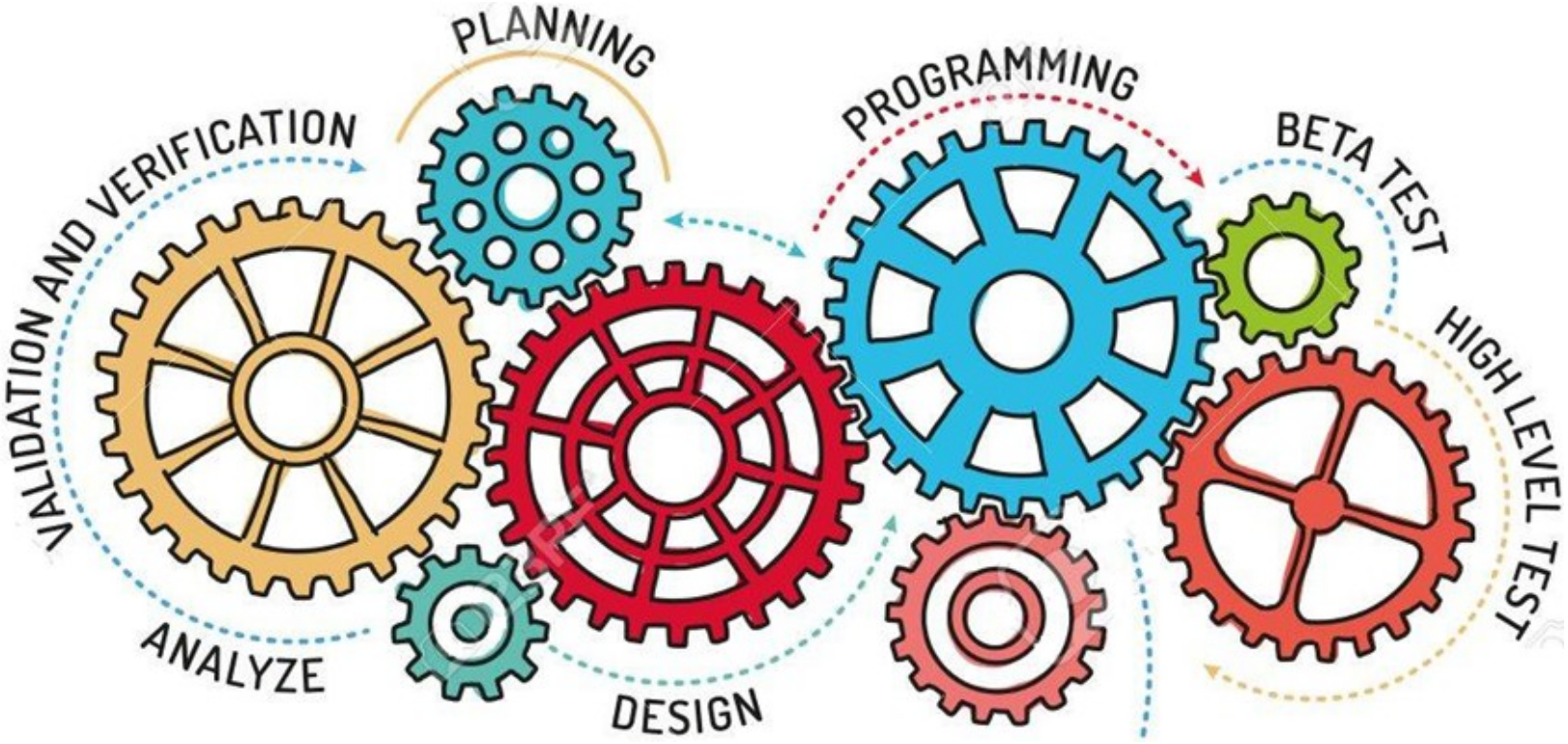
Engenharia de Software



e-mail: clenio.silva@uniube.br

Uniube

Padrão Singleton

- O padrão Singleton é um padrão de design criacional que assegura que uma classe tenha apenas uma única instância durante o ciclo de vida da aplicação:
  - fornece um ponto global de acesso a essa instância

Padrão Singleton

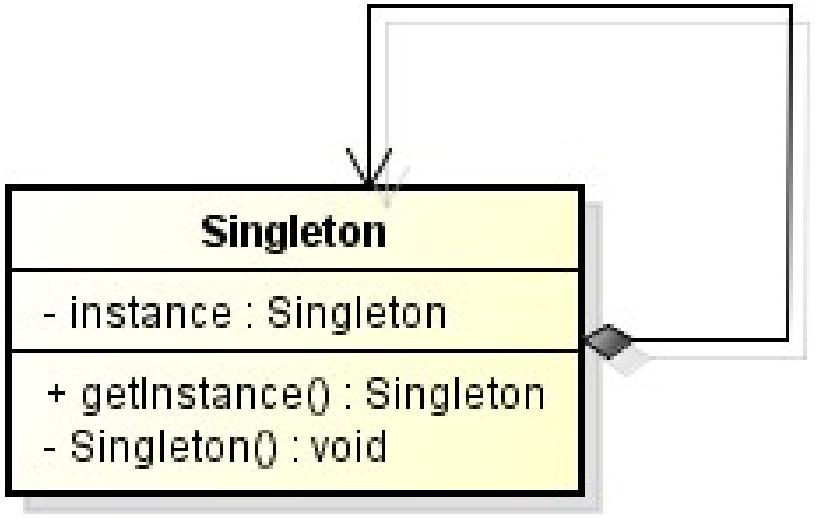
- Características do Singleton:
  - **Única instância:** Garante que apenas uma instância da classe seja criada
  - **Acesso Global:** Fornece um ponto global de acesso a essa instância

Padrão Singleton

- Como funciona a implementação do Singleton:
  - **Construtor Privado:** O construtor da classe é privado para evitar que outras classes criem instância diretamente.
  - **Instância Estática:** A classe mantém uma instância estática de si mesma.
  - **Método de Acesso Público:** Um método público e estático é fornecido para obter a instância da classe. Esse método cria a instância se ainda não existir e a retorna.

Padrão Singleton

- Exemplo Prático:



Padrão Singleton

- Exemplo Prático:

```java
public class Singleton {

    // A instância única da classe, inicializada como null
    private static Singleton instance;

    // Construtor privado para evitar instanciação externa
    private Singleton() {
        // Inicialização do Singleton
    }

    // Método público e estático para obter a instância única
    public static Singleton getInstance() {
        if (instance == null) {
            // Sincronização para garantir que apenas uma instância seja criada em ambientes multithread
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Padrão Singleton

- Exemplo Prático:
  - O construtor `private Singleton()` é privado, o que impede que a classe seja instanciada de fora.
  - private static `Singleton instance;` declara uma variável estática que mantém a única instância da classe.
  - public `static Singleton getInstance()` é um método estático que verifica se a instância já foi criada. Se não, ele a cria.

Padrão Singleton

- Exemplo Prático:
  - Outra forma é inicializar a instância no momento da definição da variável estática.
    - Isso é mais simples e evita a necessidade de sincronização

```java
public class Singleton {

    // Instância inicializada no momento da definição
    private static final Singleton instance = new Singleton();

    // Construtor privado
    private Singleton() {
        // Inicialização do Singleton
    }

    // Método público para obter a instância
    public static Singleton getInstance() {
        return instance;
    }
}
```

Padrão Singleton

- Aplicações comuns:
  - **Gerenciamento de Configurações:** Onde uma única configuração deve ser usada em toda a aplicação.
  - **Gerenciamento de Conexões:** Como em uma conexão com um banco de dados, onde apenas uma instância é desejada para otimizar recursos.
  - **Gerenciamento de Recursos:** Para objetos que devem ser únicos e compartilhados, como pools de threads ou caches.

Padrão Singleton

- Aplicações comuns: Exemplo de classe de conexão com um bando de dados

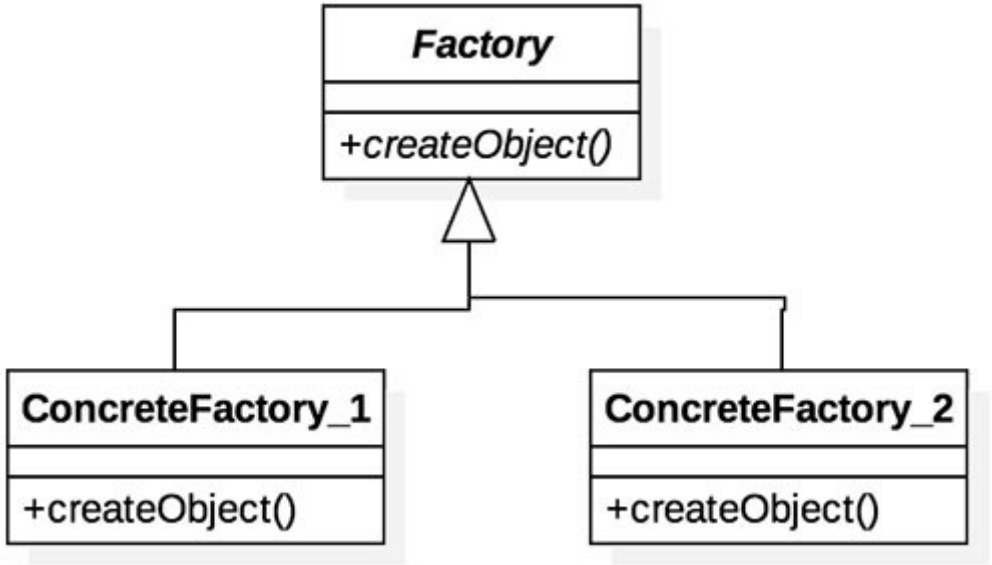
```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnection {

    // URL de conexão com o banco de dados
    private static final String URL = "jdbc:mysql://localhost:3306/aulaSingleton";
    private static final String USER = "root";
    private static final String PASSWORD = "root";

    // A instância única da classe
    private static Connection connection;

    // Construtor privado para evitar instanciação externa
    private DatabaseConnection() {
        // Evita instanciar diretamente
    }

    // Método público e estático para obter a instância única
    public static Connection getConnection() throws SQLException {
        if (connection == null || connection.isClosed()) {
            synchronized (DatabaseConnection.class) {
                if (connection == null || connection.isClosed()) {
                    try {
                        // Criar a conexão
                        connection = DriverManager.getConnection(URL, USER, PASSWORD);
                    } catch (SQLException e) {
                        throw new SQLException("Não foi possível estabelecer a conexão com o banco de dados.", e);
                    }
                }
            }
        }
        return connection;
    }

}
```

Padrão Singleton

- **Vamos práticas:**
  - **Implemente uma classe Counter que segue o padrão Singleton para garantir que apenas uma instância do contador seja criada. A classe deve permitir incrementar o contador e obter o valor atual do contador**.
    - A classe `Counter` deve seguir o padrão Singleton para garantir que apenas uma instância do contador exista.
    - A classe deve ter um método `increment()` que aumenta o valor do contador em 1.
    - A classe deve ter um método `getValue()` que retorna o valor atual do contador.
    - A classe `Counter` deve ter um construtor privado.
    - Implemente um método `getInstance()` que retorna a instância única do contador.
    - Implementar os métodos `increment()` e `getValue()`.
    - Escrever um programa de teste que use a classe **`Counter`** para incrementar e obter o valor do contador.

Padrão Abstract

- O Que é o padrão Abstract?
  - padrão Abstract é uma técnica que permite definir a estrutura de uma classe base e delegar parte da implementação para suas subclasses.



Abstract: Como funciona?

- Classe Abstrata: Define uma estrutura geral e, ás vezes, fornece uma implementação parcial para alguns métodos.
- Métodos Abstratos: Métodos que são declarados na classe abstrata, mas não têm implementação. As subclasses devem implementar esses métodos.
- Subclasses: Implementam os métodos abstratos e podem fornecer ou sobrescrever funcionalidades adicionais.

Abstract: Exemplo em Java

Vamos criar um exemplo simples para ilustrar o padrão Abstract em Java. Suponha que queremos modelar diferentes tipos de formas geométricas:

Abstract: Exemplo em Java

A classe `Forma` é abstrata e define um método abstrato `desenhar()`:

```java
// Classe abstrata
abstract class Forma {
    // Método abstrato (sem implementação)
    abstract void desenhar();

    // Método concreto
    void exibirInfo() {
        System.out.println("Esta é uma forma.");
    }
}
```

Abstract: Exemplo em Java

A classe `Circulo` implementa o método `desenhar()`:

```java
// Subclasse que estende a classe abstrata
class Circulo extends Forma {
    @Override
    void desenhar() {
        System.out.println("Desenhando um círculo.");
    }
}
```

Abstract: Exemplo em Java

A classe `Quadrado` implementa o método `desenhar()`:

```java
// Outra subclasse
class Quadrado extends Forma {
    @Override
    void desenhar() {
        System.out.println("Desenhando um quadrado.");
    }
}
```

Abstract: Exemplo em Java

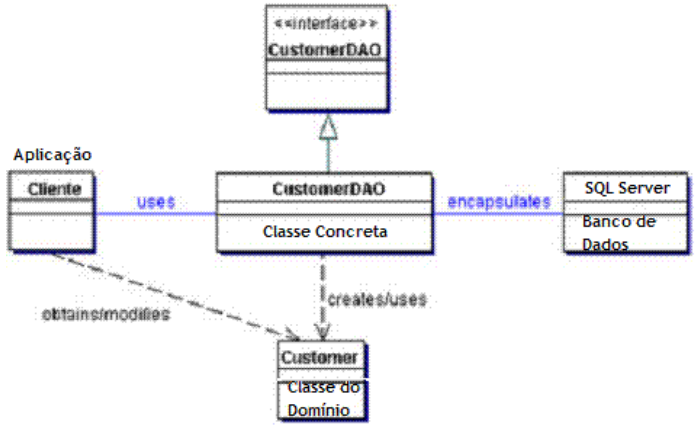
A classe Teste demonstra como criar instâncias das subclasses e usar os métodos definidos na classe abstrata:

```java
// Classe principal para testar
public class Teste {
    public static void main(String[] args) {
        Forma forma1 = new Circulo();
        Forma forma2 = new Quadrado();

        forma1.desenhar();  // Saída: Desenhando um círculo.
        forma2.desenhar();  // Saída: Desenhando um quadrado.

        forma1.exibirInfo();  // Saída: Esta é uma forma.
        forma2.exibirInfo();  // Saída: Esta é uma forma.
    }
}
```

Padrão DAO (Data Acess Object)

O padrão de desenvolvimento DAO (Data Access Object) é um padrão de design que separa a lógica de acesso a dados da lógica de negócios em uma aplicação.

- Isso é útil para manter o código mais organizado, testável e fácil de manter.
- O padrão DAO fornece uma abstração para o acesso a dados.
- Em vez de ter código que interage diretamente com a fonte de dados (como um banco de dados) misturado com a lógica de negócios, o padrão DAO define um conjunto de classes e interfaces que encapsulam as operações de acesso a dados.



DAO: Como funciona

- **Interface DAO:** Define métodos para operações básicas como criar, ler, atualizar e excluir dados.
- **Implementação DAO:** Implementa a interface DAO e contém o código real para acessar a fonte de dados.
- **Modelo (ou Entidade):** Representa a estrutura de dados que será manipulada pelo DAO.
- **Cliente:** Usa o DAO para realizar operações, sem se preocupar com os detalhes de como os dados são armazenados e recuperados.

DAO: Exemplo prático

Vamos criar um exemplo para ilustrar o padrão DAO. Suponha que estamos desenvolvendo um sistema simples para gerenciar informações de usuários.

DAO: Exemplo pratico

Primeiro, definimos uma classe Usuario que representa a estrutura de dados.

```java
public class Usuario {
    private int id;
    private String nome;
    private String email;

    public Usuario(int id, String nome, String email) {
        this.id = id;
        this.nome = nome;
        this.email = email;
    }

    // Getters e setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return "Usuario{id=" + id + ", nome='" + nome + "', email='" + email + "'}";
    }
}
```

DAO: Exemplo pratico

Definimos uma interface UsuarioDAO que declara os métodos de acesso a dados.

```java
import java.util.List;

public interface UsuarioDAO {
    void adicionarUsuario(Usuario usuario);
    Usuario buscarUsuarioPorId(int id);
    List<Usuario> listarUsuarios();
    void atualizarUsuario(Usuario usuario);
    void excluirUsuario(int id);
}
```

DAO: Exemplo pratico

Criamos uma implementação concreta da interface UsuarioDAO. Aqui, usaremos uma lista para simular o banco de dados, mas em um caso real você usaria uma conexão com um banco de dados.

```java
import java.util.ArrayList;
import java.util.List;

public class UsuarioDAOImpl implements UsuarioDAO {
    private List<Usuario> usuarios = new ArrayList<>();
    private int proximoId = 1;

    @Override
    public void adicionarUsuario(Usuario usuario) {
        usuario.setId(proximoId++);
        usuarios.add(usuario);
    }

    @Override
    public Usuario buscarUsuarioPorId(int id) {
        for (Usuario usuario : usuarios) {
            if (usuario.getId() == id) {
                return usuario;
            }
        }
        return null;
    }

    @Override
    public List<Usuario> listarUsuarios() {
        return usuarios;
    }

    @Override
    public void atualizarUsuario(Usuario usuario) {
        for (int i = 0; i < usuarios.size(); i++) {
            if (usuarios.get(i).getId() == usuario.getId()) {
                usuarios.set(i, usuario);
                return;
            }
        }
    }

    @Override
    public void excluirUsuario(int id) {
        usuarios.removeIf(usuario -> usuario.getId() == id);
    }
}
```

DAO: Exemplo pratico

O cliente usa o DAO para realizar operações sem precisar se preocupar com os detalhes do acesso aos dados.

```java
public class Main {
    public static void main(String[] args) {
        UsuarioDAO usuarioDAO = new UsuarioDAOImpl();

        // Adicionar usuários
        usuarioDAO.adicionarUsuario(new Usuario(0, "Alice", "alice@example.com"));
        usuarioDAO.adicionarUsuario(new Usuario(0, "Bob", "bob@example.com"));

        // Listar usuários
        System.out.println("Lista de usuários:");
        for (Usuario usuario : usuarioDAO.listarUsuarios()) {
            System.out.println(usuario);
        }

        // Buscar um usuário
        Usuario usuario = usuarioDAO.buscarUsuarioPorId(1);
        System.out.println("Usuário com ID 1: " + usuario);

        // Atualizar um usuário
        usuario.setNome("Alice Cooper");
        usuarioDAO.atualizarUsuario(usuario);

        // Listar usuários após atualização
        System.out.println("Lista de usuários após atualização:");
        for (Usuario u : usuarioDAO.listarUsuarios()) {
            System.out.println(u);
        }

        // Excluir um usuário
        usuarioDAO.excluirUsuario(2);

        // Listar usuários após exclusão
        System.out.println("Lista de usuários após exclusão:");
        for (Usuario u : usuarioDAO.listarUsuarios()) {
            System.out.println(u);
        }
    }
}
```

Abstract: Exercício

1. Crie uma classe abstrata chamada **Veiculo** com:
   I. Um método abstrato **exibir_info()** para exibir informações especificas do veículo.
   II. Um método abstrato **calcular_custo()** que calcula o custo de operação do veículo.
2. Crie duas subclasses que estendem a classe Veiculo:
   I. Carro: Representa um carro. Adicione um atributo para o tipo de combustível (por exemplo, “gasolina”, “dísel”).
   II. Bicicleta: Representa uma bicicleta. Não precisa de atributos adicionais.
3. Implemente o método **exibir_info()** para mostrar uma mensagem que inclua o tipo de veículo e, para o carro, o tipo de combustível
4. Implemente o método **calcular_custo()** para:
   I. Carro: Retorne um custo fictício baseado no tipo de combustível (por exemplo, gasolina = 0,2, dísel = 0,3).
   II. Bicicleta: Retorne um custo fixo de operação, por exemplo, 0,0.
5. Implemente uma classe **Teste** para testar as classes.

DAO: Exercício

I. Você deve criar um sistema simples para gerenciar livros em uma biblioteca utilizando o padrão DAO. O sistema deve permitir as seguintes operações:

1. Adicionar um novo livro: Inserir um novo livro na lista de dados ou, caso queira usar um banco, uma base de dados.
2. Buscar um livro por ID: Recuperar informações de um livro específico a partir de seu ID.
3. Listar todos os livros: Recuperar e listar todos os livros disponíveis.
4. Atualizar informações do livro: Alterar os dados de um livro existente.
5. Excluir um livro: Remover um livro a partir de seu ID.

II. Defina a Entidade **Livro**: Crie uma classe que represente um livro.

III. Crie a Interface **LivroDAO**: Defina os métodos para as operações CRUD.

IV. Implemente **LivroDAOImpl**: Forneça a implementação concreta dos métodos DAO usando uma lista (Caso queira, pode usar o JDBC).

V. Crie uma Classe de Teste: Utilize a implementação DAO para testar as operações CRUD.

Padrão Service (Service Layer)

- É uma abordagem arquitetural que ajuda a organizar a lógica de negócios em aplicações, separando as camadas de apresentação e de acesso a dados.
  - Beneficios:
    - Manutenção
    - Código limpo
    - Testabilidade
    - reutilização

Padrão Service (Service Layer)

- Principais características do padrão service:
  - Encapsulamento da Lógica de Negocio:
    - A camada de serviço contém as regras de negócios, decisões e lógica de manipulação de dados, garantindo que essa lógica não fique espalhada pela aplicação.
  - Interação com Várias Fontes de Dados:
    - A camada de serviço pode interagir com diferentes fontes de dados (bancos de dados, APIs externas, etc.) e encapsular essa complexidade, permitindo que a camada de apresentação trabalhe apenas com objetos de domínio.

Padrão Service (Service Layer)

- Principais características do padrão service:
  - Facilidade de Testes:
    - Como a lógica de negócios está centralizada, fica mais fácil escrever testes unitários e de integração para validar essa lógica.
  - Reutilização:
    - Os serviços podem ser reutilizados em diferentes partes da aplicação ou até em diferentes aplicações, reduzindo a duplicação de código.

Padrão Service (Service Layer)

- Principais características do padrão service:
  - Independência da Camada de Apresentação:
    - A camada de apresentação (como uma API REST ou uma interface de usuário) pode evoluir de forma independente da lógica de negócios, desde que siga as interfaces definidas na camada de serviço.

Padrão Service (Service Layer)

- Estruturando as camadas:
  1. Camada de Apresentação:
     a) Responsável pela interação com o usuário, pode ser uma interface gráfica ou uma API.
  2. Camada de Serviço:
     a) Contém serviços que encapsulam a lógica de negócios. Exemplo: **UserService**, **ProductService**.
  3. Camada de Acesso a Dados:
     a) Gerencia a persistência de dados, utilizando repositórios ou DAO (Data Access Object). Exemplo: **UserRepository**, **ProductRepository**.
  4. Modelo de Domínio:
     a) Representa os objetos que são utilizados na lógica de negócios. Exemplo: User, Product.

Padrão Service - Implementação

Modelo de Domínio:

```java
public class User {  no usages
    private Integer id;  3 usages
    private String name;  3 usages


    public User(Integer id, String nome){  no usages
        this.id = id;
        this.name = nome;
    }

    public Integer getId() {  no usages
        return id;
    }

    public void setId(Integer id) {  no usages
        this.id = id;
    }

    public String getName() {  no usages
        return name;
    }

    public void setName(String name) {  no usages
        this.name = name;
    }
}
```

Padrão Service - Implementação

Repositório de Usuários:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class UserRepository {  4 usages
    private List<User> users;  3 usages


    public UserRepository(){  1 usage
        this.users = new ArrayList<>();

    }

    public User getUserById(Integer id){  1 usage

        for(User user: users){
            if(Objects.equals(user.getId(), id)){
                return user;
            }
        }
        return null;
    }


    public void addUser(User user){  1 usage
        users.add(user);
    }

}
```

Padrão Service - Implementação

Camada de Serviço:

```java
public class UserService {  4 usages
    private UserRepository userRepository;  3 usages

    public UserService(UserRepository userRepository){  1 usage
        this.userRepository = userRepository;
    }

    public User getUser(Integer id){  1 usage
        User user = userRepository.getUserById(id);
        if(user == null){
            throw new RuntimeException("Usuário não encontrado");
        }
        return user;
    }

    public void addUser(User user) {  1 usage
        userRepository.addUser(user);
    }
}
```

Padrão Service - Implementação

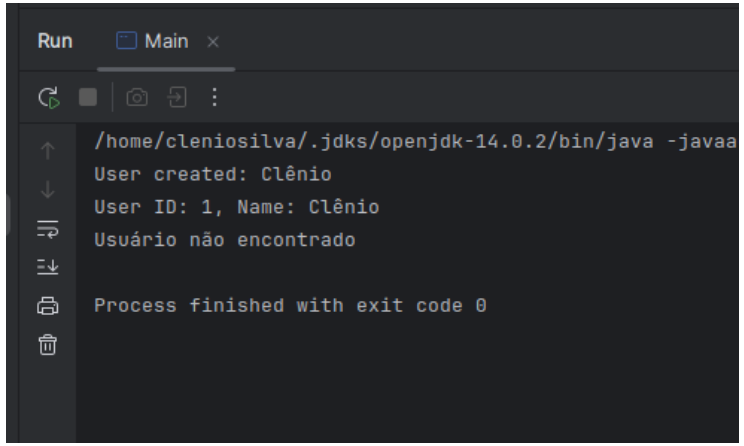
Camada de Apresentação (Controller):

```
public class UserController {  2 usages
    private UserService userService;  3 usages

    public UserController(UserService userService) {  1 usage
        this.userService = userService;
    }

    public void displayUser(Integer id) {  2 usages
        try {
            User user = userService.getUser(id);
            System.out.println("User ID: " + user.getId() + ", Name: " + user.getName());
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
        }
    }


    public void createUser(Integer id, String name) {  1 usage
        User newUser = new User(id, name);
        userService.addUser(newUser);
        System.out.println("User created: " + newUser.getName());
    }
}
```

Padrão Service - Implementação

Classe Principal (Main):

```java
public class Main {
    public static void main(String[] args) {
        UserRepository userRepository = new UserRepository();
        UserService userService = new UserService(userRepository);
        UserController userController = new UserController(userService);

        // Adicionando um usuário
        userController.createUser( id: 1,  name: "Clênio");

        // Exibindo o usuário
        userController.displayUser( id: 1);

        // Tentando exibir um usuário inexistente
        userController.displayUser( id: 2);

    }
}
```



Padrão SOLID

- O padrão SOLID é um conjunto de cinco princípios de design de software que visam melhorar a legibilidade, manutenção e escalabilidade do código.
- SOLID é um acrônimo que representa cinco princípios distintos:
  - 1. **S** – Single Responsibility: Princípio da Responsabilidade Única
  - 2. **O** – Open/Closed Principle: Princípio do Aberto/Fechado
  - 3. **L** – Liskov Substiution Principle: Princípio da Substituição de Liskov
  - 4. **I** – Interface Segregation Principle: Princípio da Segregação de Interface
  - 5. **D** – Dependency Inversion Principle: Princípio da Inversão de Dependência

Padrão SOLID

S - Single Responsibility Principle (SRP): Princípio da Responsabilidade Única

- Cada classe deve ter uma única responsabilidade, ou seja, deve ter apenas uma razão para mudar.

```java
// Classe que viola o SRP
public class Funcionario {
    private String nome;
    private double salario;

    public void calcularSalario() {
        // Lógica para calcular o salário
    }

    public void salvar() {
        // Lógica para salvar o funcionário no banco
    }
}
```

```java
// Classe que segue o SRP
public class Funcionario {
    private String nome;
    private double salario;

    public double calcularSalario() {
        // Lógica para calcular o salário
        return salario;
    }
}

public class FuncionarioRepositorio {
    public void salvar(Funcionario funcionario) {
        // Lógica para salvar o funcionário no banco
    }
}
```

Padrão SOLID

O - Open/Closed Principle (OCP): Princípio do Aberto/Fechado

- As classes devem ser abertas para extensão, mas fechadas para modificação.

```java
// Classe violadora do OCP
public class CalculadoraDeImpostos {
    public double calcularImposto(Produto produto) {
        if (produto.getCategoria().equals("eletronico")) {
            return produto.getPreco() * 0.2;
        } else {
            return produto.getPreco() * 0.1;
        }
    }
}
```

```java
// Classe que segue o OCP
public abstract class Imposto {
    public abstract double calcular(Produto produto);
}

public class ImpostoEletronico extends Imposto {
    public double calcular(Produto produto) {
        return produto.getPreco() * 0.2;
    }
}

public class ImpostoGenerico extends Imposto {
    public double calcular(Produto produto) {
        return produto.getPreco() * 0.1;
    }
}
```

Padrão SOLID

L - Liskov Substitution Principle (LSP): Princípio da Substituição de Liskov

- Subtipos devem ser substituíveis por seus tipos base, sem alterar as propriedades do programa.

```java
// Classe violadora do LSP
public class Carro {
    public void ligar() {
        System.out.println("Carro ligado");
    }
}

public class CarroEletrico extends Carro {
    @Override
    public void ligar() {
        System.out.println("Carro elétrico ligado");
    }
}

public class CarroComMotorV8 extends Carro {
    @Override
    public void ligar() {
        System.out.println("Carro com motor V8 ligado");
    }
}
```

```java
// Classe que segue o LSP
public abstract class Veiculo {
    public abstract void ligar();
}

public class CarroEletrico extends Veiculo {
    @Override
    public void ligar() {
        System.out.println("Carro elétrico ligado");
    }
}

public class CarroComMotorV8 extends Veiculo {
    @Override
    public void ligar() {
        System.out.println("Carro com motor V8 ligado");
    }
}
```

Padrão SOLID

I - Interface Segregation Principle (ISP): Princípio da Segregação de Interface

- Uma classe não deve ser forçada a implementar interfaces que ela não utiliza.

```java
// Classe que viola o ISP
public interface Animal {
    void comer();
    void voar();
}

public class Pato implements Animal {
    public void comer() {
        System.out.println("Pato comendo");
    }

    public void voar() {
        System.out.println("Pato voando");
    }
}

public class Cavalo implements Animal {
    public void comer() {
        System.out.println("Cavalo comendo");
    }

    public void voar() {
        // Cavalo não pode voar, mas precisa implementar o método
    }
}
```

```java
// Classe que segue o ISP
public interface Animal {
    void comer();
}

public interface Ave extends Animal {
    void voar();
}

public class Pato implements Ave {
    public void comer() {
        System.out.println("Pato comendo");
    }

    public void voar() {
        System.out.println("Pato voando");
    }
}

public class Cavalo implements Animal {
    public void comer() {
        System.out.println("Cavalo comendo");
    }
}
```

Padrão SOLID

D - Dependency Inversion Principle (DIP): Princípio da Inversão de Dependência.

- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

```java
// Classe que viola o DIP
public class Repositorio {
    public void salvar(Usuario usuario) {
        System.out.println("Usuário salvo no banco de dados");
    }
}

public class UsuarioService {
    private Repositorio repositorio;

    public UsuarioService() {
        this.repositorio = new Repositorio();
    }

    public void salvarUsuario(Usuario usuario) {
        repositorio.salvar(usuario);
    }
}
```

```java
// Classe que segue o DIP
public interface Repositorio {
    void salvar(Usuario usuario);
}

public class RepositorioMySQL implements Repositorio {
    public void salvar(Usuario usuario) {
        System.out.println("Usuário salvo no MySQL");
    }
}

public class RepositorioPostgres implements Repositorio {
    public void salvar(Usuario usuario) {
        System.out.println("Usuário salvo no PostgreSQL");
    }
}

public class UsuarioService {
    private Repositorio repositorio;

    public UsuarioService(Repositorio repositorio) {
        this.repositorio = repositorio;
    }

    public void salvarUsuario(Usuario usuario) {
        repositorio.salvar(usuario);
    }
}
```