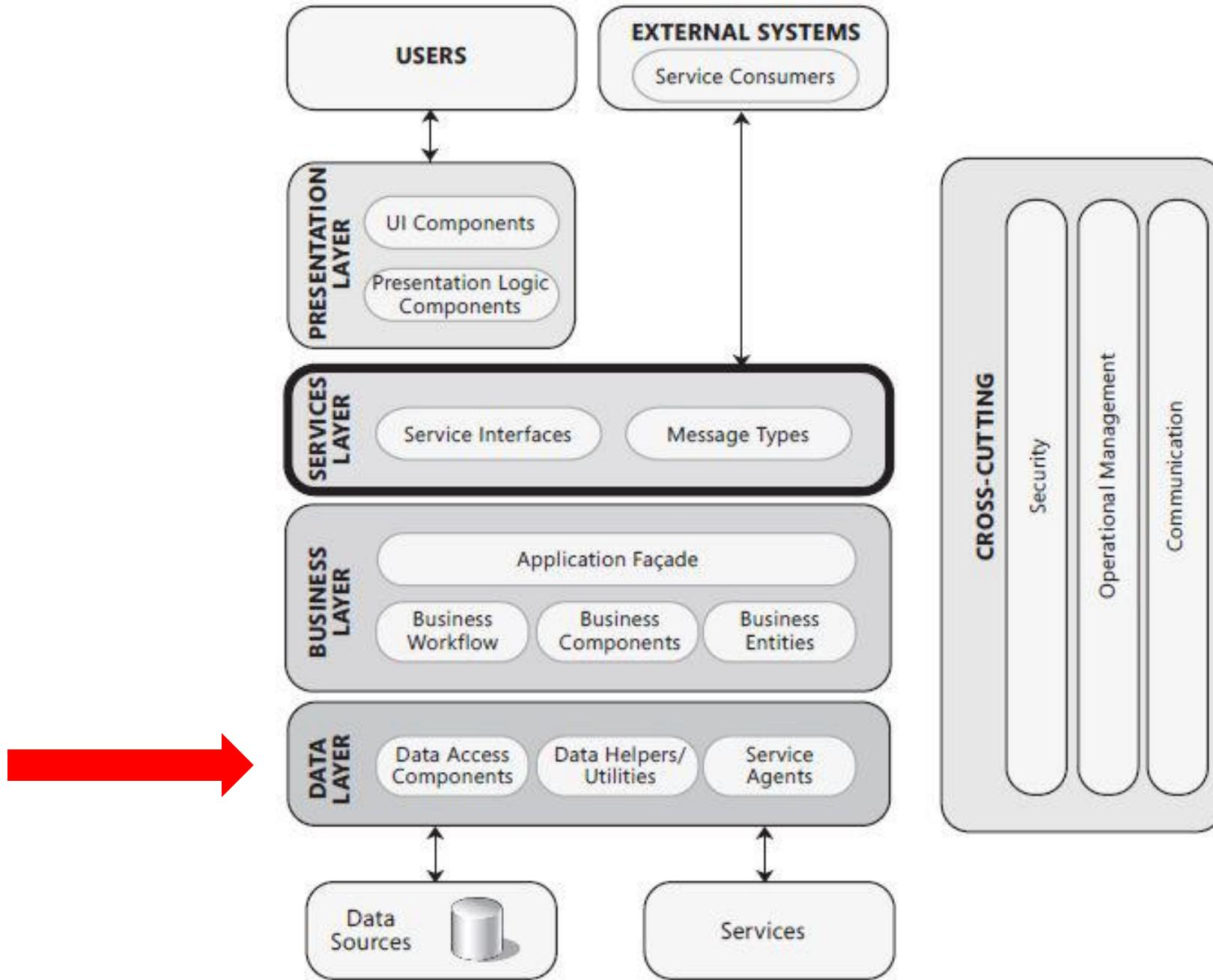
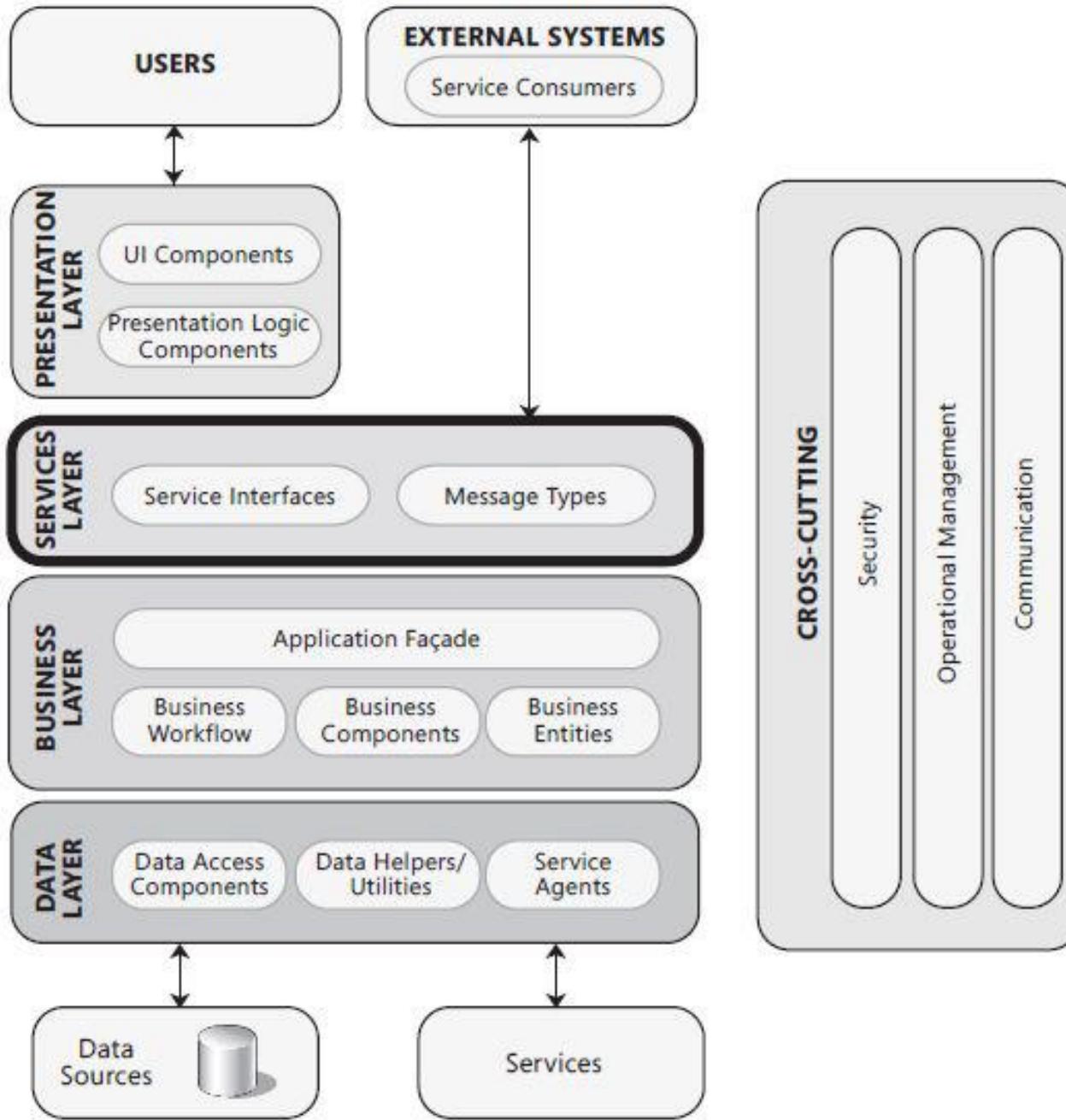


Projeto de Camada de Dados

João Pedro Oliveira Batisteli



Qual a necessidade da camada de dados?



Camada de Dados

- Para **isolar os objetos do negócio** de detalhes de comunicação com o SGBD, uma camada de dados pode ser utilizada.
- O objetivo de uma camada de dados é **isolar os objetos da aplicação de mudanças no mecanismo de armazenamento**.
 - Se um SGBD diferente tiver que ser utilizado pelo sistema , por exemplo, somente a camada de persistência é modificada.
 - Os objetos da camada de negócio permanecem intactos
- A **diminuição do acoplamento** entre os objetos e a estrutura do banco de dados torna a aplicação mais flexível e mais portável.

Camada de Dados

Modelo de Domínio:

- Representa os **conceitos e regras do mundo real** que a aplicação manipula.
- Define os **objetos de domínio** (ex.: Cliente, Pedido, Produto) e seus **estados e comportamentos**.
- Mostra como os objetos **se relacionam e interagem**.
- Captura as **regras de negócio**, garantindo que o sistema siga o comportamento esperado.

Camada de Dados

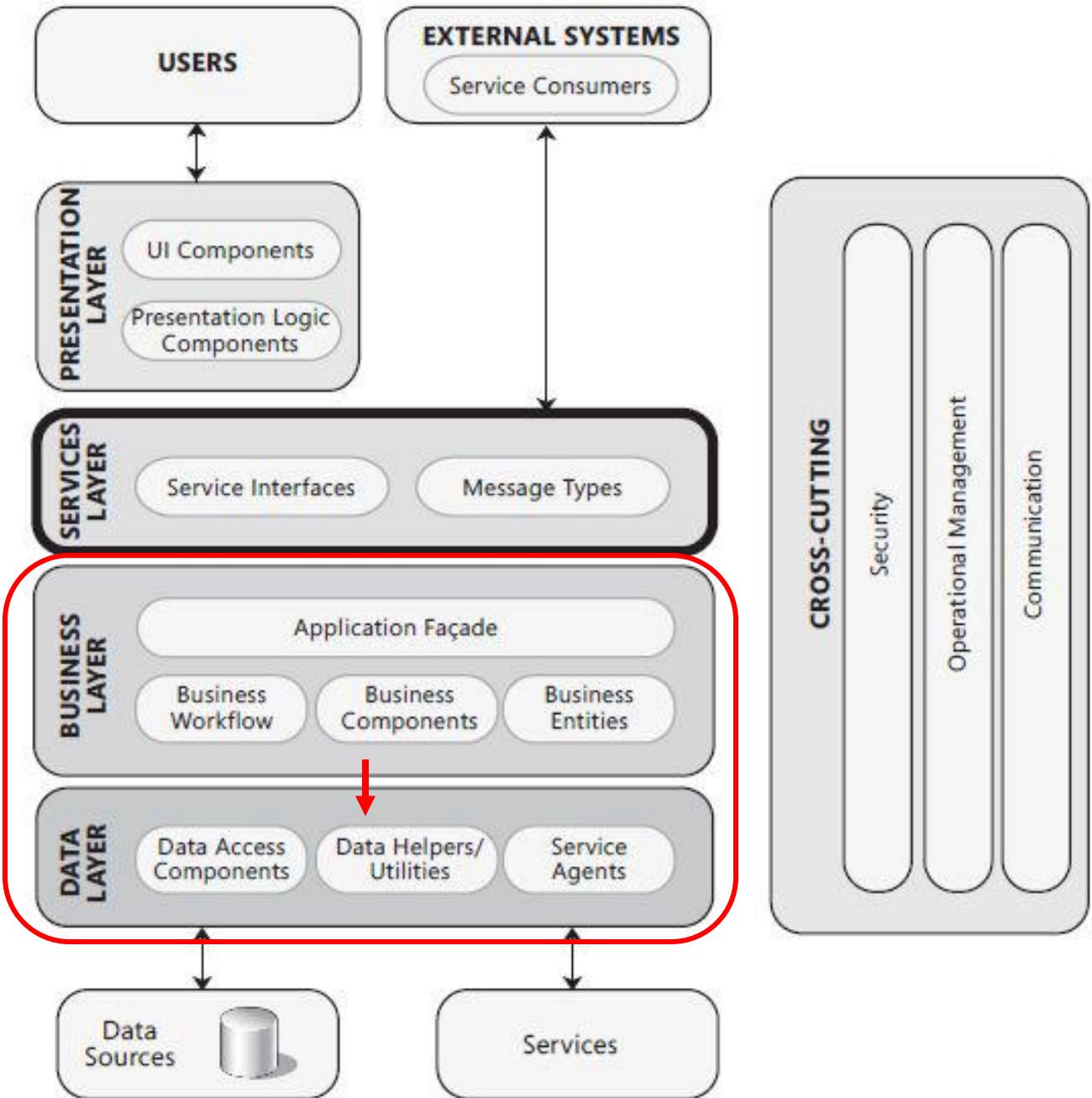
Objetos do Modelo de Domínio:

- Representam os principais **estados e comportamentos** da aplicação.
- Características comuns:
 - Podem ser acessados e utilizados por **vários usuários ao mesmo tempo**.
 - São **armazenados e recuperados** entre diferentes execuções da aplicação.
 - Sua capacidade de **existir além do ciclo de execução** da aplicação é chamada de **Persistência de Objetos**.

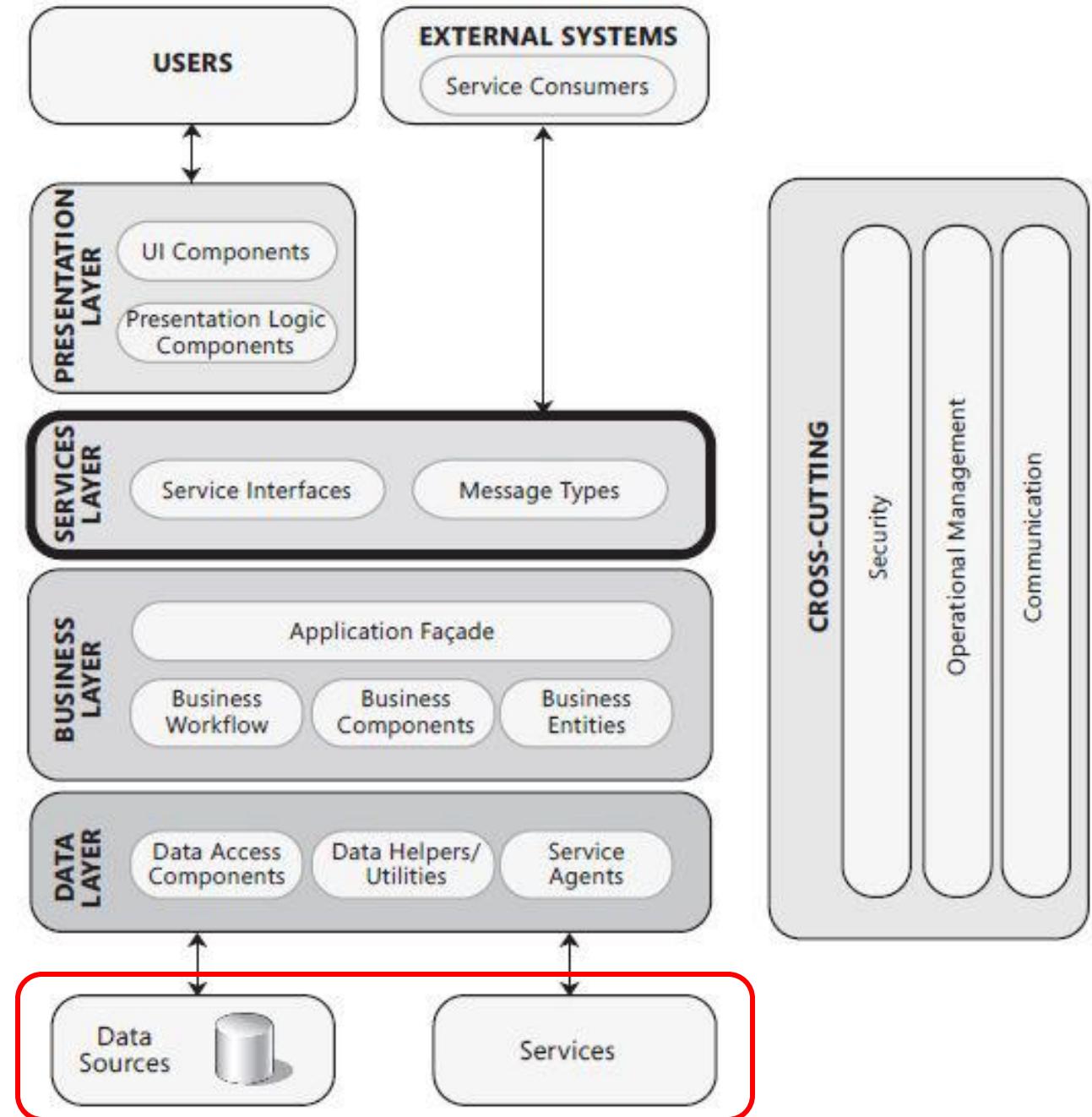
Projeto de Camada de Dados

- A Camada de dados agrupa classes que têm por finalidade **prover criação, remoção, alteração e recuperação de dados persistentes.**
- Não é o próprio mecanismo de persistência (banco de dados ou arquivo), mas um “front-end” que empacota o acesso a ele.

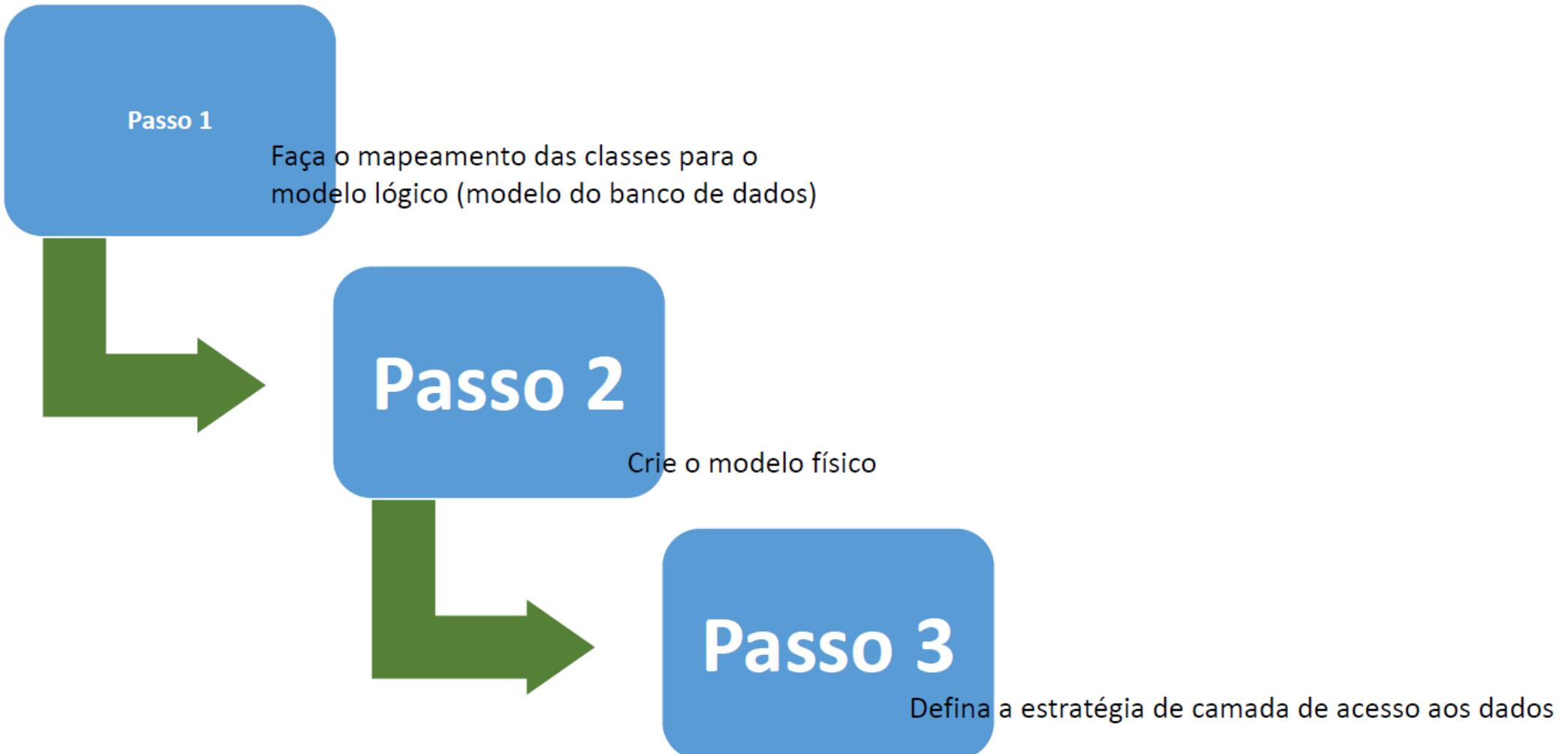
O fluxo de mensagem é da **Camada de Negócio** para a **Camada de dados**.



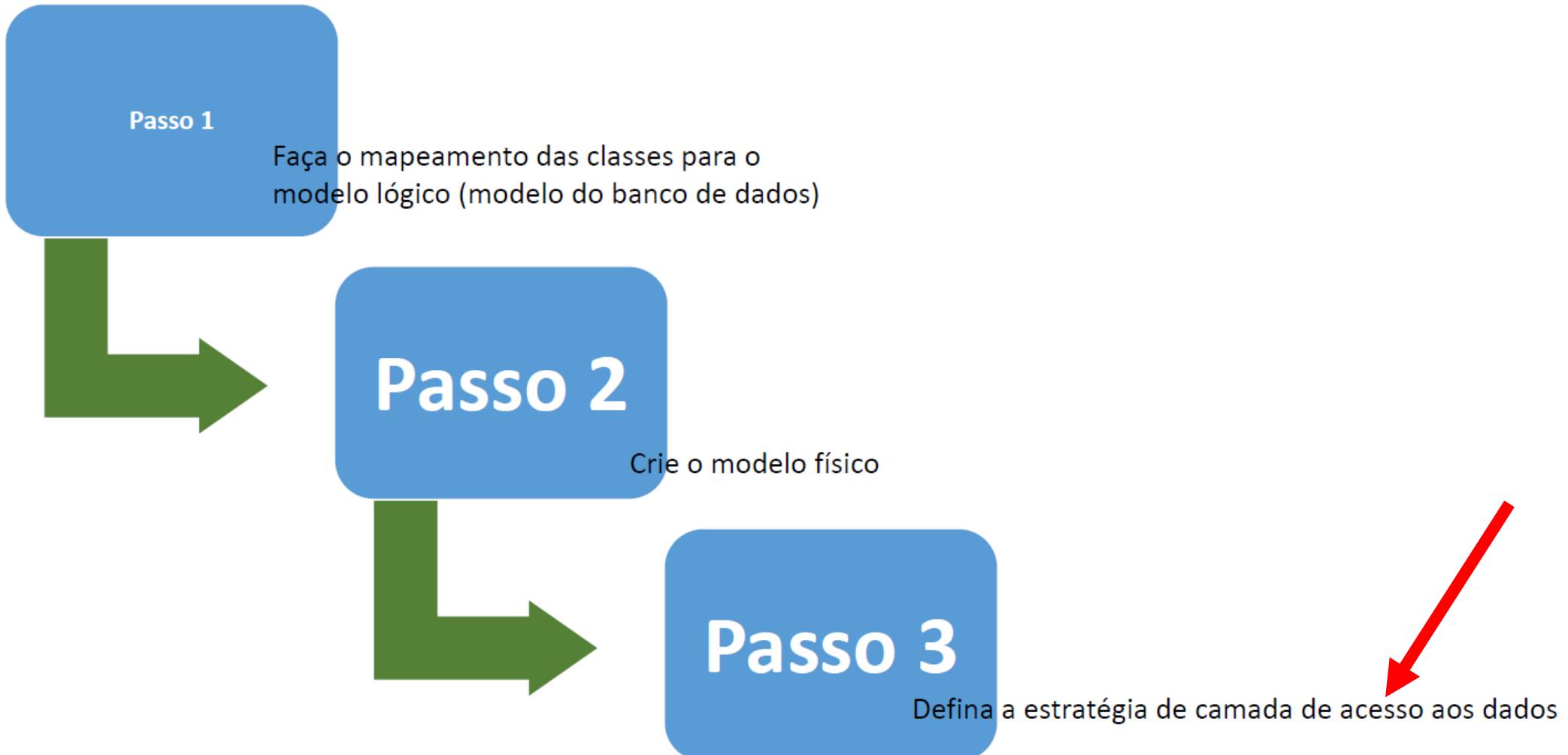
Benefício: torna possível realizar alterações na forma de persistência de dados sem impacto para o restante do sistema.



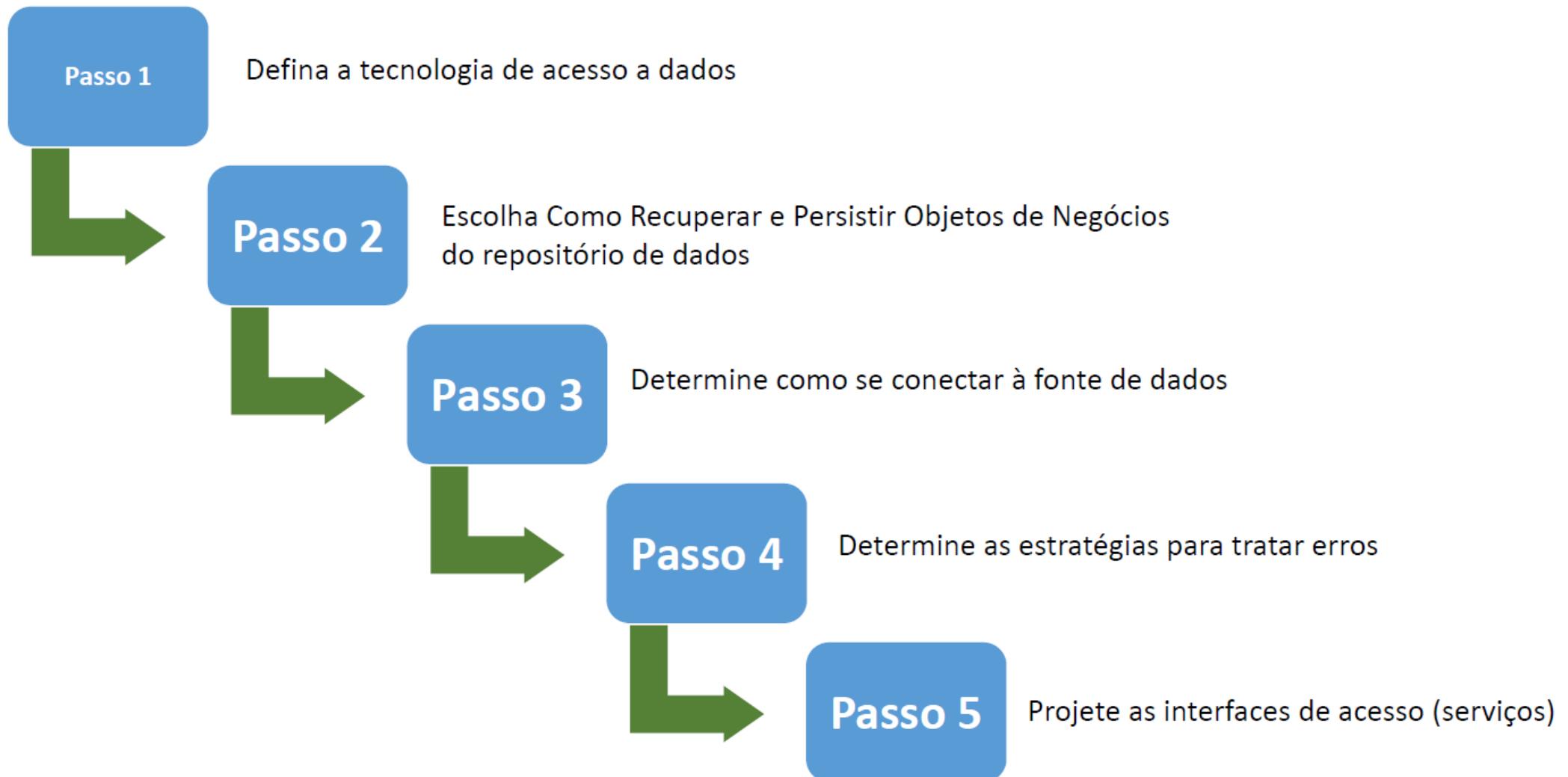
Passos para o design da camada de dados



Passos para o design da camada de dados



Passos para o design de acesso de dados



Passos para o design de acesso de dados

1 - Defina a tecnologia de acesso a dados

- Uso de um SGBDOO ou de um SGBDOR
- Acesso direto ao banco de dados
- Uso do padrão DAO (Data Access Object)
- Uso padrão Repository
- Uso de um framework ORM
- Variações ORM (Dapper)
- Uso do padrão Active Record
- Acesso NoSQL

Passos para o design de acesso de dados

1 - Defina a tecnologia de acesso a dados

- Uso de um SGBDOO ou de um SGBDOR
- Acesso direto ao banco de dados
- **Uso do padrão DAO** (Data Access Object)
- **Uso padrão Repository**
- Uso de um framework ORM
- Variações ORM (Dapper)
- Uso do padrão Active Record
- Acesso NoSQL

Padrão DAO

- **Objetivo:** **abstrair e encapsular o acesso a dados**, isolando a lógica de persistência da lógica de negócios.
- A aplicação interage com objetos DAO, sem precisar conhecer os detalhes de como os dados são armazenados (banco relacional, NoSQL, arquivo, API etc.).

Padrão DAO - exemplo

```
//Primeiramente, criamos a classe de domínio
public class Usuario{
    private Long id;
    private String nome;
    private String email;

    //getters e setters
}
```

Padrão DAO - exemplo

```
//Na sequência, criamos uma interface que provê um CRUD simples para a classe
public interface UsuarioDao{
    void criar(Usuario usuario);
    Usuario buscar(Long id);
    void atualizar(Usuario usuario);
    void deletar(Long id);
}
```

Padrão DAO - exemplo

```
//Por fim, fazemos a implementação
public class UsuarioDaoImpl implements UsuarioDao{
    private final EntityManager em;

    @Override
    public void criar(Usuario usuario){
        em.persist(usuario);
    }

    @Override
    public Usuario buscar(Long id){
        return em.find(Usuario.class,id);
    }
    //...
}
```

Padrão DAO - exemplo

```
//Por fim, fazemos a implementação
public class UsuarioDaoImpl implements UsuarioDao{
    private final EntityManager em;

    @Override
    public void criar(Usuario usuario){
        em.persist(usuario);
    }

    @Override
    public Usuario buscar(Long id){
        return em.find(Usuario.class,id);
    }
    //...
}
```

Padrão Repository

- “*Repository* é um mecanismo para encapsular o banco, recuperar e buscar comportamento que simulam uma coleção de objetos.”
- O repository também lida com dados e esconde *queries* de forma semelhante ao padrão DAO.
- Porém, o repository se trata de uma padrão de **alto nível**, ou seja, muito **mais próximo da lógica de negócio de uma aplicação**.

Padrão Repository

- Um *repository* pode usar um DAO **para pegar dados do banco e popular** um objeto de domínio.
- Ele também pode usar um DAO para **pegar informações do objeto de domínio** e mandar para o **armazenamento no banco**.

Padrão Repository - exemplo

```
//Primeiramente, criamos a classe de domínio
public class Usuario{
    private Long id;
    private String nome;
    private String email;

    //getters e setters
}
```

Padrão Repository - exemplo

```
public interface UsuarioRepository{  
    Usuario get(Long id);  
    void adicionar(Usuario usuario);  
    void atualizar(Usuario usuario);  
    void remover(Usuario usuario);  
}
```

Padrão Repository - exemplo

```
public class UsuarioRepositoryImpl implements UsuarioRepository{

    //Aqui vamos utilizar o DAO
    private UsuarioDaoImpl usuarioDaoImpl;

    @Override
    public Usuario get(Long id){
        return usuarioDaoImpl.buscar(id);
    }

    @Override
    public void adicionar(Usuario usuario){
        usuarioDaoImpl.criar(usuario);
    }

    //...
}
```

Padrão Repository - exemplo

```
public class UsuarioRepositoryImpl implements UsuarioRepository{  
  
    //Aqui vamos utilizar o DAO  
    private UsuarioDaoImpl usuarioDaoImpl;
```

MUITO SIMILAR AO PADRÃO DAO!

```
        usuarioDaoImpl.criar(usuario);  
    }  
  
    //...  
}
```

Padrão Repository - exemplo

```
public class UsuarioRedeSocial extends Usuario{  
    private List<Tweet> tweets;  
}
```

Padrão Repository - exemplo

```
public class UsuarioRepositoryImpl implements UsuarioRepository{

    //Aqui vamos utilizar o DAO
    private UsuarioDaoImpl usuarioDaoImpl;
    //Pulamos a etapa de criar o Tweet e TweetDao, mas só para acelerar a compreensão de uma classe de Domínio mais rica
    private TweetDaoImpl

    @Override
    public Usuario get(Long id){
        //não queremos mais apenas salvar o usuário, mas queremos o usuário e seus tweets
        UsuarioRedeSocial usuario = (UsuarioRedeSocial) usuarioDaoImpl.buscar(id);
        List<Tweet> tweets = tweetDaoImpl.fetchTweets(usuario.getEmail());
        usuario.setTweets(tweets);

        return usuario;
    }

    @Override
    public void adicionar(Usuario usuario){
        usuarioDaoImpl.criar(usuario);
    }

    //...
}
```

Passos no design de acesso de dados

2 - Escolha como recuperar e persistir objetos de negócios do repositório de dados

- ORM
- JSON
- XML

Passos no design de acesso de dados

3 – Determine como se conectar à fonte de dados

- **Connections**

- Definição da forma de estabelecer a comunicação entre a aplicação e a fonte de dados (ex.: string de conexão, credenciais, protocolo).

- **Connection Pooling**

- Técnica para **reutilizar conexões já abertas**, reduzindo o custo de criar/destró-las a cada requisição e melhorando o desempenho.

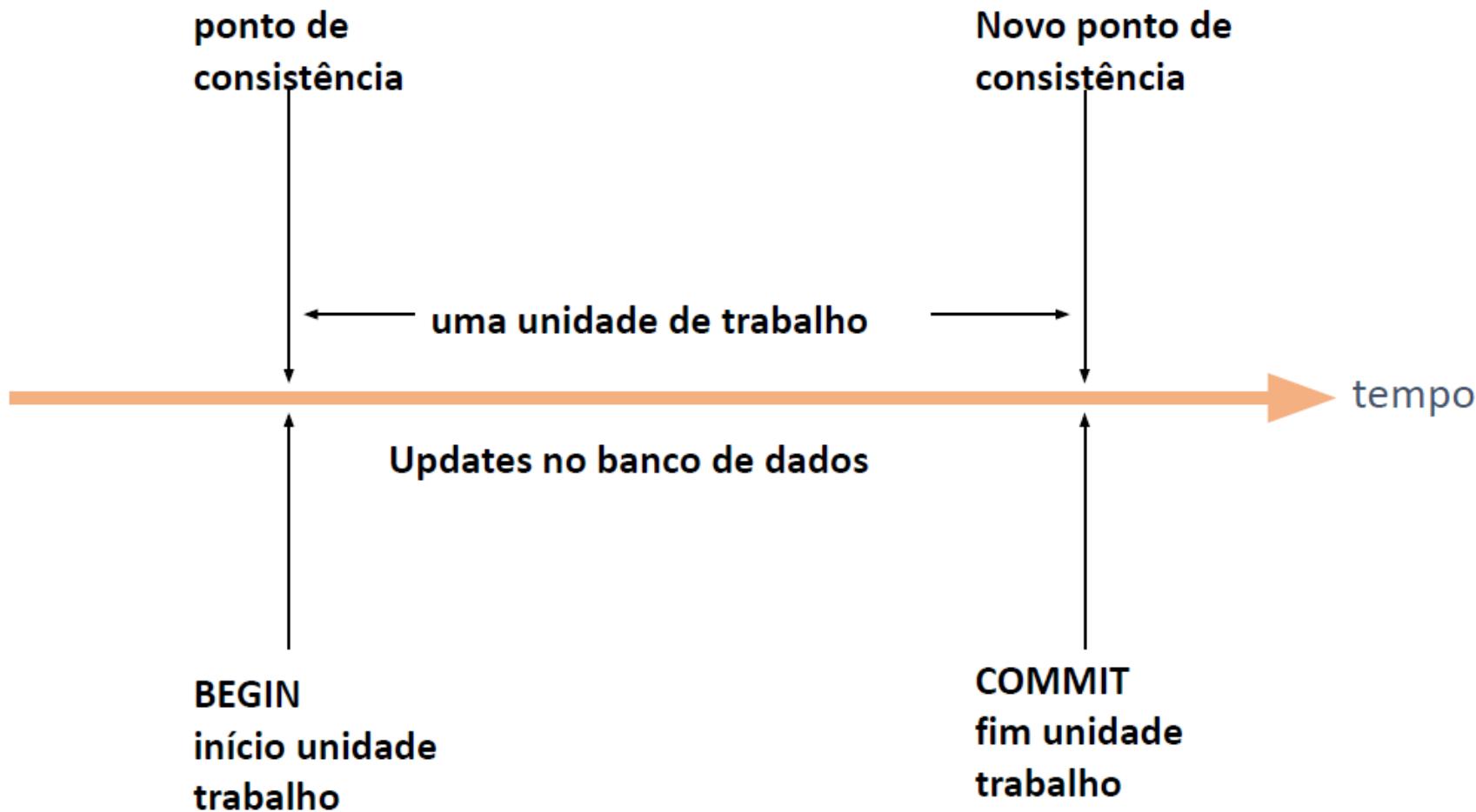
- **Transactions and Concurrency**

- Mecanismos para garantir consistência e integridade dos dados, controlando **operações atômicas** (transações) e o **acesso simultâneo de múltiplos usuários** (concorrência).

Transações no BD

- **Definição:** Uma transação é um conjunto de operações que são executadas como uma **unidade atômica de trabalho**.
- **Delimitação:**
 - Início → BEGIN TRANSACTION
 - Término → COMMIT (confirma) ou ROLLBACK (desfaz)
- **Escopo:** Inclui todas as operações realizadas entre o início e o término da transação.
- **Importância:**
 - Toda comunicação com o banco de dados ocorre dentro de uma transação.
 - Garante **consistência, integridade e recuperação** em caso de falhas.
 - Utilizada tanto para **leitura** quanto para **escrita** de dados.

Transações no BD



Controle de Concorrência em Transações

- **Objetivo:** garantir que múltiplas transações executem ao mesmo tempo sem comprometer a consistência dos dados.
- Técnicas mais comuns:
 - **Travamento em Duas Fases** (2PL – Two-Phase Locking)
 - Cada transação passa por duas etapas:
 - Crescimento: adquire bloqueios (locks).
 - Encolhimento: libera bloqueios.
 - Isso garante isolamento entre as transações.

2PL

- **O que são *Locks*?**
 - Um **lock (trava)** é como um **cadeado** colocado sobre um dado no banco.
 - Serve para **controlar quem pode acessar** (ler ou escrever) aquele dado em um determinado momento.
 - Ele evita que duas transações façam mudanças conflitantes **ao mesmo tempo**.
- **Existem dois tipos principais:**
 - **Read Lock (Compartilhado):** vários podem ler, mas ninguém pode escrever.
 - **Write Lock (Exclusivo):** só um pode ler e escrever, todos os outros ficam bloqueados.

2PL

- Toda transação tem **duas fases bem definidas**:
- **Fase de Crescimento (Growing phase)**
 - A transação pode **adquirir locks** (de leitura ou escrita).
 - **Não pode liberar** nenhum lock nessa fase.
- **Fase de Encolhimento (Shrinking phase)**
 - A transação começa a **liberar locks**.
 - A partir do momento em que libera o primeiro lock, **não pode adquirir novos**.

2PL

Tipos de Locks (Travamentos)

- **Read Lock (Compartilhado):**

- Várias transações podem **ler** o mesmo item.
- Nenhuma pode escrever enquanto o lock existir.

- **Write Lock (Exclusivo):**

- Apenas uma transação pode ler/escrever o item.
- Bloqueia o acesso de todas as outras.

- **Unlocked:**

- Item sem travas → qualquer transação pode acessá-lo.

- **Operações:**

- `read_lock(X)` → coloca trava de leitura.
- `write_lock(X)` → coloca trava de escrita.
- `unlock(X)` → libera a trava.

2PL

- Cada item bloqueado é registrado em uma **tabela de travamento**, que contém:
 - Nome do item de dados.
 - Tipo da trava (leitura ou escrita).
 - Número de leituras simultâneas (se for compartilhado).
 - Transações que detêm o bloqueio.

Passos no design de acesso de dados

4 - Determine as estratégias para tratar erros

- **Tratamento de Exceções no Acesso a Dados**
- **Princípio geral:**
 - Capture exceções.
 - Repassa para camadas superiores **apenas quando a falha comprometer** a responsividade ou a funcionalidade da aplicação.

Boas Práticas de Tratamento

- **Exceptions**
 - Identifique e trate erros esperados (ex.: conexão perdida, consulta inválida).
 - Registre (log) para auditoria e depuração.
- **Retry Logic**
 - Repetir a operação em falhas transitórias (ex.: perda momentânea de rede).
 - Definir número máximo de tentativas e intervalos.
- **Timeouts**
 - Estabeleça limites de tempo para evitar que operações travem indefinidamente.
 - Garante melhor experiência e responsividade.

Passos no design de acesso de dados

5 - Projete as interfaces de acesso (serviços)

- **Definição:** Serviços que expõem operações de acesso aos dados da aplicação.
- **Boas Práticas:**
 - Defina as interfaces de serviço de forma clara e consistente.
 - Utilize ferramentas adequadas para adicionar e gerenciar referências de serviço.
 - Especifique como e por quem o serviço será consumido:
 - Pela própria aplicação (camadas internas).
 - Por agentes ou sistemas externos (ex.: APIs).