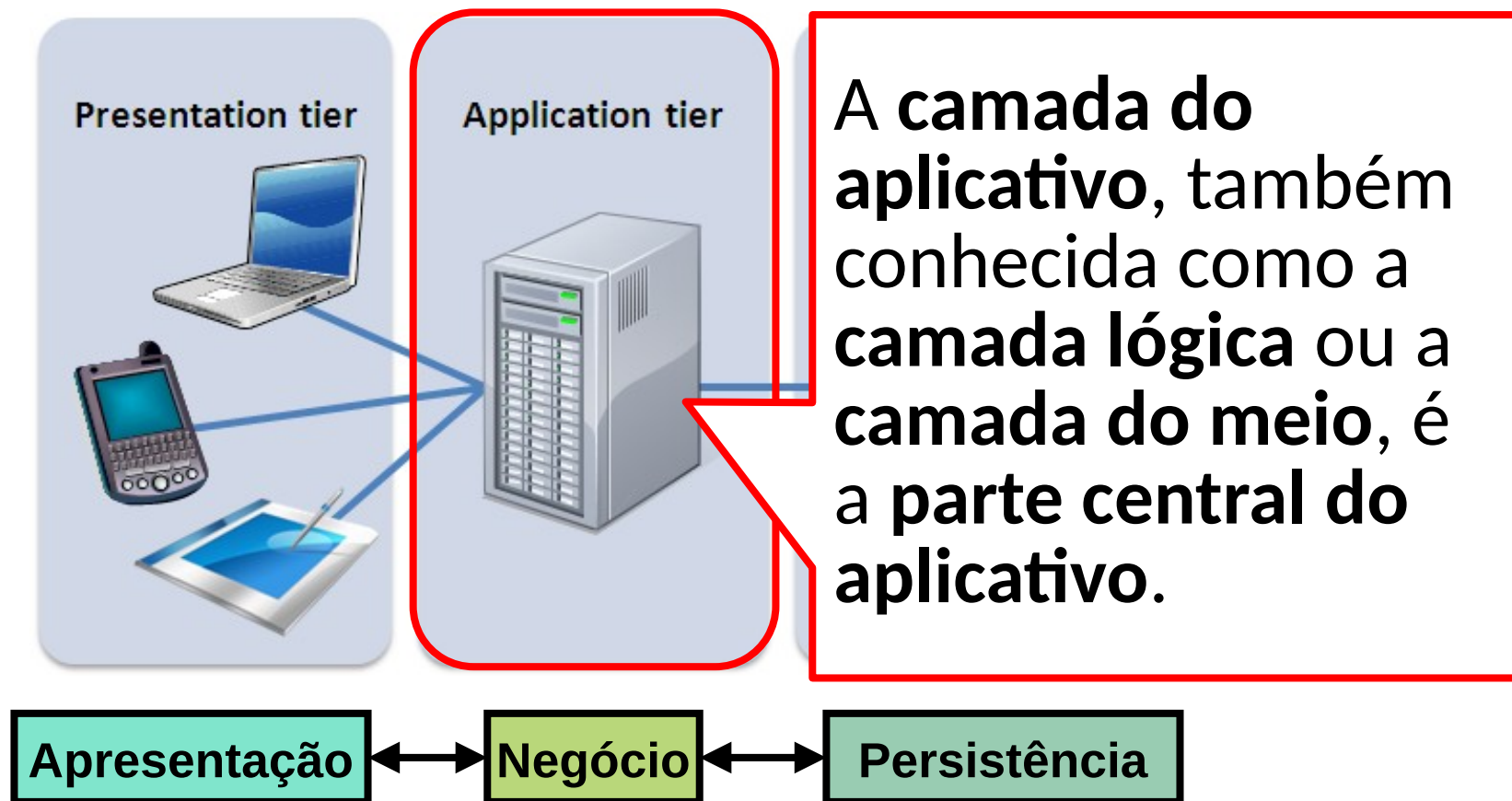
# 1. Arquitetura para Desenvolvimento



# 1. Arquitetura para Desenvolvimento



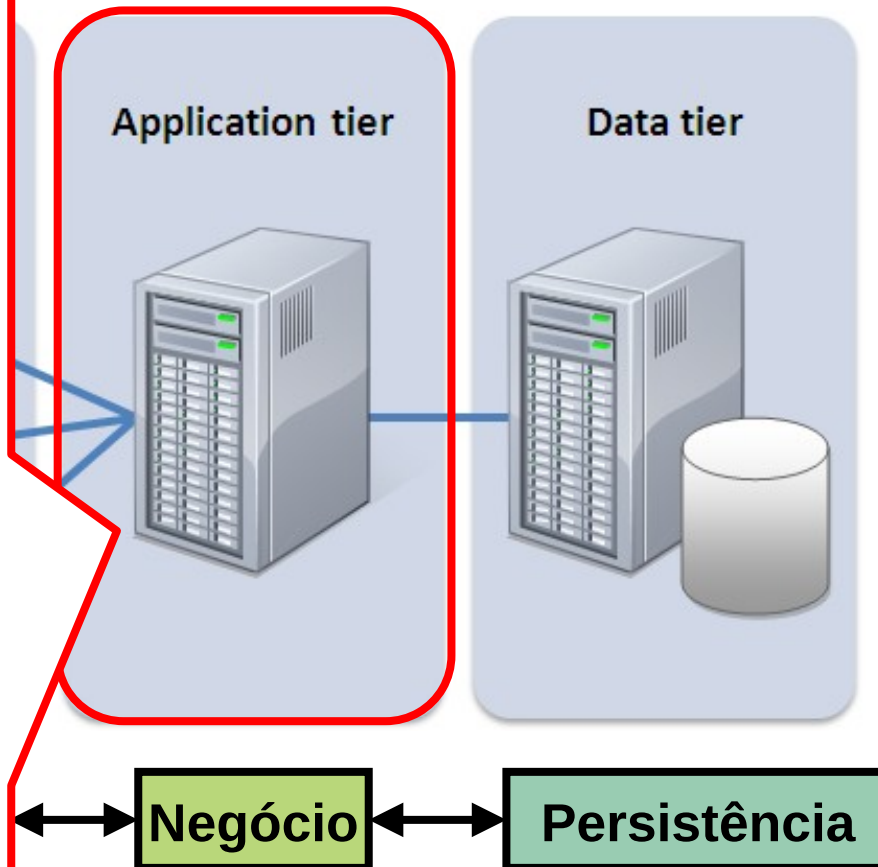
# 1. Arquitetura para Desenvolvimento



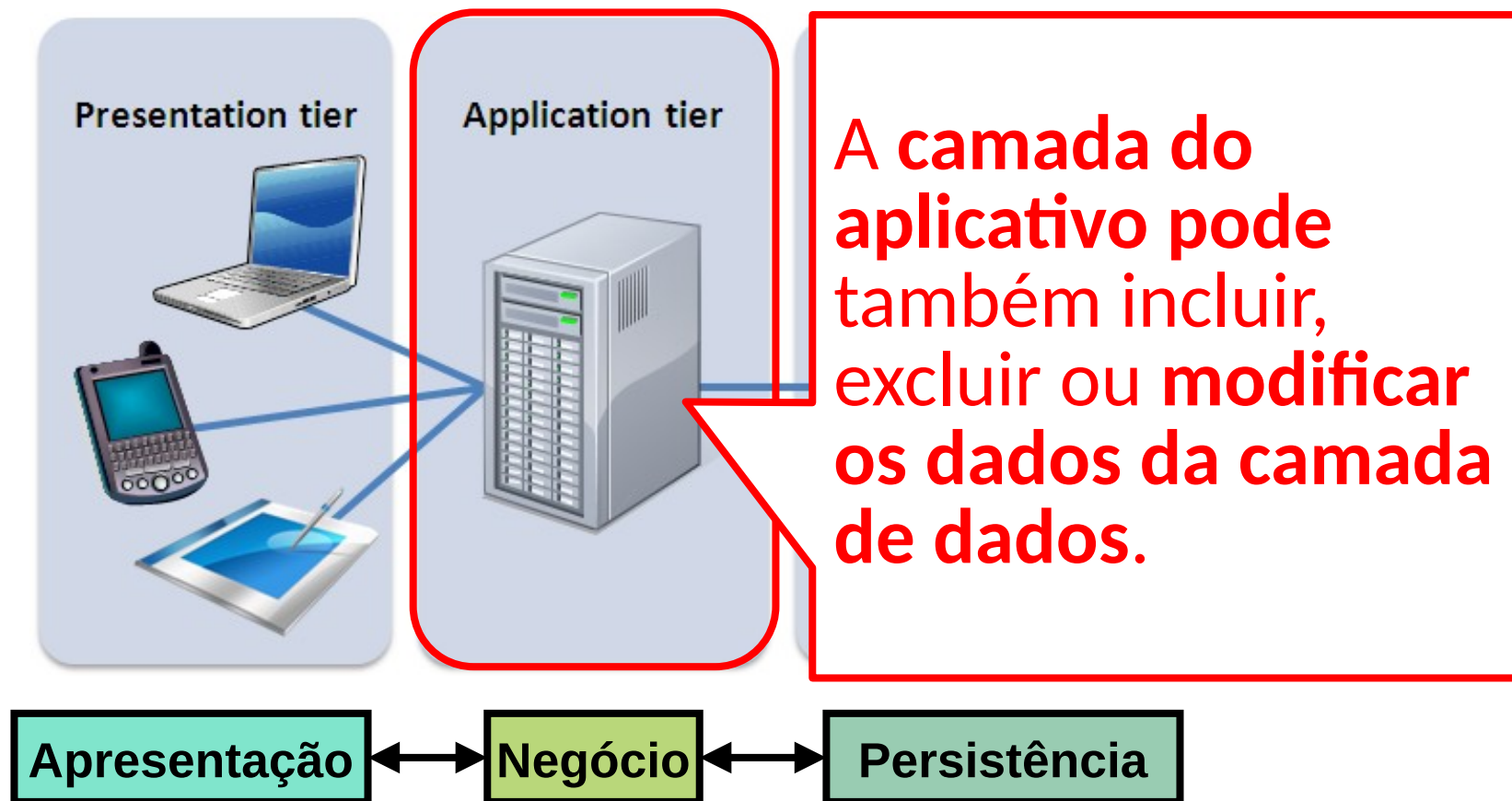


Nessa camada, as **informações** coletadas na camada de apresentação **são processadas**, algumas vezes em relação a outras informações da camada de dados, **usando a lógica de negócios** que é um conjunto específico de **regras de negócios**.

## para Desenvolvimento



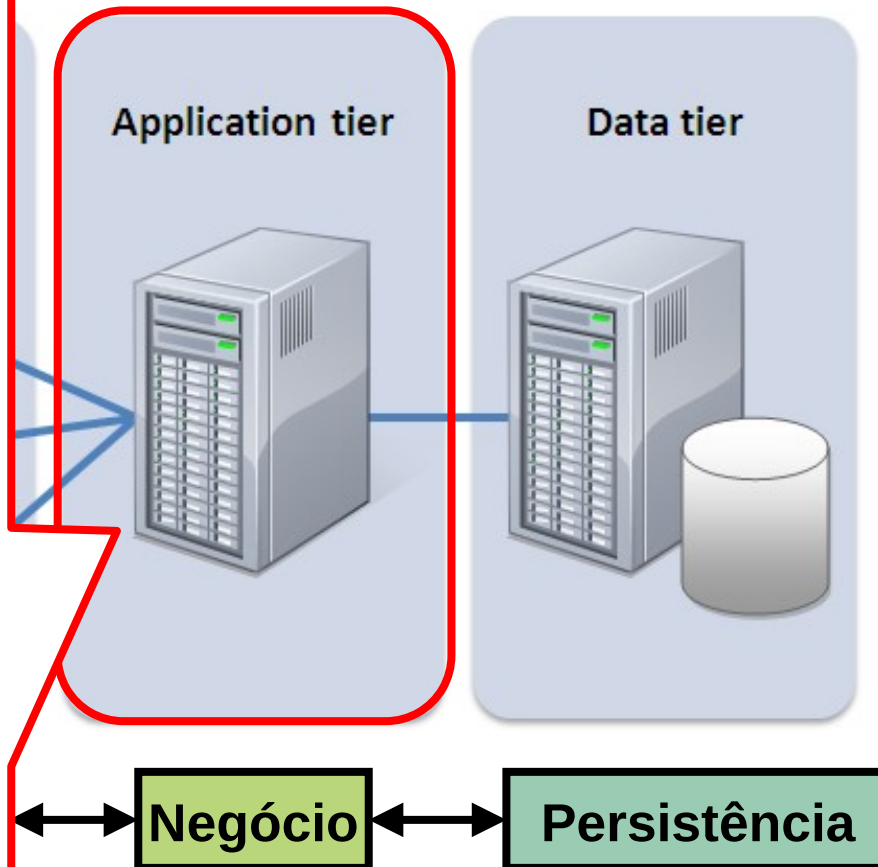
# 1. Arquitetura para Desenvolvimento





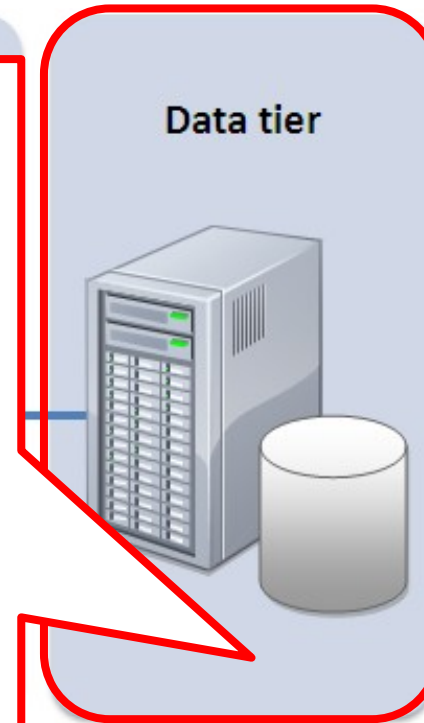
# 1. Arquitetura para Desenvolvimento

A camada do aplicativo é geralmente desenvolvida usando **Python, Java, Perl, PHP, Javascript** ou Ruby e se comunica com a camada de dados usando chamadas de API.



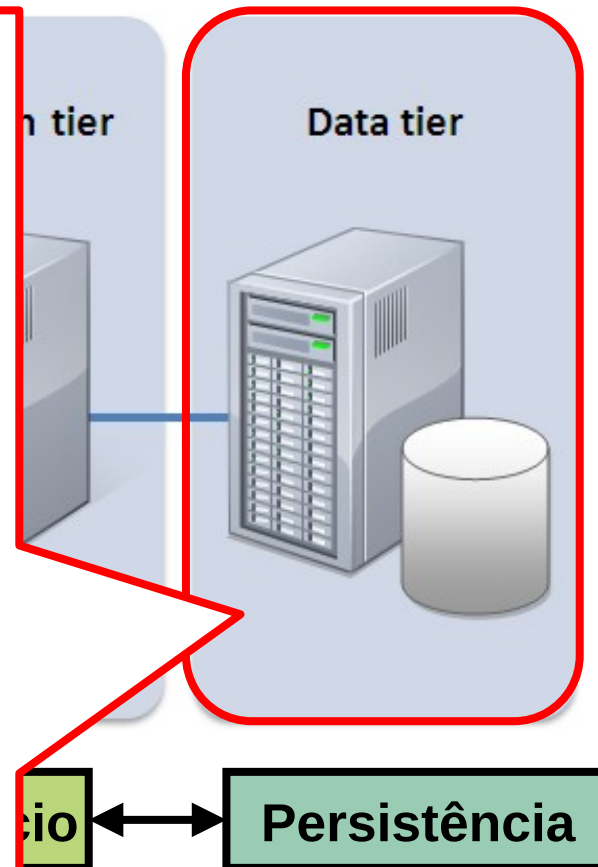
# 1. Arquitetura para Desenvolvimento

A **camada de dados**, por vezes chamada de camada de banco de dados, **camada de acesso a dados ou back-end**, é na qual as informações processadas pelo aplicativo são armazenadas e gerenciadas.



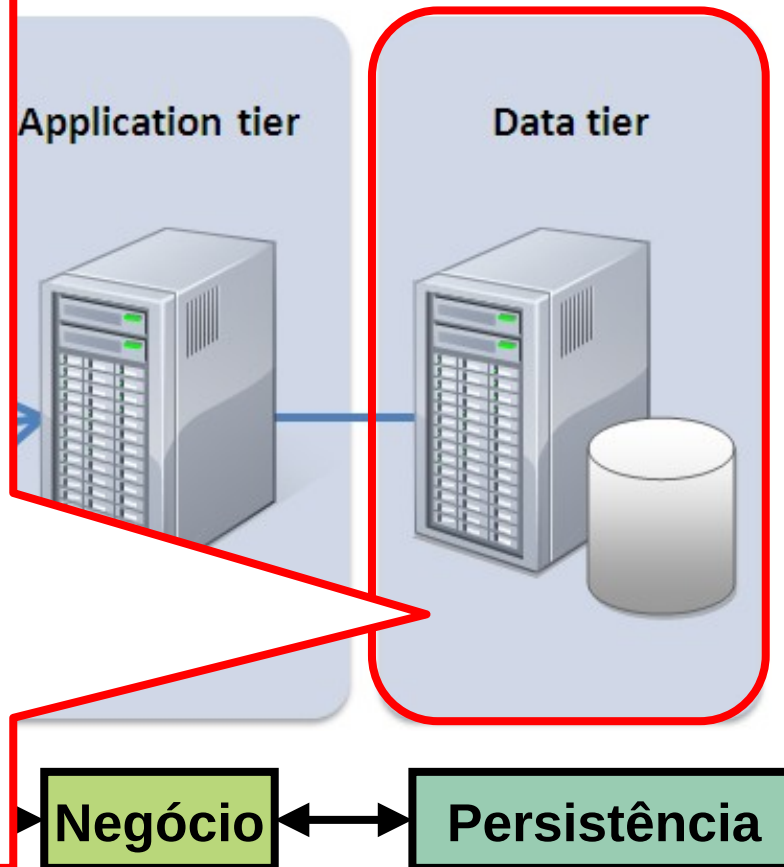
# 1. Arquitetura para Desenvolvimento

Este pode ser um sistema de gerenciamento de banco de dados relacional, como **PostgreSQL, MySQL, MariaDB, Oracle, DB2, Informix** ou **Microsoft SQL Server** ou em um servidor de banco de dados **NoSQL**, como **Cassandra, CouchDB** ou **MongoDB**.



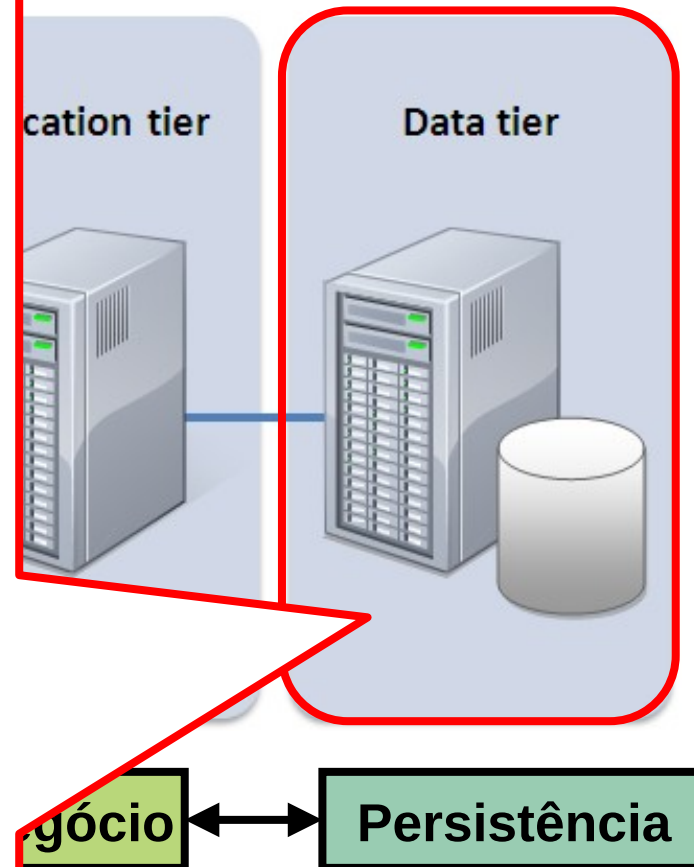
# 1. Arquitetura para Desenvolvimento

Em um aplicativo de três camadas, **toda a comunicação** passa pela camada do aplicativo. A **camada de apresentação** e a **camada de dados** não podem se comunicar diretamente entre si.



# 1. Arquitetura para Desenvolvimento

- Portanto, a separação das abstrações de código deve ser a mais desacoplada possível.
- Assim, padrões de projetos são fundamentais para projetar sistemas de forma mais eficiente, reutilizável e de fácil manutenção.





## 2. Benefícios de três camadas (tiers)

- O principal benefício da arquitetura de três camadas é a separação lógica e física da funcionalidade.
- Cada camada pode ser executada em um sistema operacional e plataforma de servidor diferentes, por exemplo, servidor da web, servidor de aplicativos, servidor de banco de dados, que mais atende aos seus requisitos funcionais.



## 2. Benefícios de três camadas (tiers)

- E cada camada é executada em pelo menos um hardware de servidor dedicado ou servidor virtual, por isso os serviços de cada camada podem ser **customizados** e **otimizados** sem impactar as outras camadas.





## 2. Benefícios de três camadas (tiers)

- Outros benefícios incluem:
  - **Desenvolvimento mais rápido:** uma vez que todas as camadas podem ser desenvolvidas simultaneamente por diferentes equipes, uma organização pode lançar o aplicativo no mercado mais rapidamente e os programadores podem usar as linguagens e ferramentas mais recentes e melhores para cada camada.



## 2. Benefícios de três camadas (tiers)

- Outros benefícios incluem:
  - **Segurança aprimorada:** uma vez que a camada de apresentação e a camada de dados não podem se comunicar diretamente, uma camada do aplicativo bem projetada pode ter a função de uma espécie de firewall interno, impedindo injeções SQL e outros exploradores de vulnerabilidade mal-intencionados.



## 2. Benefícios de três camadas (tiers)

- Outros benefícios incluem:
  - **Escalabilidade:** qualquer camada pode ser dimensionada independentemente das outras conforme necessário.
  - **Maior confiabilidade:** uma indisponibilidade em uma camada é menos propensa a impactar a disponibilidade ou o desempenho das outras camadas.



### 3. Camada (tier) vs. nível (layer)

- Nas discussões sobre arquitetura de três camadas, o termo "layer" é frequentemente utilizado erroneamente como sinônimo de "tier", como em "camada de apresentação" ou "camada de lógica de negócios".



### 3. Camada (tier) vs. nível (layer)

- Eles não são a mesma coisa. Uma "layer" refere-se a uma **divisão funcional do software**, mas um "tier" refere-se a uma **divisão funcional do software que é executada em infraestrutura separada** das outras divisões.
- O aplicativo de Contatos no seu telefone, por exemplo, é uma aplicação de três camadas (*three-layer*), mas é uma aplicação de um único "tier", porque todas as três *layers* são executadas no seu telefone.

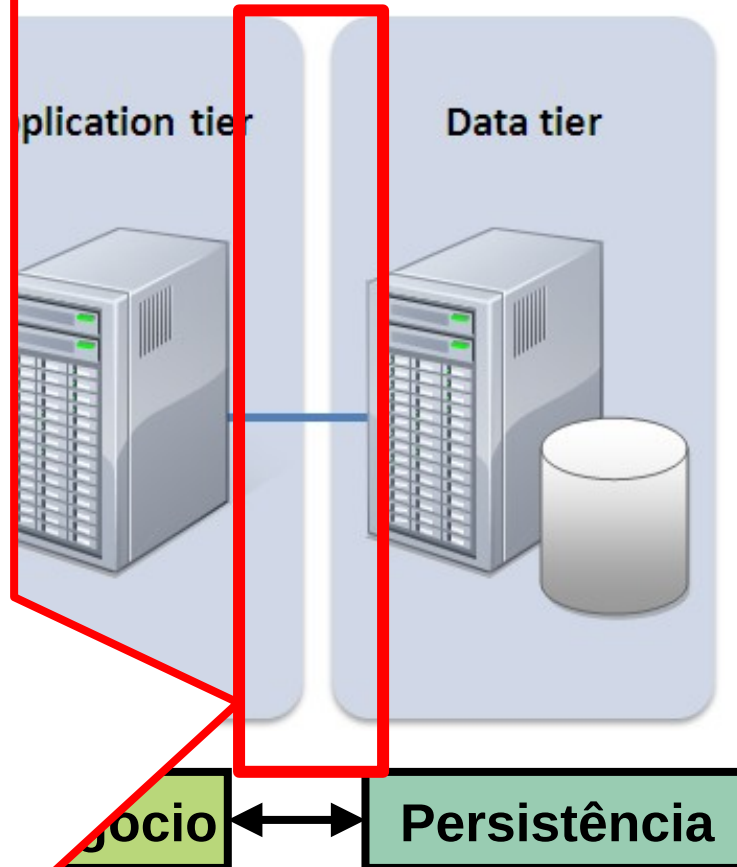


## 4. Camada de persistência

- A camada de persistência (*persistence layer*), também conhecida como camada de acesso a dados ou camada de persistência de dados, é um componente de um aplicativo de software ou sistema responsável por armazenar e recuperar dados de um armazenamento persistente, como um banco de dados ou sistema de arquivos.

## 4 Camadas de Persistência

A camada de **persistência** atua como **intermediária** entre a **camada de lógica de negócios do aplicativo** e o **armazenamento de dados subjacente**. Ela **encapsula as operações** e mecanismos necessários para interagir com o armazenamento de dados, incluindo **leitura, escrita, atualização e exclusão de dados**.







## 4. Camada de persistência

- O **objetivo** principal da camada de persistência é fornecer uma **forma contínua e consistente** de acessar e manipular dados, **abstraindo os detalhes específicos da tecnologia de armazenamento** de dados subjacente.
- Ela garante que o aplicativo possa trabalhar com os dados **sem estar fortemente acoplado à implementação** específica de armazenamento de dados.



## 4. Camada de persistência

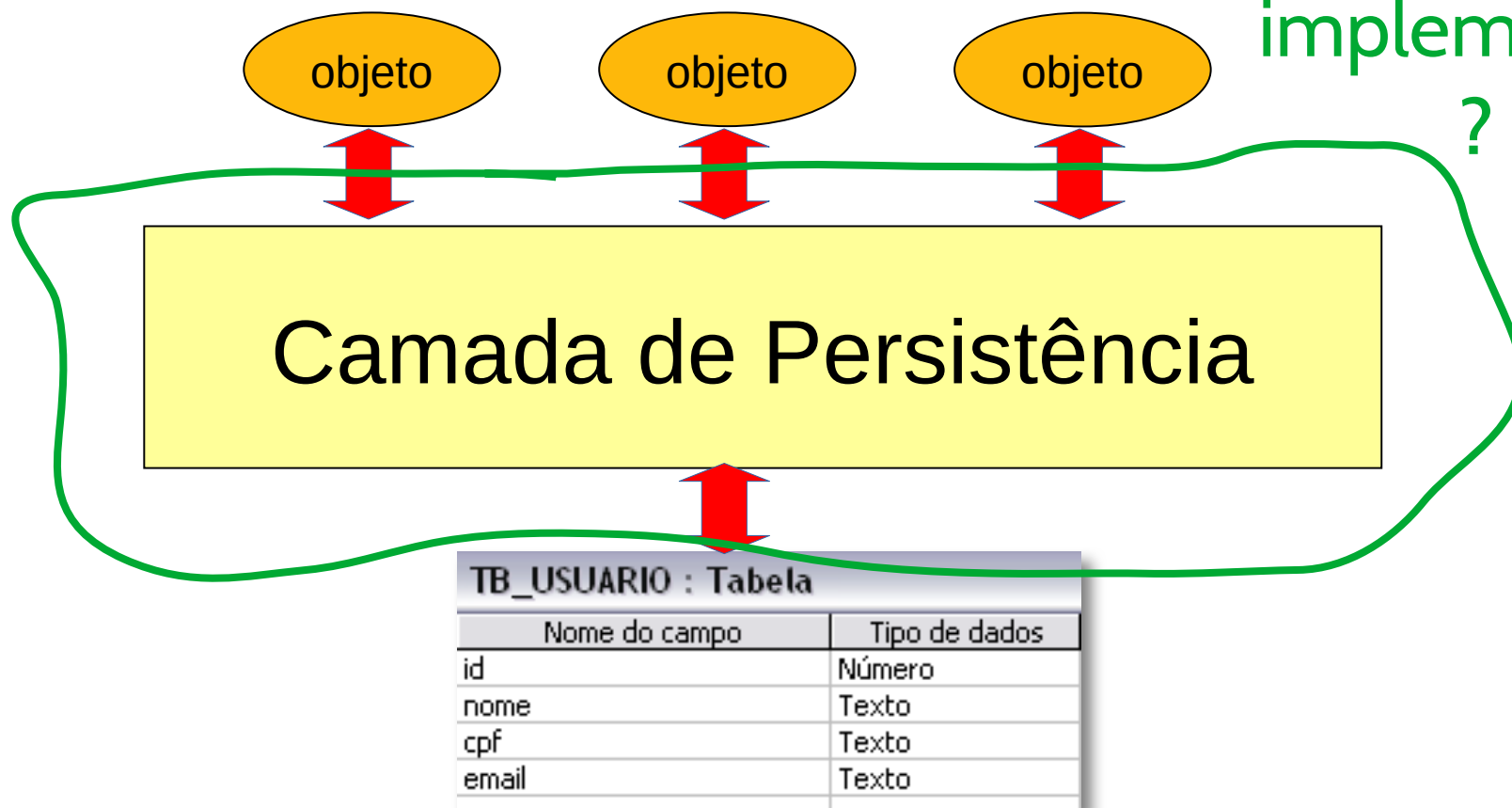
- Ao separar as preocupações de persistência do restante do aplicativo, a **camada de persistência permite um design modular**, facilidade de **manutenção e escalabilidade**.
- Também permite que **diferentes tecnologias de armazenamento** de dados sejam usadas de forma intercambiável, **desde que sigam a mesma interface** fornecida pela camada de persistência.

## 4. Camada de persistência



## 4. Camada de persistência

Como  
implementar  
?





## 4. Camada de persistência

- Existem várias alternativas para implementar o acesso e persistência de dados em aplicativos de software. Algumas das alternativas comumente usadas incluem:
  - Data Access Object (DAO)
  - Object-Relational Mapping (ORM) frameworks
  - Repository Pattern
  - Active Record Pattern
  - Query Builders



## 4. Camada de persistência

- **Object-Relational Mapping (ORM) frameworks**
  - ORM (Object Relational Mapper) é uma técnica de **mapeamento objeto relacional** que permite fazer uma relação dos objetos com os dados que os mesmos representam.
  - Ultimamente tem sido muito utilizada e vem crescendo bastante nos últimos anos.
  - Este crescimento tem se dado principalmente pelo fato de muitos desenvolvedores **não se sentirem a vontade em escrever código SQL.**

4

Portanto, **não** deve ser utilizado no escopo desta matéria.

a

M) frameworks

per) é uma

to relacional

que pode fazer uma relação dos objetos com dados que os mesmos representam.

- Ultimamente tem sido muito utilizada e vem crescendo bastante nos últimos anos.
- Este crescimento tem se dado principalmente pelo fato de muitos desenvolvedores **não se sentirem a vontade em escrever código SQL.**





## 4. Camada de

**Não** deve ser utilizado no escopo desta matéria.

- Object-Relational

- Principais ORM

- **Java:** Hibernate, A
- **Kotlin:** Exposed
- **C#:** Entity, Nhibernate
- **Node:** Sequelize
- **PHP:** Doctrine, Eloquent
- **Ruby:** Ruby On Rails ActiveRecord
- **Python:** DjangoORM

## 4. Camada de persistência

- **Active Record Pattern**

- É um design pattern comumente utilizado como interface no contexto ORM.
- No padrão Active Record, cada tabela de banco de dados é representada por uma classe correspondente no código.
- Essa classe contém os campos da tabela como atributos e também possui métodos para recuperar, salvar e atualizar os dados no banco de dados.



## 4. Camada de persistência

- **Active Record Pattern**
  - Em termos simples, ele permite que você trabalhe com dados de um banco de dados como se estivesse manipulando objetos em uma linguagem de programação orientada a objetos.



## 4. Camada de persistência

- **Active Record Pattern**
  - O padrão Active Record abstrai as operações de banco de dados e permite que você manipule os dados usando uma interface orientada a objetos, em vez de escrever consultas SQL diretamente.



## 4. Camada de persistência

- Active Record Pattern

- Ou seja, o padrão Active Record permite que você trabalhe com dados de um banco de dados por meio de objetos, fornecendo uma camada de abstração que simplifica a manipulação e o acesso aos dados.



## 4. Camada de persistência

- Active Record Pattern

- Por exemplo, se tivermos uma tabela chamada “`User`” com campos como “`nome`”, “`email`” e “`senha`”, poderíamos criar uma classe chamada “`User`” na base de código.
- Essa classe teria os **atributos** nome, e-mail e senha, e também **métodos** para **recuperar** usuários do banco de dados, **salvar** novos usuários e **atualizar** informações existentes.



## 4. Camada de persistência

- Active Record Pattern

- Por exemplo (Trecho de código simplificado em Python que ilustra o **exemplo de uso** "User" usando o padrão Active Record):

```
user = User("Pedro", "pedro@example.com", "password123")
user.save()

retrieved_user = User.find_by_email("pedro@example.com")
if retrieved_user:
    print("User found:", retrieved_user.name)
else:
    print("User not found.")
```



## 4. Camada

Neste exemplo, a classe “User” representa a **tabela “Users”**.

- Active Record

- Por exemplo

em Python que usa o **exemplo de uso** “User” usando o padrão Active Record):

```
user = User("Pedro", "pedro@example.com", "password123")
user.save()
```

```
retrieved_user = User.find_by_email("pedro@example.com")
if retrieved_user:
    print("User found:", retrieved_user.name)
else:
    print("User not found.")
```





## 4. Camada

- Active Record

- Por exemplo

em Python que

"User" usando

Nesta linha, o novo **objeto** "User" criado com os atributos "nome", "email" e "senha" preenchidos.

o **exemplo de uso** "User" usando o padrão Active Record):

```
user = User("Pedro", "pedro@example.com", "password123")
user.save()
```

```
retrieved_user = User.find_by_email("pedro@example.com")
if retrieved_user:
    print("User found:", retrieved_user.name)
else:
    print("User not found.")
```



## 4. Camada

- Active Record

- Por exemp

em Python que

"User" usando

A chamada o método **save()** salva os atributos no banco de dados.

o exemplo de uso

padrão Active Record):

```
user = User("pedro", "pedro@example.com", "password123")
user.save()
```

```
retrieved_user = User.find_by_email("pedro@example.com")
if retrieved_user:
    print("User found:", retrieved_user.name)
else:
    print("User not found.")
```



## 4. Camada

- Active Record

- Por exemplo, em Python que il "User" usando o

Então, o método **find\_by\_email()** é usado para recuperar o usuário com base no email.

exemplo de uso  
o Active Record):

```
user = User("Pedro", "pedro@example.com", "password123")
user.save()

retrieved_user = User.find_by_email("pedro@example.com")
if retrieved_user:
    print("User found:", retrieved_user.name)
else:
    print("User not found.")
```



## 4. Consultas de usuário

- Por fim, o **nome do usuário** recuperado é impresso caso o usuário exista, ou uma mensagem indicando que o usuário não foi encontrado.

simplificado  
o de uso  
Record):

```
user = User("Pedro", "pedro@example.com", "password123")
user.save()

retrieved_user = User.find_by_email("pedro@example.com")
if retrieved_user:
    print("User found:", retrieved_user.name)
else:
    print("User not found.")
```



## 4. Camada de persistência

- Active Record Pattern
  - Implementação da classe **User**

```
class User:
    def __init__(self, name, email, password):
        self.name = name
        self.email = email
        self.password = password
        self._connection = sqlite3.connect('database.db')

    def save(self):
        cursor = self._connection.cursor()
        cursor.execute(
            "INSERT INTO Users (name, email, password) VALUES (?, ?, ?)",
            (self.name, self.email, self.password))
        self._connection.commit()

    def find_by_email(self, email):
        cursor = self._connection.cursor()
        cursor.execute(
            "SELECT name, email, password FROM Users WHERE email=?",
            (email,))
        user_data = cursor.fetchone()
        if user_data:
            name, email, password = user_data
            return User(name, email, password)
        else:
            return None
```

## 4. Camada de persistência

- Repository Pattern

- É um padrão de projeto semelhante ao Active Record, mas que separa as responsabilidades de acesso e manipulação de dados em uma camada de repositório dedicada.
- A diferença chave entre o padrão Active Record e o padrão Repository é a maneira como eles abordam a separação de responsabilidades no acesso e manipulação de dados.



## 4. Camada de persistência

- Repository Pattern
  - O **Active Record** coloca essas responsabilidades em cada objeto individual, enquanto o **Repository** centraliza essas responsabilidades em uma camada dedicada.





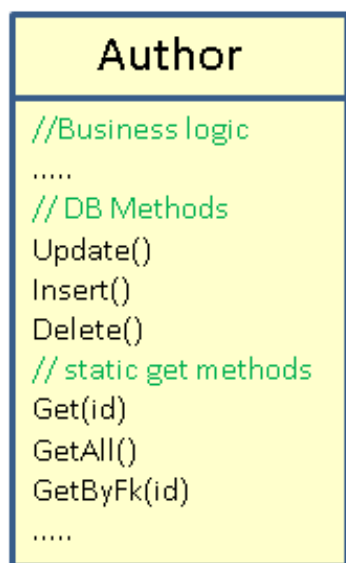
## 4. Camada de persistência

- Repository Pattern

- Em resumo, O padrão **Active Record** define **um objeto que envolve uma linha em uma tabela** ou exibição de um banco de dados, encapsula o acesso aos dados e adiciona lógica de domínio a esses dados (a classe corresponde à linha).
- No padrão Repository, **todo o acesso aos dados é colocado em uma classe separada** e é acessado por meio de métodos de instância. Para mim, apenas fazer isso é benéfico, pois o acesso aos dados agora está encapsulado em uma classe separada, permitindo que o objeto de negócio se concentre nos negócios. Isso deve evitar a infeliz mistura de acesso aos dados e lógica de negócios que geralmente ocorre com o Active Record.

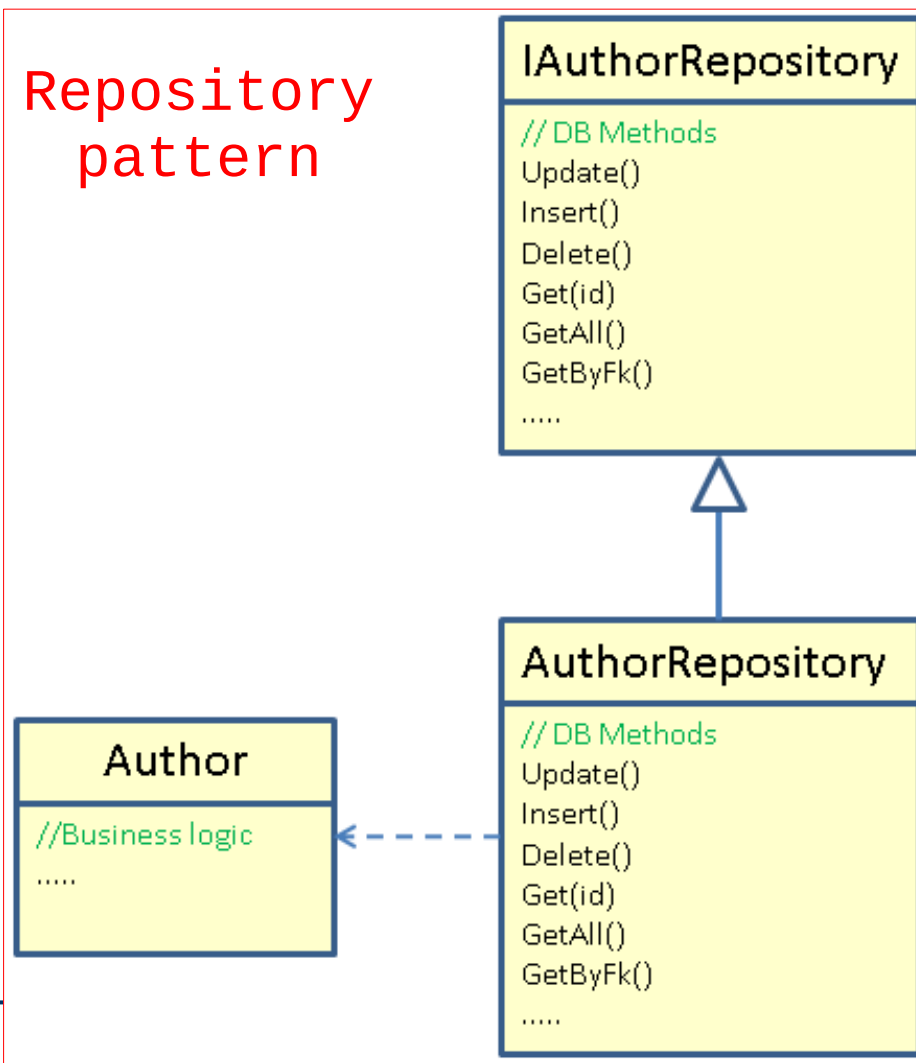
## 4. Camada de persistência

- Repository Pattern



Active Record

Repository  
pattern





## 5. Data Access Object (DAO)

- DAO é um objeto responsável por **persistir uma entidade separada** em seu domínio (i.e., outra classe).
- O Active Record, visto anteriormente, é um método específico de implementar um DAO em que a classe que contém os valores de uma única linha de uma tabela também é responsável por consultas, atualizações, inserções e exclusões nessa tabela.



## 5. Data Access Object (DAO)

- Em outras palavras, o DAO é uma interface dedicada à persistência de um objeto modelo/domínio em uma fonte de dados.
- O padrão ActiveRecord funciona de maneira semelhante, mas coloca os **métodos de persistência diretamente no objeto do modelo**, enquanto o DAO define uma interface discreta.

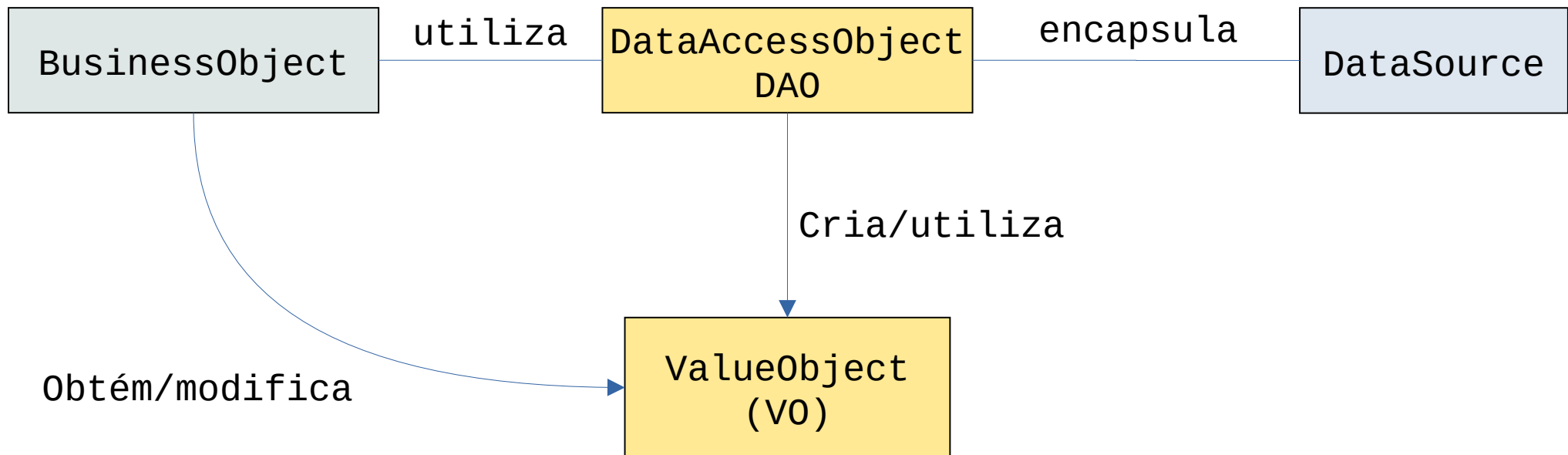


## 5. Data Access Object (DAO)

- Vantagem do padrão DAO é que é **fácil definir outro estilo de persistência**, por exemplo, migrar de um banco de dados para a nuvem, sem alterar a implementação subjacente, enquanto a interface externa permanece a mesma, portanto, sem afetar outras classes.
- As preocupações de persistência são **modularizadas e separadas** das preocupações principais do objeto do modelo.



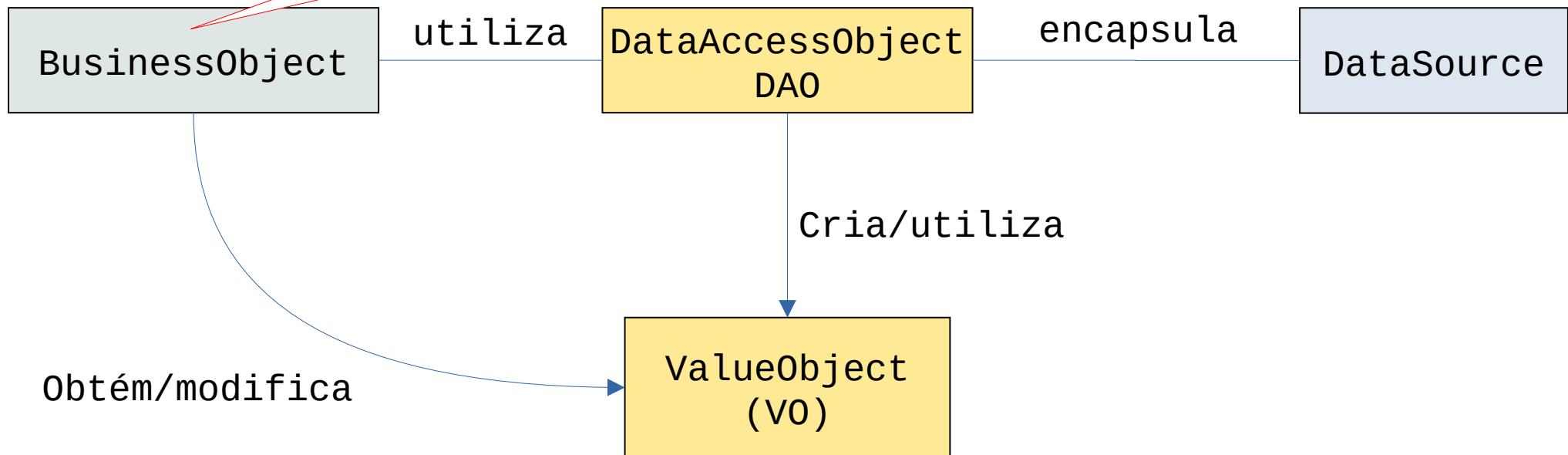
## 5. Data Access Object (DAO)





## 5. Data Access

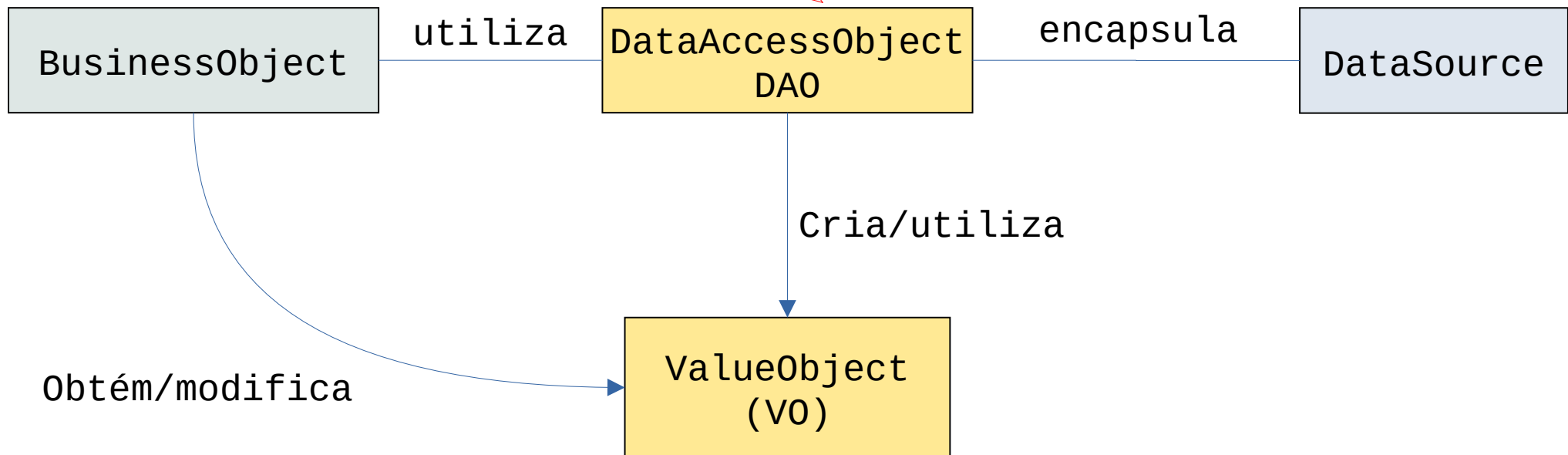
**BusinessObject: o cliente do dados**





## 5. Data Access

**DataAccessObject:**  
encapsula o acesso a  
fonte de dados

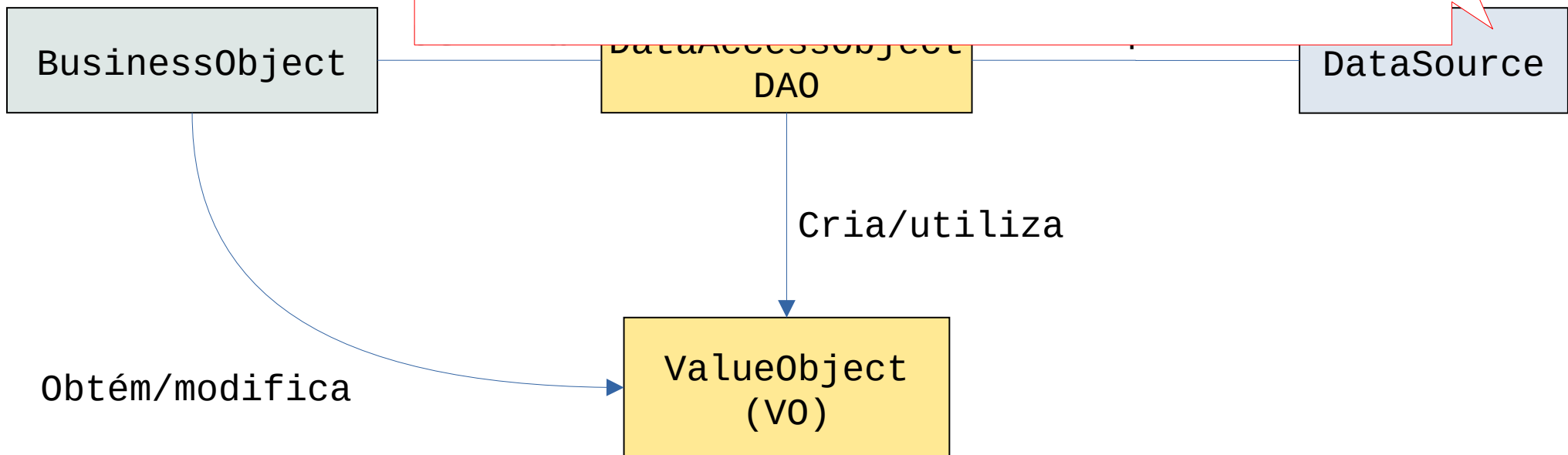






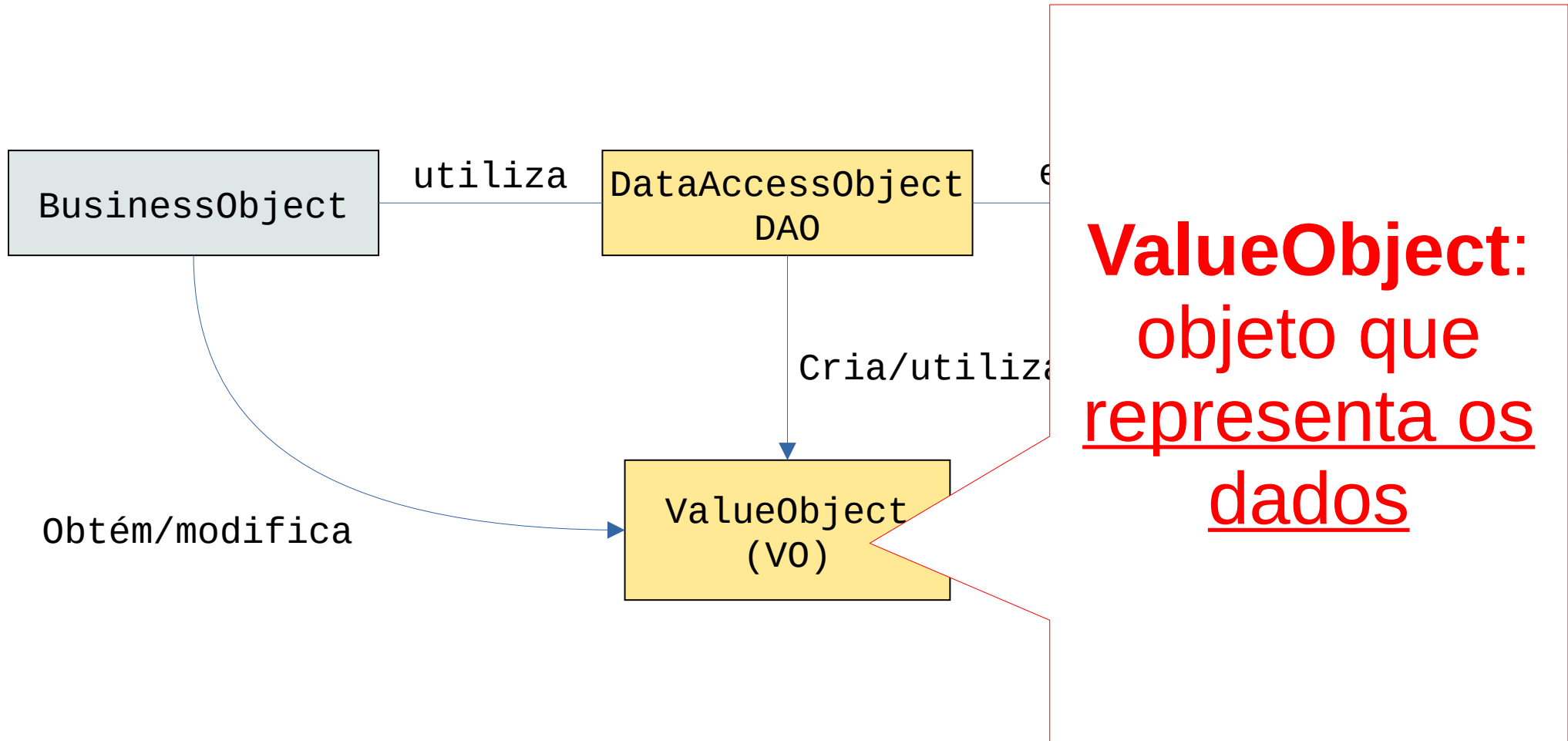
## 5. Data Access Object (DAO)

**DataSource: fonte de dados  
(e.g., SGBD)**





## 5. Data Access Object (DAO)





## 5. Data Access Object (DAO)

```
public class ObjetoDAO{
    public boolean inserirUmObjeto(Objeto o){
        // INSERT
    }

    public boolean apagarObjeto(Objeto o){
        //DELETE
    }

    public boolean apagarObjeto(chavePrimaria){
        //DELETE
    }

    public boolean atualizarObjeto(Objecto object){
        //UPDATE
    }

    public Objeto buscarPorCHavePrimariaObjeto(int chavePrimaria){
        // SELECT
    }

    public Collection buscarTodosObjeto(){
        // SELECT
    }
}
```

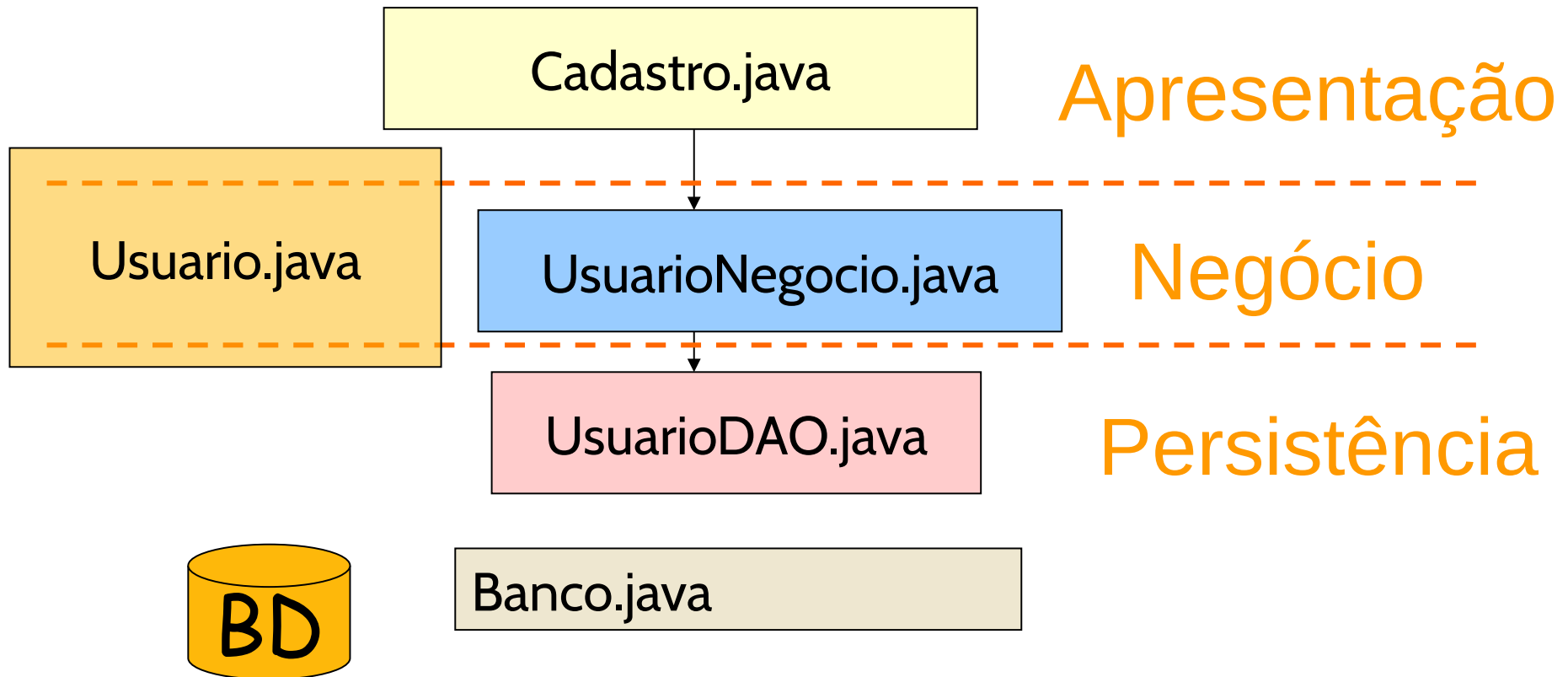


## 5. Data Access Object (DAO)

```
public class ObjetoDAO{  
    public boolean inserirUmObjeto(Objeto o){  
        // INSERT  
    }  
  
    public boolean apagarObjeto(Objeto o){  
        //DELETE  
    }  
  
    public boolean apagarObjeto(chavePrimaria){  
        //DELETE  
    }  
  
    public boolean atualizarObjeto(Objecto object){  
        //UPDATE  
    }  
  
    public Objeto buscarPorCHavePrimariaObjeto(int chavePrimaria){  
        // SELECT  
    }  
  
    public Collection buscarTodosObjeto(){  
        // SELECT  
    }  
}
```

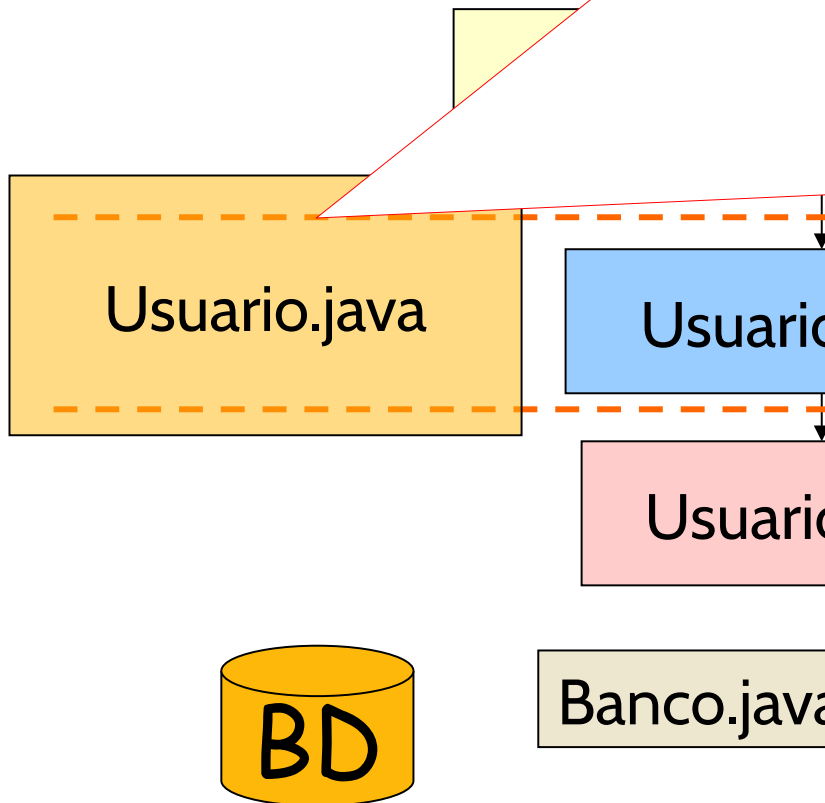
Substitua “Objeto” por qualquer entidade modelada pelo sistema que tenha correspondente no banco de dados (e.g., User, Servidor, Funcionario, Departamento, etc)

## 6. Exemplo:





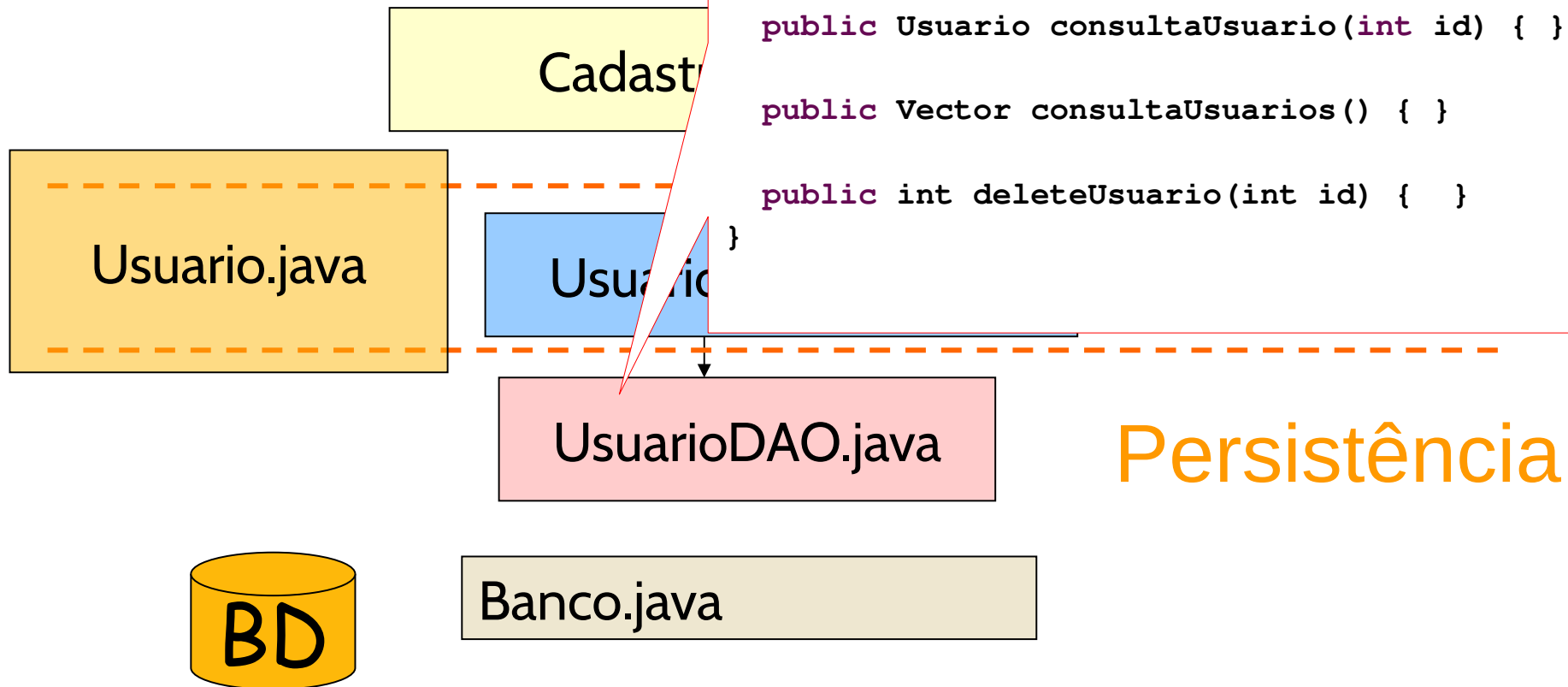
## 6. Exemplo:



```
public class Usuario {  
  
    public Usuario() {}  
  
    public Usuario(int id, String nome,  
                    String endereco, String email) {  
        this.setId(id);  
        this.setNome(nome);  
        this.setEndereco(endereco);  
        this.setEmail(email);  
    }  
  
    private int id;  
    private String nome;  
    private String endereco;  
    private String email;  
  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String newName) {  
        nome = newName;  
    }  
  
    public void setEmail(String newEmail) {  
        email = newEmail;  
    }  
    public String getEmail() {  
        return email;  
    }  
}
```



## 6. Exemplo:





## 6. Ex

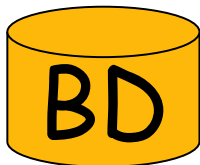
```
public class Banco {  
    private static String DRIVER_BD = "sun.jdbc.odbc.JdbcOdbcDriver";  
    private static String URL_BD = "jdbc:odbc:agenda";  
    private static String usuario = "";  
    private static String senha = "";  
    private static Connection con;  
  
    public Banco() {  
  
    }  
  
    public static Connection getConexao() {  
    }  
}
```

0

Usu }

oDAO.java

Banco.java

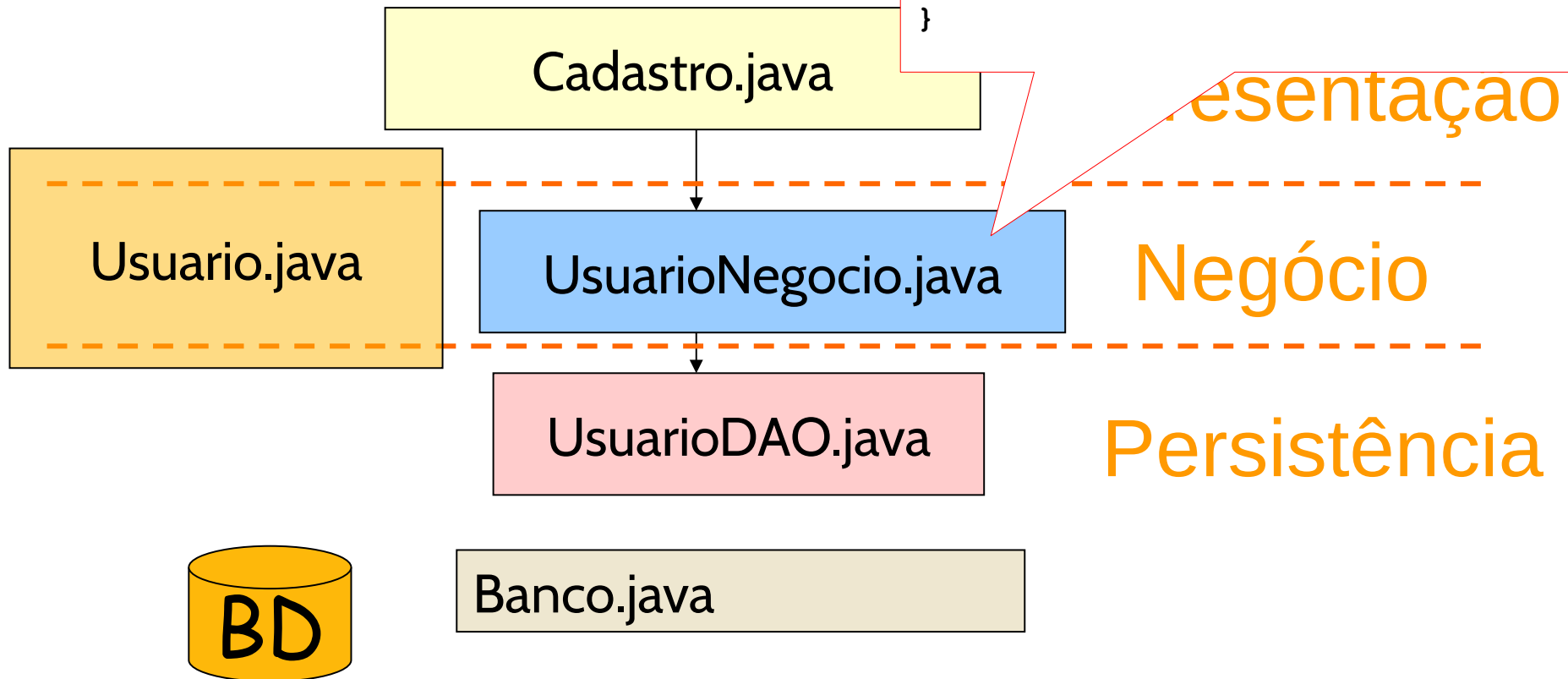


# Persistência





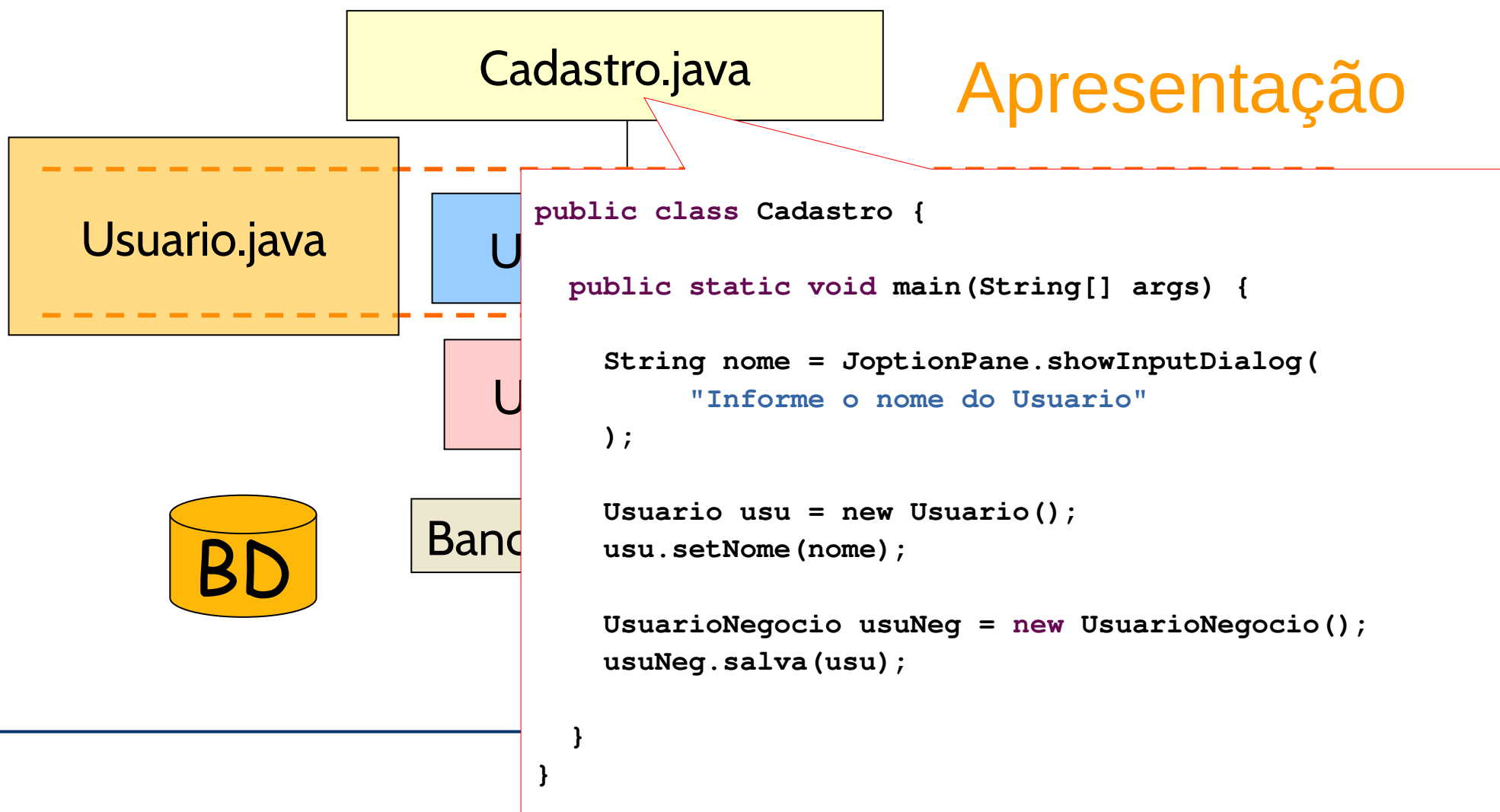
## 6. Exemplo:



```
public class UsuarioNegocio {  
  
    public UsuarioNegocio() {}  
  
    public void salva(Usuario u) {  
        UsuarioDAO udao = new UsuarioDAO();  
        udao.insereUsuario(u);  
    }  
}
```



## 6. Exemplo:





## 6.1 Cadastro.java

```
public class Cadastro {  
  
    public static void main(String[] args) {  
        String nome = JOptionPane.showInputDialog("Informe o nome do Usuario");  
  
        Usuario u = new Usuario();  
        u.setNome(nome);  
  
        UsuarioNegocio un = new UsuarioNegocio();  
        un.salva(u);  
  
    }  
}
```



## 6.2 UsuarioNegocio.java

```
public class UsuarioNegocio {  
  
    public UsuarioNegocio() {}  
  
    public void salva(Usuario u) {  
        UsuarioDAO usuBD = new UsuarioDAO();  
        usuBD.insereUsuario(u);  
    }  
}
```



## 6.3 Banco.java

```
import java.sql.*;

public class Banco {
    private static String DRIVER_BD = "sun.jdbc.odbc.JdbcOdbcDriver";
    private static String URL_BD = "jdbc:odbc:agenda";
    private static String usuario = "";
    private static String senha = "";
    private static Connection con;

    public Banco() {
        try {
            Class.forName(DRIVER_BD);
        } catch (ClassNotFoundException e) {
            System.out.println("Problemas ao carregar o driver");
        }
    }
}
```



## 6.3 Banco.java

```
public static Connection getConexao() {  
    try {  
        if ((con == null) || (con.isClosed()))  
            con = DriverManager.getConnection(URL_BD, usuario, senha);  
    } catch (SQLException e) {  
        System.out.println("Problemas ao abrir a conexao com o BD");  
    }  
    return con;  
}
```



## 6.4 Usuario.java

```
public class Usuario {  
  
    public Usuario() {}  
  
    public Usuario(int id, String nome,  
                   String endereco, String email) {  
        this.setId(id);  
        this.setNome(nome);  
        this.setEndereco(endereco);  
        this.setEmail(email);  
    }  
}
```



## 6.4 Usuario.java

```
private int id;
private String nome;
private String endereco;
private String email;

public String getNome() {
    return nome;
}
public void setNome(String newName) {
    nome = newName;
}

public void setEmail(String newEmail) {
    email = newEmail;
}
public String getEmail() {
    return email;
}
}
```





## 6.5 UsuarioDAO.java

```
import java.sql.*;

public class UsuarioDAO {
    public UsuarioDAO() {
        Banco banco = new Banco();
    }

    public void insereUsuario(Usuario usu) {
        try {
            Connection con = Banco.getConexao();
            Statement stmt = con.createStatement();
            String comandoSQL = "INSERT INTO TB_USUARIO(NOME, ENDERECO, EMAIL) " +
                " VALUES ('" + usu.getNome() + "', '" + usu.getEndereco() + "', '" +
usu.getEmail() + "' ) ";
            stmt.executeUpdate(comandoSQL);
            stmt.close();
            con.commit();
            con.close();
        } catch (SQLException e) {
            System.out.println("Problemas ao abrir a conexão com o BD");
        }
    }
}
```



## 6.5 UsuarioDAO.java

```
public Usuario consultaUsuario(int id) {  
  
    try {  
        Connection con = Banco.getConexao();  
        Usuario u = new Usuario();  
        Statement stat = con.createStatement();  
        ResultSet res = stat.executeQuery(  
            "SELECT * FROM TB_USUARIO where id = " + id);  
        if (res.next()) {  
            u.setCpf(res.getInt("ID"));  
            u.setNome(res.getString("NOME"));  
            u.setEndereco(res.getString("ENDERECO"));  
            u.setCep(res.getString("EMAIL"));  
        }  
        return u;  
    } catch (SQLException e) {  
        System.out.println("Erro = " + e.getMessage());  
        return null;  
    }  
}
```



## 6.5 UsuarioDAO.java

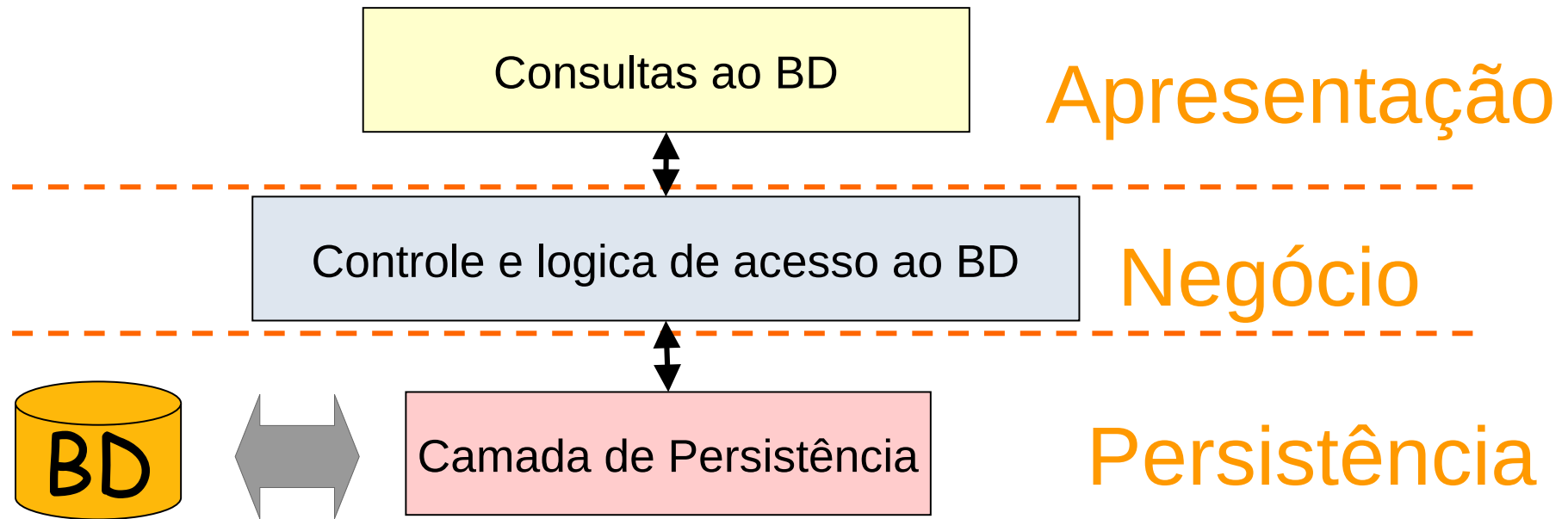
```
public Vector consultaUsuarios() {  
    Vector listaUsuarios = new Vector();  
    Usuario usu = null;  
    try {  
        Connection con = Banco.getConexao();  
        Statement stmt = con.createStatement();  
        String query = "SELECT * FROM TB_USUARIO";  
        ResultSet res = stmt.executeQuery(query);  
        while (res.next()) {  
            u = new Usuario(  
                res.getInt("ID"),  
                res.getString("NOME"),  
                res.getString("ENDERECO"),  
                res.getString("EMAIL"));  
            listaUsuarios.add(u);  
        }  
        stmt.close();  
        con.close();  
    } catch (SQLException e) { System.out.println("BD Error"); }  
}
```



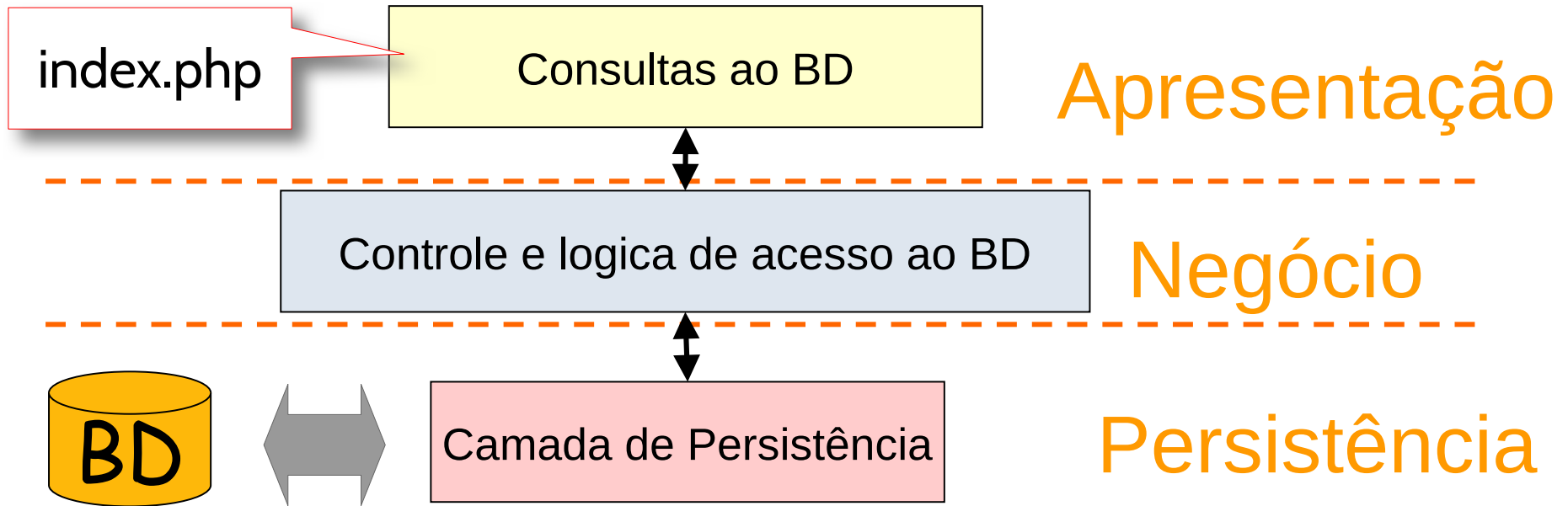
## 6.5 UsuarioDAO.java

```
public int deleteUsuario(int id) {  
    try {  
        Connection con = Banco.getConexao();  
        Statement stat; = con.createStatement();  
        stat.executeUpdate("Delete FROM Usuario WHERE id = " + id);  
        stat.close();  
        return (0);  
    } catch (SQLException e) {  
        return (1);  
    }  
}
```

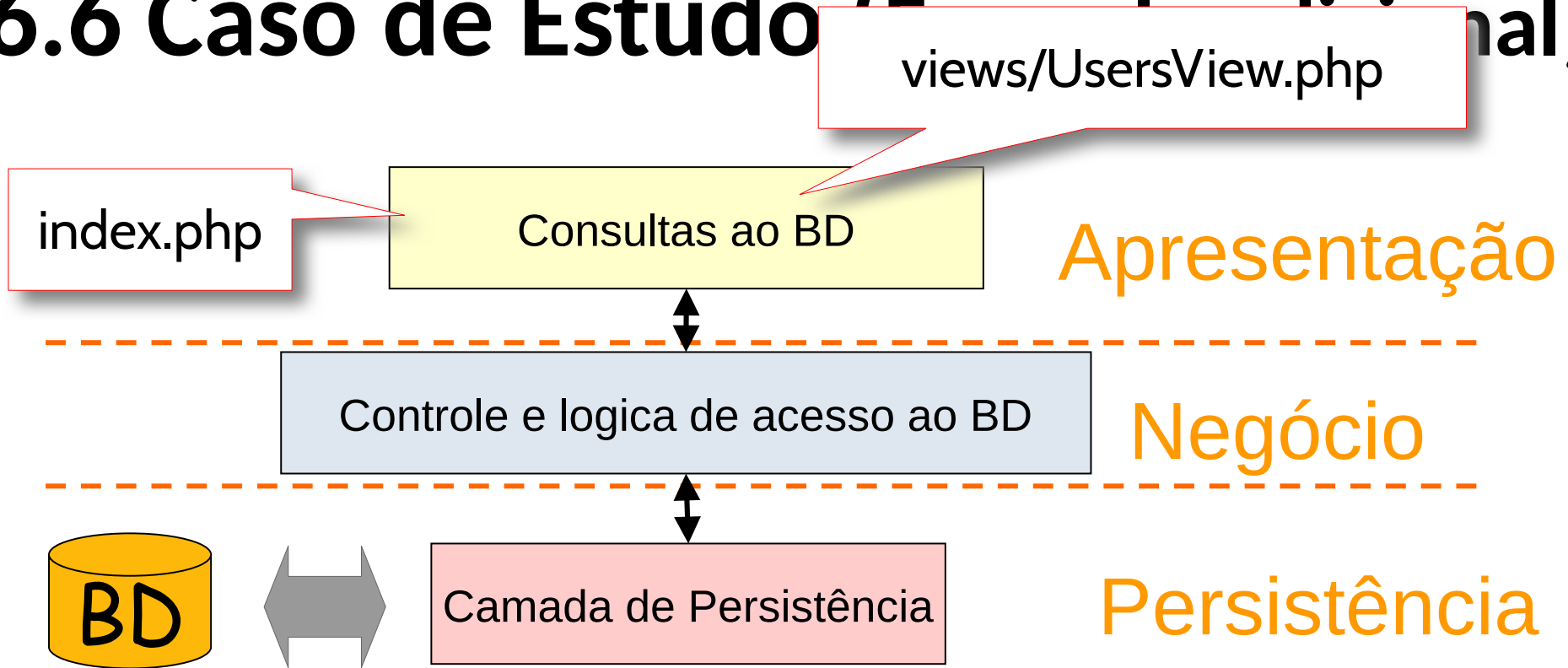
## 6.6 Caso de Estudo (Exemplo adicional)



## 6.6 Caso de Estudo (Exemplo adicional)



## 6.6 Caso de Estudo (Sistema de Gestão de Documentos)



## 6.6 Caso de Estudo (Exemplo adicional)

controllers/UserController.php

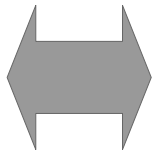
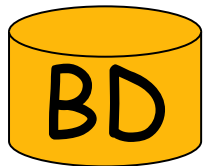
Consultas ao BD

Apresentação

Controle e logica de acesso ao BD

Negócio

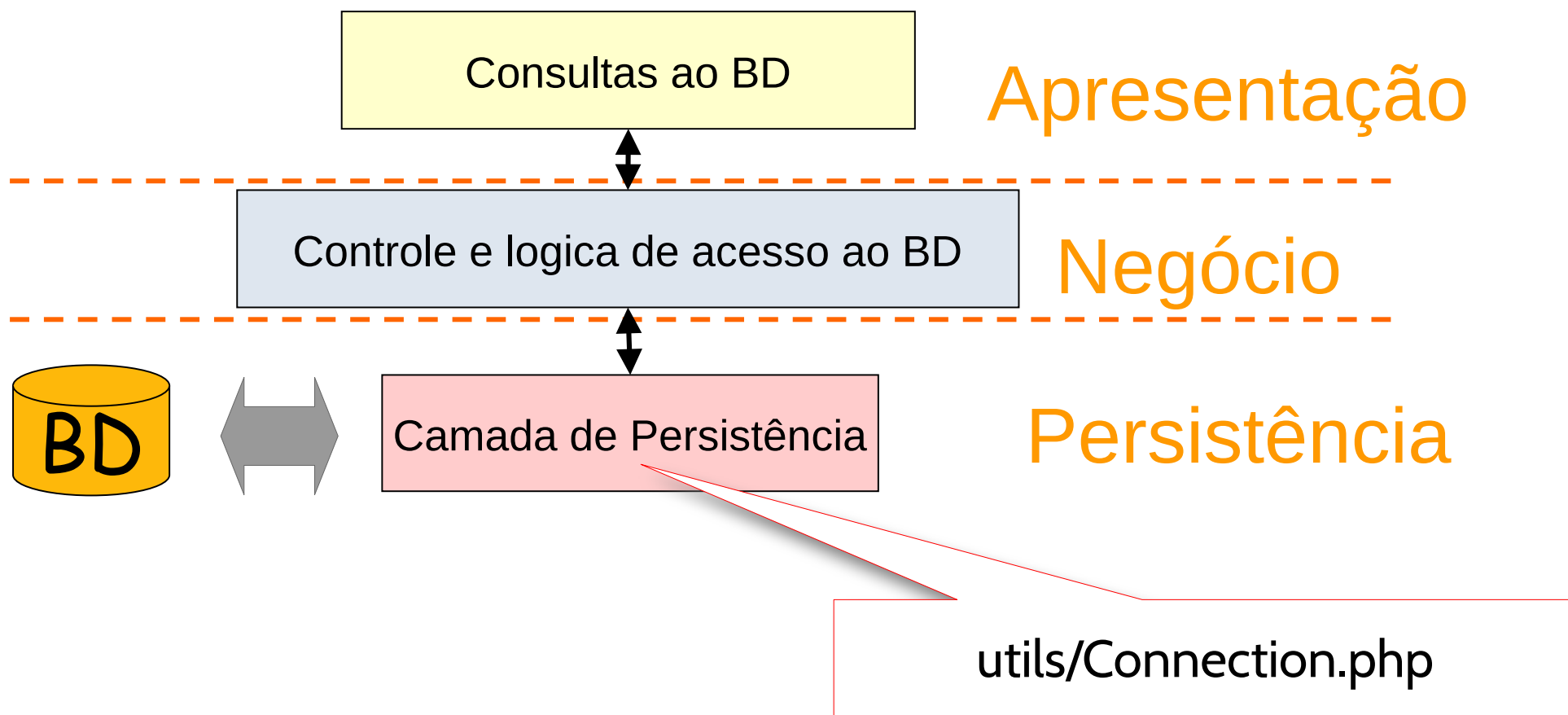
Persistência



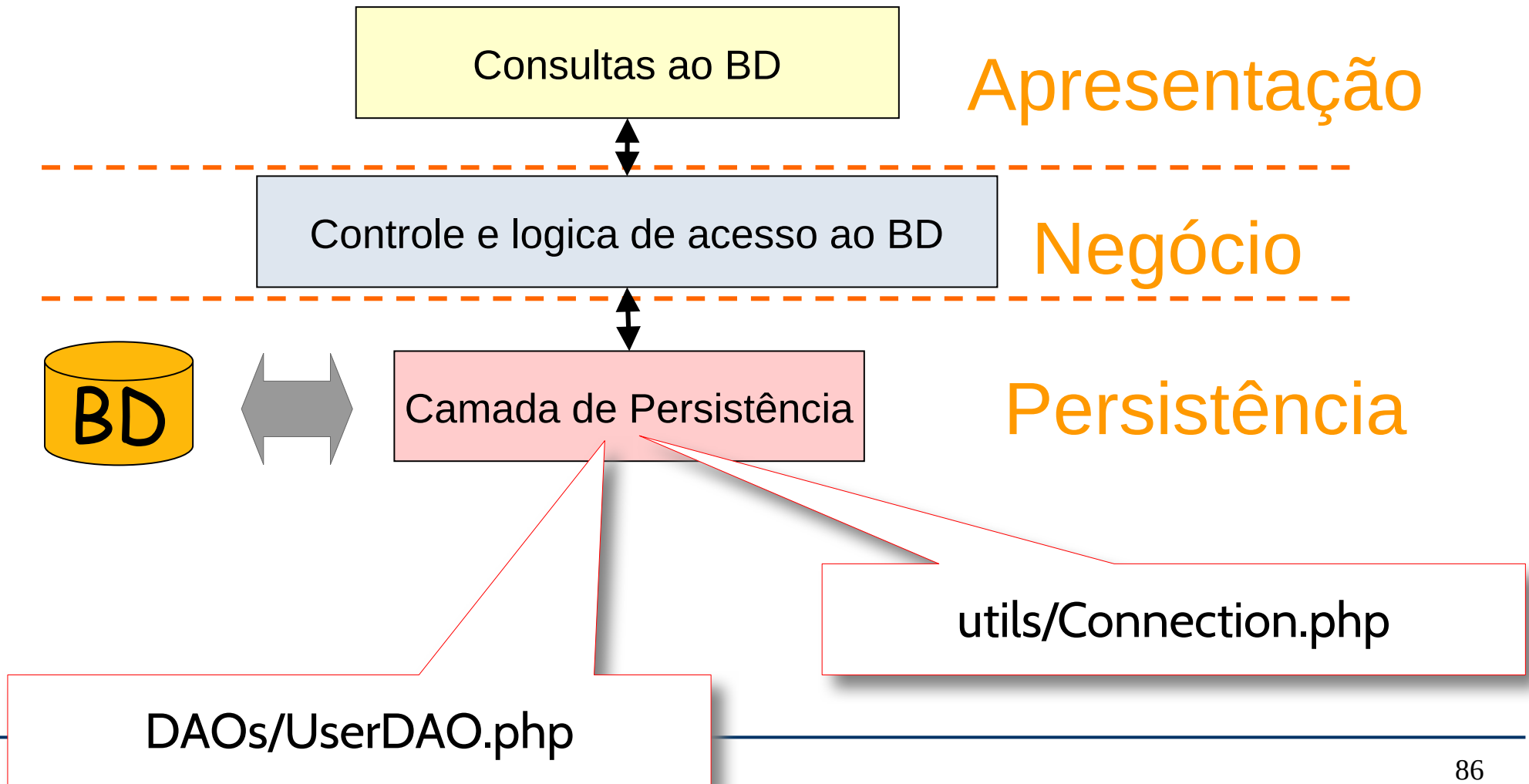
Camada de Persistência



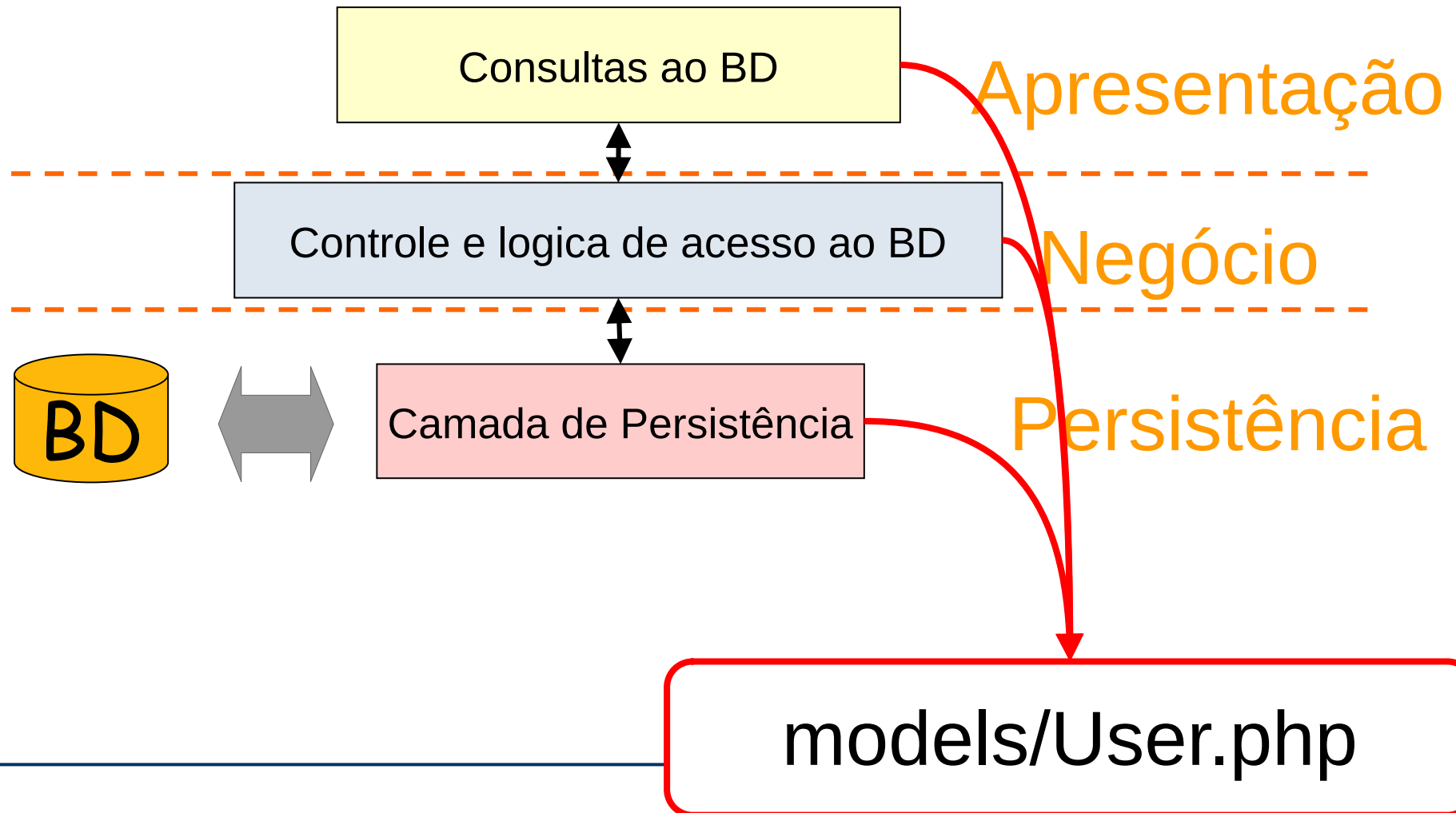
## 6.6 Caso de Estudo (Exemplo adicional)



## 6.6 Caso de Estudo (Exemplo adicional)



## 6.6 Caso de Estudo (Exemplo adicional)



[illegible]



# Dúvidas?

