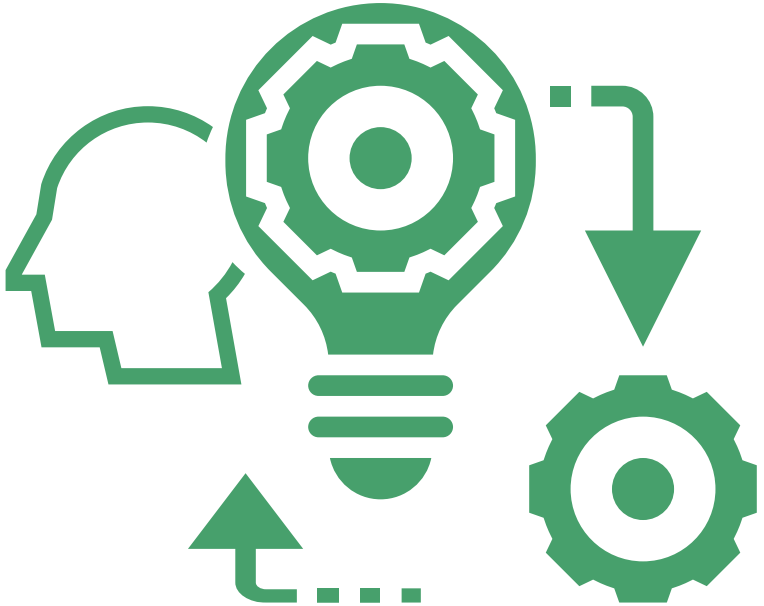
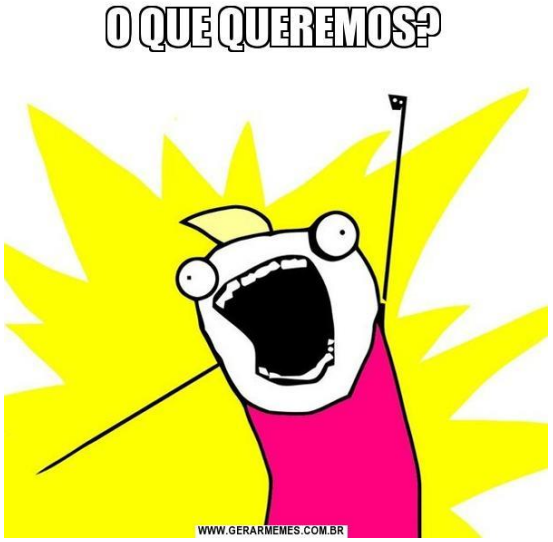


Paradigmas de Programação

Aula 04 – Como vai ser ???

- Apresentação dos trabalhos
  - Grupo 04 → 05 → 06
- Princípios, padrões e boas práticas
  - High Cohesion & Low Coupling
  - KISS, YAGNI e DRY
  - DDD
  - Lei de Demeter
- Entregáveis Aula 03
- Microservices
- Entregáveis Aula 04


O QUE QUEREMOS?
WWW.GERARMEMES.COM.BR


LOOSE COUPLING
HIGH COHESION
makeameme.org

O que é Alta Coesão na Programação?

- É um princípio de design de software que se refere ao grau em que os elementos dentro de um **módulo, classe, método ou função** estão relacionados e trabalham juntos para cumprir uma única responsabilidade ou propósito.

- Em outras palavras, um módulo, classe, método ou função com alta coesão realiza um conjunto de tarefas intimamente relacionadas e bem definidas.

O que é **Alta Coesão** na Programação?

Mas esse conceito só vale para Programação Orientada a Objetos ???

se refere ao grau em que os
mé
pri

- Em outras palavras, um módulo, classe, méto
coesão realiza um conjunto de tarefas intima
bem definidas.

O q ção?

- É um o grau em que os elem u função estão relac única resp

- Em c unção com alta coesão re de software que se refere ao grau em bem defin m **módulo, classe, método ou função** e m juntos para cumprir uma única

Características de Alta Coesão

- **Foco e Clareza:** Cada módulo, classe, método ou função tem funcionalidades específicas e bem delimitadas, facilitando a compreensão do código.
- **Reutilização:** Classes ou módulos com alta coesão são mais fáceis de reutilizar em outros contextos, pois eles encapsulam funcionalidade específica de forma isolada.
- **Facilidade de Refatoração:** As responsabilidades estão bem organizadas e os impactos das mudanças são previsíveis.
- **Testabilidade:** Suas responsabilidades são bem definidas e limitadas, o que permite a criação de testes unitários claros e focados.

E o Baixo Acoplamento ???

- É um princípio de design de software que se refere à ideia de que os diferentes módulos, classes, métodos ou funções, ou seja, os **componentes de um sistema**, devem ter o menor número possível de dependências uns dos outros.
- Em outras palavras, componentes com baixo acoplamento funcionam de maneira mais independente, minimizando o impacto que mudanças em uma parte do sistema podem ter sobre outras partes.

E o Baixo Acoplamento ???

Mas esse conceito só vale para Programação Orientada a Objetos ???

- Em outras palavras, componentes com baixo
de maneira mais independente, minimizand
mudanças em uma parte do sistema podem

E o

- É ur à ideia de que os dife u seja, os **com** úmero possível de dep
- Em amento funcionam de r acto que mudanças em uma parte do sistema podem ter sobre outras partes

diferentes módulos, classes, me
componentes de um sistema, d
dependências uns dos outros

Características de <u>Baixo Acoplamento</u>

- **Independência:** Componentes com baixo acoplamento são mais independentes, o que significa que podem ser modificados, substituídos ou removidos sem afetar significativamente outras partes do sistema.
- **Facilidade de Manutenção:** A manutenção é facilitada, pois mudanças em um módulo têm menos probabilidade de causar efeitos colaterais em outros módulos.
- **Reutilização:** São mais facilmente reutilizáveis, já que eles não dependem fortemente de outros componentes específicos.
- **Testabilidade:** É mais fácil de testar, uma vez que os módulos podem ser testados de forma isolada, sem a necessidade de configurar muitas dependências.

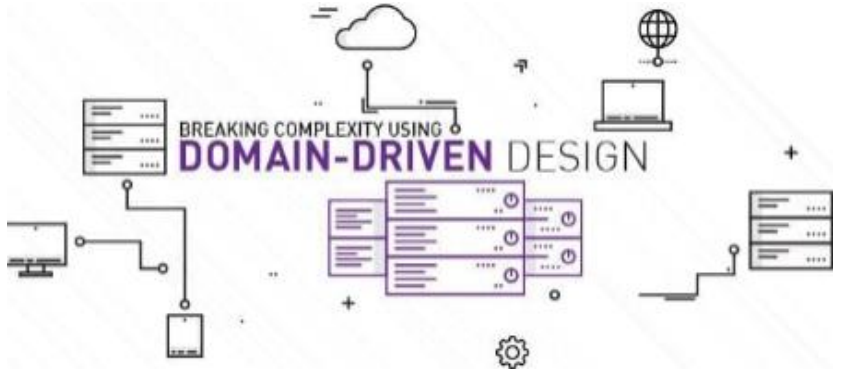
1. **KISS (Keep It Simple, Stupid)**: Você deve manter suas soluções o mais simples possível. Evite criar soluções complicadas quando uma abordagem mais simples atender ao objetivo. Isso ajuda a reduzir a complexidade do código, facilitando a manutenção e a compreensão.
2. **DRY (Don't Repeat Yourself)**: O DRY incentiva a reutilização de código. Em vez de repetir você deve modularizar o código e criar funções ou componentes reutilizáveis sempre que possível. Isso torna o código mais limpo, mais eficiente e mais fácil de manter.
3. **YAGNI (You Ain't Gonna Need It)**: O YAGNI aconselha a não adicionar funcionalidades ou recursos ao seu código até que você realmente precise deles. Evite o desperdício de tempo e esforço em recursos que não são necessários no momento presente. Isso mantém o código mais simples e evita a sobrecarga de funcionalidades não utilizadas.

- Lembrando que esses princípios são diretrizes úteis para escrever código de qualidade e manutenível. Eles promovem a simplicidade, a reutilização e a eficiência no desenvolvimento de software.

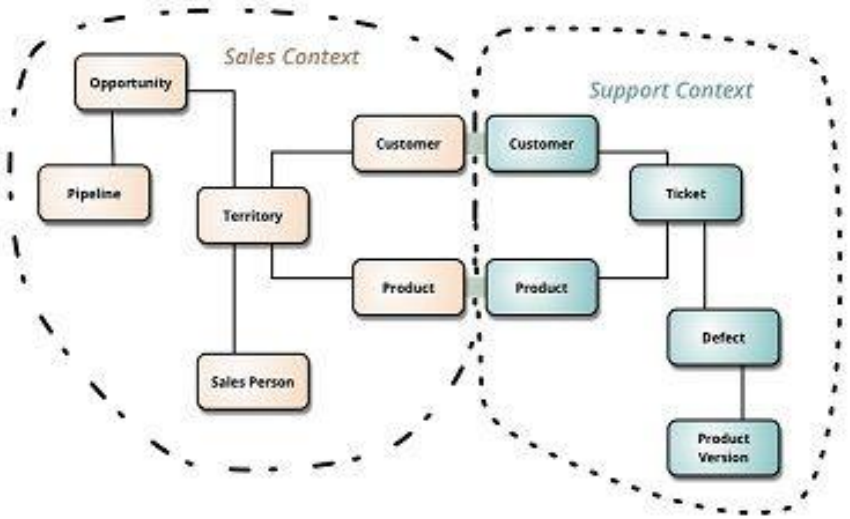
Domain-Driven Design

- Abordagem de desenvolvimento de software que se concentra na criação de modelos de domínio eficazes e na colaboração estreita entre especialistas do domínio e desenvolvedores de software.
- **Bounded Contexts (Contextos Delimitados)**:
  - São limites lógicos que definem onde um determinado modelo de domínio é válido dentro de um sistema maior. Eles são uma parte fundamental do DDD para evitar conflitos e ambiguidades na linguagem e no significado dos termos do domínio, especialmente em sistemas complexos.
- **Linguagem Ubíqua**:
  - Cada Bounded Context deve ter sua própria "Linguagem Ubíqua" (também conhecida como "Linguagem Comum" ou "Linguagem Compartilhada"). Isso significa que dentro de cada contexto, os termos de domínio devem ser definidos de forma clara e específica, e essa definição deve ser compartilhada entre todos os membros da equipe de desenvolvimento e especialistas do domínio que trabalham nesse contexto.

Domain-Driven Design



- **<u>Context Map (Mapa de Contexto)</u>**
  - É uma ferramenta usada no Domain-Driven Design (DDD) para representar visualmente as relações entre os Bounded Contexts em um sistema de software. Ele ajuda a documentar e comunicar como os diferentes contextos delimitados interagem entre si. O Context Map é especialmente útil em sistemas complexos com vários Bounded Contexts, onde a clareza sobre as dependências e interações é fundamental.



Lei de Demeter

- Um objeto deve ter conhecimento limitado sobre outros objetos **(encapsulamento)**, e ele deve interagir apenas com seus amigos mais próximos. Não deve se intrometer nas entranhas de objetos que não são diretamente relacionados a ele.
- **Baixo Acoplamento**: O acoplamento entre classes ou módulos é reduzido, o que facilita a manutenção e a evolução do código.
- **Maior Reusabilidade**: Classes e módulos se tornam mais independentes, o que torna mais fácil reutilizá-los em diferentes partes do sistema.
- **Legibilidade Melhorada**: O código segue relações mais claras e diretas entre objetos, tornando-o mais fácil de entender.
- **Testabilidade Aprimorada**: Como os objetos têm menos dependências, é mais fácil criar testes unitários para eles.

```java
class Motor {
    private int potencia;

    public Motor(int potencia) {
        this.potencia = potencia;
    }

    public int getPotencia() {
        return potencia;
    }
}
```

```java
class Carro {
    private Motor motor;

    public Carro(Motor motor) {
        this.motor = motor;
    }

    public Motor getMotor() {
        return motor;
    }
}
```

```java
class Motorista {
    public void acelerar(Carro carro) {
        int potencia = carro.getMotor().getPotencia();
        System.out.println("Acelerando com potência: " + potencia);
    }
}
```

```java
class Carro {
    private Motor motor;

    public Carro(Motor motor) {
        this.motor = motor;
    }

    public int getPotenciaDoMotor() {
        return motor.getPotencia();
    }
}
```

```java
class Motorista {
    public void acelerar(Carro carro) {
        int potencia = carro.getPotenciaDoMotor();
        System.out.println("Acelerando com potência: " + potencia);
    }
}
```

Entregáveis – Aula 03

Questão 1

Single Responsability Principle

Information Expert

High Cohesion

Pure Fabrication

```java
public class OrdemCompra {
    private List<Item> itens;

    public void addItem(Item item) {
        itens.add(item);
    }

    public void imprimirRelatorio() {
        // Imprime um relatório da ordem
    }
}

class Item {
    double getPreco() {
        return 2;
    }
}
```

```java
class Calculadora {
    public double calculaTotalOrdemCompra(List<Item> itens) {
        double total = 0;
        for (Item item : itens) {
            total += item.getPreco();
        }
        return total;
    }
}
```

a) **Princípio da Responsabilidade Única (SRP):** Identifique uma violação deste princípio no código acima e sugira como corrigir.

b) **Padrão Especialista da Informação (Information Expert):** O método `calculaTotalOrdemCompra` na classe `Calculadora` respeita o padrão de Especialista da Informação? Justifique sua resposta.

c) **Padrão Alta Coesão (High Cohesion):** A classe `OrdemCompra` possui alta coesão? Se não, o que poderia ser modificado para melhorar sua coesão?

d) **Padrão Fabricação Pura (Pure Fabrication):** Como o padrão Fabricação Pura pode ser aplicado na organização desse código para melhorar a separação de responsabilidades?

```java
public class OrdemCompra {
    private List<Item> itens;

    public void addItem(Item item) {
        itens.add(item);
    }

    public double calculateTotal() {
        return itens.stream().mapToDouble(Item::getPreco).sum();
    }
```

```java
public class RelatorioOrdemCompra {
    public void imprimirRelatorio(OrdemCompra ordemCompra) {
        // Imprime um relatório da ordem de compra
    }
}
```

Questão 2

Open-Closed Principle

Dependency Inversion Principle

```java
public class User {
    private EmailNotificationService emailNotificationService;

    public User() {
        this.emailNotificationService = new EmailNotificationService();
    }

    public void notifyUser(String message) {
        emailNotificationService.sendNotification(message);
    }
}

class EmailNotificationService {
    public void sendNotification(String message) {
        // Envia e-mail
    }
}
```

a) O código acima está em conformidade com o **Princípio Aberto-Fechado (OCP)**? Justifique sua resposta e sugira como o código poderia ser modificado para melhor atender a esse princípio.

b) Identifique uma violação do **Princípio de Inversão de Dependência (DIP)** no código acima. Como você poderia refatorar o código para seguir o DIP, mantendo a funcionalidade original?

```java
public interface NotificationService {
    void sendNotification(String message);
}

public class EmailNotificationService implements NotificationService {
    public void sendNotification(String message) {
        // Envia e-mail
    }
}

public class User {
    private NotificationService notificationService;

    public User(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    public void notifyUser(String message) {
        notificationService.sendNotification(message);
```

Questão 3

Liskov Substitution Principle

Interface Segregation Principle

```java
abstract class FormaGeometrica {
    public abstract double calculaArea();
    public abstract double calculaVolume();
}
```

```java
class Circulo extends FormaGeometrica {
    private double raio;
    public Circulo(double raio) {…
    @Override
    public double calculaArea() {
        return Math.PI * raio * raio;
    }
    @Override
    public double calculaVolume() {
        throw new UnsupportedOperationException(
            "Círculos não possuem Volume");
    }
}
```

```java
class Cubo extends FormaGeometrica {
    private double lado;
    public Cubo(double lado) {…
    @Override
    public double calculaArea() {
        return 6 * lado * lado;
    }
    @Override
    public double calculaVolume() {
        return lado * lado * lado;
    }
}
```

```java
public class CalculadoraMedidas {
    public double calculaArea(FormaGeometrica formaGeometrica) {
        return formaGeometrica.calculaArea();
    }
    public double calculaVolume(FormaGeometrica formaGeometrica) {
        return formaGeometrica.calculaVolume();
    }
    public static void main(String[] args) {
        FormaGeometrica circle = new Circulo(5);
        FormaGeometrica cube = new Cubo(3);
        CalculadoraMedidas calculator = new CalculadoraMedidas();
        System.out.println("Área do círculo: " + calculator.calculaArea(circle));
        System.out.println("Área do cubo: " + calculator.calculaArea(cube));
        System.out.println("Volume do cubo: " + calculator.calculaVolume(cube));
    }
}
```

a) O código acima está em conformidade com o **Princípio de Substituição de Liskov (LSP)**? Justifique sua resposta e sugira uma maneira de refatorar o código para aderir a esse princípio.

b) O código fornecido respeita o **Princípio de Segregação de Interfaces (ISP)**? Explique sua resposta e proponha uma modificação para melhorar a aderência ao ISP.

```java
// Interface superior que todas as formas geométricas irão implementar
interface FormaGeometrica {
    double calculaArea();
}

// Interface para formas bidimensionais, que só precisam calcular a área
interface FormaBidimensional extends FormaGeometrica {
    // Nenhum método adicional, pois o cálculo de área é comum a todas as formas
}

// Interface para formas tridimensionais, que precisam calcular tanto a área quanto o
interface FormaTridimensional extends FormaGeometrica {
    double calculaVolume();
}
```

```java
public class CalculadoraMedidas {
    public double calculaArea(FormaGeometrica formaGeometrica) {
        return formaGeometrica.calculaArea();
    }

    public double calculaVolume(FormaTridimensional formaTridimensional) {
        return formaTridimensional.calculaVolume();
    }

    public static void main(String[] args) {
        FormaGeometrica circle = new Circulo(5);
        FormaTridimensional cube = new Cubo(3);

        CalculadoraMedidas calculator = new CalculadoraMedidas();
        System.out.println("Área do círculo: " + calculator.calculaArea(circle));
        System.out.println("Área do cubo: " + calculator.calculaArea(cube));
        System.out.println("Volume do cubo: " + calculator.calculaVolume(cube));
```

Questão 4

Creator

Low Coupling

```java
public class Pedido {
    private Fatura fatura;
    public Pedido() {
        this.fatura = FaturaFactory.criarFatura(this);
    }
    public Fatura getFatura() {
        return fatura;
    }
}
```

```java
class FaturaFactory {
    public static Fatura
        criarFatura(Pedido pedido) {
            return new Fatura(pedido);
    }
}
```

```java
class Fatura {
    private Pedido pedido;
    public Fatura(Pedido pedido) {
        this.pedido = pedido;
    }
    public Pedido getPedido() {
        return this.pedido;
    }
    // Métodos adicionais
}
```

a) O código acima utiliza o **padrão de projeto Creator**? Justifique sua resposta explicando qual classe, neste contexto, é responsável pela criação de objetos e como isso afeta a coesão.

b) Como o uso de uma fábrica (`FaturaFactory`) para criar a `Fatura` contribui para o **baixo acoplamento** entre as classes? Explique como essa abordagem afeta a flexibilidade e a manutenção do código.

Questão 5

Polimorphism

Protected Variation

```java
public class Pagamentos {
    public void processarPagamento(double valor, String metodo) {
        if (metodo.equals("CreditCard")) {
            // Processa pagamento com cartão de crédito
        } else if (metodo.equals("PIX")) {
            // Processa pagamento com PIX
        }
    }
}
```

1. Polimorfismo:

a) O código fornecido utiliza o conceito de polimorfismo? Justifique sua resposta e proponha uma refatoração para aplicar o polimorfismo ao processamento de pagamentos, tornando o código mais extensível e menos dependente de condições baseadas em strings.

2. Proteção de Variações:

b) Como o código atual lida com variações no método de pagamento e como isso pode ser melhorado para proteger o código contra mudanças futuras? Explique como uma abordagem alternativa pode melhorar a proteção de variações.

```java
interface MetodoPagamento {
    void processar(double valor);
}
```

```java
class PagamentoComCartaoCredito implements MetodoPagamento {
    @Override
    public void processar(double valor) {
        // Processa pagamento com cartão de crédito
        System.out.println("Processando pagamento com cartão de crédito: " + valor);
    }
}
```

```java
class PagamentoComPIX implements MetodoPagamento {
    @Override
    public void processar(double valor) {
        // Processa pagamento com PIX
        System.out.println("Processando pagamento com PIX: " + valor);
    }
}
```

```java
public class Pagamentos {
    private MetodoPagamento metodoPagamento;
    // Injeção de dependência via construtor
    public Pagamentos(MetodoPagamento metodoPagamento) {···

    public void processarPagamento(double valor) {
        metodoPagamento.processar(valor);
    }
    public static void main(String[] args) {
        // Configuração do método de pagamento via injeção de dependência
        MetodoPagamento cartaoCredito = new PagamentoComCartaoCredito();
        MetodoPagamento pix = new PagamentoComPIX();

        // Injeção de dependência ao criar instâncias de Pagamentos
        Pagamentos pagamentosComCartaoCredito = new Pagamentos(cartaoCredito);
        Pagamentos pagamentosComPIX = new Pagamentos(pix);

        pagamentosComCartaoCredito.processarPagamento(100.0);
        pagamentosComPIX.processarPagamento(50.0);
    }
}
```

Bom senso é a chave!!!

- A aplicação de princípios, padrões e boas práticas, deve ser guiada pelo bom senso e pela consideração do contexto em que o código será mantido e evoluído.

- Se a violação resulta em um código mais limpo, intuitivo e fácil de manter, pode ser preferível aceitar a violação em vez de criar métodos adicionais que apenas obscurecem a simplicidade do design.

```java
class Estado {
    private String nome;

    public String getNome() {
        return nome;
    }
}
```

```java
class Cidade {
    private Estado estado;

    public Estado getEstado() {
        return estado;
    }
}
```

```java
class Pessoa {
    private Cidade cidade;

    public Cidade getCidade() {
        return cidade;
    }
}
```

```java
Pessoa pessoa = new Pessoa();
String nomeDoEstado = pessoa.getCidade().getEstado().getNome();
```

Vamos programar!!!

spring®

What Spring can do

Microservices

Quickly deliver production-grade features with independently evolvable microservices.

Reactive

Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.

Cloud

Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.

Web apps

Frameworks for fast, secure, and responsive web applications connected to any data store.

Serverless

The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.

Event Driven

Integrate with your enterprise. React to business events. Act on your streaming data in realtime.

Batch

Automated tasks. Offline processing of data at a time to suit you.

Spring Framework e Spring Boot

- O **Spring Framework** é um framework de código aberto que fornece um ambiente de desenvolvimento abrangente para aplicativos Java. Ele foi lançado pela primeira vez em 2002 e desde então se tornou uma escolha popular para o desenvolvimento de aplicativos corporativos. Algumas características-chave do Spring Framework incluem:

  - **Injeção de Dependência (Dependency Injection - DI)**
  - **Programação Orientada a Aspectos (Aspect-Oriented Programming – AOP)**
  - **MVC (Model-View-Controller**)
  - **Integração com Banco de Dados**
  - **Segurança**

Spring Framework e Spring Boot

- O **Spring Boot** é uma extensão do Spring Framework que simplifica muito o desenvolvimento de aplicativos Java. Ele foi projetado para acelerar o processo de configuração e desenvolvimento, permitindo que os desenvolvedores se concentrem mais na lógica de negócios e menos na configuração. Algumas características do Spring Boot incluem:

  - **Configuração Automática**
  - **Embedded Web Server**
  - **Pronto para Produção**
  - **Spring Boot Starter**
  - **Facilidade de Teste**
  - **Ampla Comunidade**

Entregáveis

Vamos lá. Está na hora!!!

Ciência da
Computação