semana 5- aula 01
Pilares da programação orientada a objetos
Introdução à programação orientada a objetos e seus pilares
Código da aula: [SIS]ANO1C3B1S5A1

Objetivos da Aula:

- ❖ Conhecer os fundamentos da programação orientada a objetos (POO) no contexto de seus conceitos e aplicações.
- ❖ Prestar apoio técnico na elaboração da documentação de sistemas. Conhecer frameworks de desenvolvimento ágeis, utilizando tecnologias de CI e CD que trabalham em conjunto com a segurança do ambiente funcional e as entregas divididas em partes que agregam valor ao negócio de forma rápida. Trabalhar a resolução de problemas computacionais.
- ❖ Recurso audiovisual para exibição de vídeos e imagens;
- ❖ Caderno para anotações. Recursos didáticos Competências da unidade (técnicas e socioemocionais)

Exposição:

Paradigmas de Programação: Uma Definição

Paradigmas de programação são abordagens ou estilos fundamentais de construir a estrutura e a organização de um programa de computador. Eles representam diferentes maneiras de pensar sobre o problema a ser resolvido e de como o código deve ser escrito e executado. Cada paradigma oferece um conjunto de conceitos, princípios e técnicas que influenciam a forma como os desenvolvedores modelam problemas e implementam soluções.

Além da Programação Orientada a Objetos, outros paradigmas de programação comuns incluem:

- Programação Imperativa: Foca em descrever passo a passo como o programa deve executar, alterando o estado do programa através de comandos. Exemplos incluem C e Pascal.
- Programação Declarativa: Foca em descrever o que o programa deve alcançar, sem especificar explicitamente como fazê-lo. Exemplos incluem SQL e Prolog.
- Programação Funcional: Trata a computação como a avaliação de funções matemáticas puras, evitando efeitos colaterais e mutabilidade de estado. Exemplos incluem Haskell e Lisp.
- Programação Procedural: Organiza o código em um conjunto de procedimentos (funções ou sub-rotinas) que realizam tarefas específicas. É

uma forma de programação imperativa. Exemplo clássico é o C (em sua forma mais básica).

É importante notar que muitos projetos de software modernos podem utilizar uma abordagem multi-paradigma, combinando elementos de diferentes paradigmas para aproveitar seus respectivos benefícios em diferentes partes do sistema. A escolha do paradigma ou da combinação de paradigmas depende das características do problema a ser resolvido, dos requisitos do projeto e das preferências da equipe de desenvolvimento.

Programação Orientada a Objetos (POO): Uma Definição Detalhada

A programação orientada a objetos (POO) é um paradigma de programação com base na concepção de "objetos" que podem conter dados de campos e de códigos na forma de procedimentos. Esses objetos combinam dados (atributos) e comportamentos (métodos) que operam sobre esses dados. A POO oferece uma estrutura clara para programas de software que são fáceis de compreender, modificar e manter. Em vez de se concentrar em funções que manipulam dados separados, a POO modela o mundo real em entidades coesas que interagem entre si.

Em essência, a POO busca criar um código mais modular, reutilizável, flexível e fácil de manter, através da abstração de entidades do mundo real em componentes de software.

Conceitos Fundamentais da POO:

Para entender a POO em profundidade, é crucial compreender seus quatro pilares principais:

1. Encapsulamento: Este princípio envolve agrupar os dados (atributos) e os métodos (comportamentos) que operam sobre esses dados dentro de uma única unidade, a classe. O encapsulamento também implementa o ocultamento de dados, protegendo o estado interno de um objeto de acesso direto e modificação externa não autorizada. O acesso aos dados é controlado através de métodos específicos (getters e setters), permitindo a implementação de lógica de validação ou manipulação antes que os dados sejam acessados ou modificados.

   Exemplo: Considere uma classe `ContaBancaria`. Os atributos como `saldo` e `numeroConta` são encapsulados dentro da classe. O acesso e a modificação do `saldo` só podem ser feitos através de métodos como `depositar()` e `sacar()`, que podem incluir verificações de segurança (por exemplo, não permitir

saques negativos).

2. Herança: A herança é um mecanismo que permite que uma classe (a subclasse ou classe filha) herde propriedades (atributos) e comportamentos (métodos) de outra classe (a superclasse ou classe pai). Isso promove a reutilização de código, pois características comuns podem ser definidas na superclasse e automaticamente disponibilizadas para suas subclasses. As subclasses também podem adicionar novos atributos e métodos ou sobrescrever os métodos herdados para fornecer um comportamento mais específico.

   Exemplo: Podemos ter uma superclasse `Veiculo` com atributos como `marca`, `modelo` e métodos como `ligar()` e `desligar()`. Subclasses como `Carro` e `Moto` podem herdar esses atributos e métodos, e também definir seus próprios atributos (por exemplo, `numeroPortas` em `Carro`, `cilindrada` em `Moto`) e comportamentos específicos (por exemplo, um método `acelerarRapido()` em `Moto`).

3. Polimorfismo: Polimorfismo (que significa "muitas formas") permite que objetos de diferentes classes respondam ao mesmo método de maneiras distintas. Isso é alcançado através da sobrescrita de métodos (overriding) ou da sobrecarga de métodos (overloading).

   - Sobrescrita (Overriding): Uma subclasse fornece uma implementação específica para um método que já está definido em sua superclasse.
   - Sobrecarga (Overloading): Uma classe pode ter múltiplos métodos com o mesmo nome, mas com diferentes listas de parâmetros (em número, tipo ou ordem). O compilador ou interpretador determina qual método chamar com base nos argumentos fornecidos.

4. Exemplo: Voltando ao exemplo de `Veiculo`, tanto `Carro` quanto `Moto` podem ter um método `acelerar()`. No entanto, a implementação desse método será diferente em cada classe, refletindo a forma como cada veículo acelera. Outro exemplo seria um método `calcularArea()` em uma classe `Forma`. Subclasses como `Retangulo` e `Circulo` implementariam `calcularArea()` de maneiras diferentes, com base em suas respectivas fórmulas.

5. Abstração: A abstração envolve simplificar a complexidade do mundo real, modelando apenas os aspectos essenciais de um objeto que são relevantes para o contexto em questão. Ela se concentra no "o quê" um objeto faz, em vez de "como" ele faz. Classes abstratas e interfaces são mecanismos comuns para implementar a abstração.

   - Classes Abstratas: Classes que não podem ser instanciadas diretamente e podem conter métodos abstratos (métodos sem

implementação). Subclasses concretas devem fornecer implementações para esses métodos abstratos.
    - Interfaces: Contratos que definem um conjunto de métodos que uma classe deve implementar. Uma classe pode implementar múltiplas interfaces.
6. Exemplo: Podemos definir uma classe abstrata `Animal` com um método abstrato `emitirSom()`. Subclasses concretas como `Cachorro` e `Gato` seriam obrigadas a implementar o método `emitirSom()` com seus respectivos sons ("Au Au" e "Miau"). A abstração nos permite trabalhar com objetos `Animal` de forma genérica, sabendo que todos eles podem emitir um som, sem nos preocuparmos com os detalhes específicos de cada animal.

- Reusabilidade de Código: A herança permite que novas classes sejam construídas a partir de classes existentes, economizando tempo e esforço de desenvolvimento.
- Modularidade: A POO organiza o código em unidades independentes (objetos), tornando-o mais fácil de entender, depurar e manter.
- Flexibilidade: O polimorfismo permite que o código se adapte a diferentes tipos de objetos de forma transparente.
- Manutenibilidade: A estrutura orientada a objetos tende a produzir um código mais organizado e fácil de modificar e estender.
- Modelagem do Mundo Real: A POO facilita a modelagem de sistemas complexos, pois os objetos podem representar entidades do mundo real e suas interações.

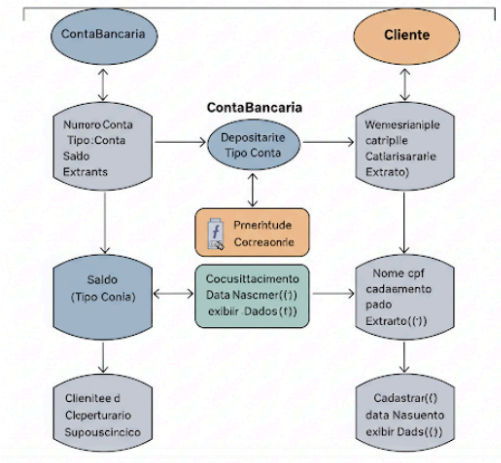
Exemplo Prático Simplificado em Python:

Python

```python
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def emitir_som(self):
        print("Som genérico de animal")

class Cachorro(Animal):
    def emitir_som(self):
        print("Au Au!")

class Gato(Animal):
    def emitir_som(self):
        print("Miau!")

# Criando objetos
cachorro = Cachorro("Rex")
gato = Gato("Mingau")

# Polimorfismo em ação
animais = [cachorro, gato]
for animal in animais:
    print(f"{animal.nome} faz:")
    animal.emitir_som()
```

Neste exemplo, `Animal` é a superclasse, e `Cachorro` e `Gato` são subclasses que herdam de `Animal`. Ambas as subclasses sobrescrevem o método `emitir_som()` para fornecer seus próprios comportamentos específicos, demonstrando o polimorfismo.

Exemplo de um programa usando POO:



Diagrama

O Diagrama de Classes UML para este exemplo seria representado da seguinte forma:

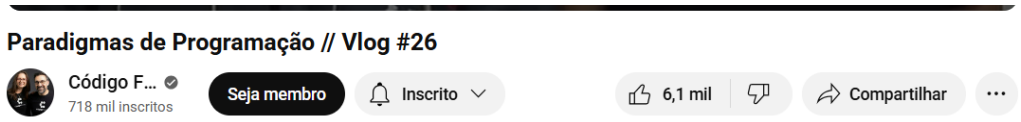
Snippet de código

```
@startuml
class ContaBancaria {
  - numeroConta: String
  - saldo: Double
  - titular: Cliente
  + depositar(valor: Double): void
  + sacar(valor: Double): boolean
  + obterSaldo(): Double
  + obterNumeroConta(): String
}

class Cliente {
  - nome: String
  - cpf: String
  + obterNome(): String
  + obterCpf(): String
}

ContaBancaria --* Cliente : titular

@enduml
```

Vídeos:

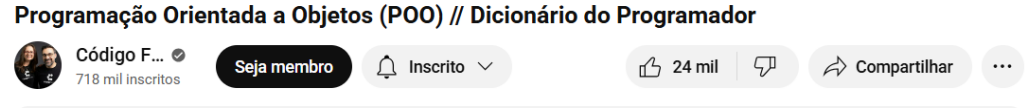
Paradigmas de Programação

https://youtu.be/EefVmQ2wPIM?si=5yfUf67o5VCr8A73



Programação Orientada a Objeto

https://youtu.be/QY0Kdg83orY?si=Vb6vBlrgzSRuK-cK



semana 5- aula 02
Pilares da programação orientada a objetos
Introdução à programação orientada a objetos e seus pilares
Código da aula: [SIS]ANO1C3B1S5A2

Objetivos da Aula:

- Conhecer os fundamentos da programação orientada a objetos (POO) no contexto de seus conceitos e aplicações.
- Prestar apoio técnico na elaboração da documentação de sistemas. Conhecer frameworks de desenvolvimento ágeis, utilizando tecnologias de CI e CD que trabalham em conjunto com a segurança do ambiente funcional e as entregas divididas em partes que agregam valor ao negócio de forma rápida. Trabalhar a resolução de problemas computacionais.
- Recurso audiovisual para exibição de vídeos e imagens;
- Caderno para anotações. Recursos didáticos Competências da unidade (técnicas e socioemocionais)

Exposição:

Slide 5
Classe
É uma estrutura fundamental na POO que serve como um molde para criar algo mais concreto: os objetos. Pense na classe como uma receita de bolo. A receita detalha os ingredientes e passos necessários para fazer o bolo, mas não é o bolo em si. Da mesma forma, uma classe detalha as características e os comportamentos (por meio de atributos e métodos) que os objetos derivados dela terão, mas não é o objeto. Por exemplo, uma classe "Automóvel" define que todo automóvel tem uma marca, um modelo, uma cor e uma função para "acelerar" ou "frear". No entanto, a classe "Automóvel" em si não é um carro que você pode dirigir; ela apenas descreve o que um carro é e faz.

Slide 6
Objeto
São as entidades criadas usando classes como modelo. Continuando com a analogia da receita, um objeto é o bolo que você faz usando a receita. Cada bolo pode ter sabores variados ou decorações diferentes, mas todos seguem a base da receita. No caso da classe "Automóvel", um objeto seria um carro específico, como um Ford Mustang vermelho de 2020. Esse carro específico tem características concretas (atributos como marca, modelo e cor) e pode realizar ações específicas (comportamentos definidos pelos métodos como acelerar e frear).

Slide 7
Métodos
São como as instruções na receita que dizem o que fazer com os ingredientes. Em uma classe, métodos definem ações ou comportamentos que os objetos podem executar. Eles são como funções que operam dentro de uma classe e muitas vezes manipulam os atributos dessa classe. Por exemplo, um método em uma classe "Automóvel" pode ser "ligarMotor", que muda o estado do carro de "motor desligado" para "motor ligado". Outro método pode ser "exibirDetalhes", que fornece informações sobre o carro, como a marca, o modelo e a cor.

Exemplo

O Diagrama de Classes UML para este exemplo seria representado da seguinte forma:

Snippet de código

```
@startuml
class ContaBancaria {
  - numeroConta: String
  - saldo: Double
  - titular: Cliente
  + depositar(valor: Double): void
  + sacar(valor: Double): boolean
  + obterSaldo(): Double
  + obterNumeroConta(): String
}

class Cliente {
  - nome: String
  - cpf: String
  + obterNome(): String
  + obterCpf(): String
}

ContaBancaria --* Cliente : titular

@enduml
```

semana 5- aula 03
Pilares da programação orientada a objetos
Introdução à programação orientada a objetos e seus pilares
Código da aula: [SIS]ANO1C3B1S5A3

Objetivos da Aula:

- ❖ Conhecer os fundamentos da programação orientada a objetos (POO) no contexto de seus conceitos e aplicações.
- ❖ Prestar apoio técnico na elaboração da documentação de sistemas. Conhecer frameworks de desenvolvimento ágeis, utilizando tecnologias de CI e CD que trabalham em conjunto com a segurança do ambiente funcional e as entregas divididas em partes que agregam valor ao negócio de forma rápida. Trabalhar a resolução de problemas computacionais.
- ❖ Recurso audiovisual para exibição de vídeos e imagens;
- ❖ Caderno para anotações. Recursos didáticos Competências da unidade (técnicas e socioemocionais)

Exposição:

slide 6

O design orientado a objetos é uma fase crucial no desenvolvimento de software, em que a organização do sistema é planejada usando objetos e classes. Duas áreas principais nesse domínio são os padrões de design e os princípios SOLID. Padrões de projeto São soluções típicas para problemas comuns em design de software. Eles representam as melhores práticas que um programador pode utilizar para resolver problemas de design em seu código.

Slide 08

Os princípios SOLID são um conjunto de cinco princípios de design de software que visam tornar os projetos orientados a objetos mais compreensíveis, flexíveis, manuteníveis e extensíveis. Eles foram introduzidos por Robert C. Martin (Uncle Bob) e são considerados pilares fundamentais para a criação de software robusto e de qualidade. O acrônimo SOLID representa a primeira letra de cada um dos cinco princípios:

S - Single Responsibility Principle (Princípio da Responsabilidade Única)

- Definição: Uma classe deve ter apenas uma razão para mudar. Em outras palavras, uma classe deve ter uma única responsabilidade. Se uma classe assume muitas responsabilidades, qualquer alteração em uma dessas responsabilidades pode afetar as outras, levando a um código mais frágil e difícil de manter.

- Objetivo: Promover a alta coesão (elementos relacionados dentro de uma classe) e o baixo acoplamento (dependência mínima entre classes).
- Exemplo: Considere uma classe `Relatorio` que é responsável por gerar um relatório e também por persistir esse relatório em um banco de dados. Segundo o SRP, essas duas responsabilidades deveriam ser separadas em duas classes distintas: uma classe `GeradorDeRelatorio` e uma classe `PersistenciaDeRelatorio`. Assim, uma mudança no formato do relatório não afetaria a lógica de persistência, e vice-versa.

O - Open/Closed Principle (Princípio Aberto/Fechado)

- Definição: As entidades de software (classes, módulos, funções, etc.) devem ser abertas para extensão, mas fechadas para modificação.[1] Isso significa que você deve ser capaz de adicionar novas funcionalidades sem alterar o código existente.[2]
- Objetivo: Tornar o software mais extensível e menos propenso a introduzir bugs ao adicionar novos comportamentos.
- Exemplo: Imagine um sistema que calcula a área de diferentes formas geométricas (círculo, retângulo, triângulo). Em vez de modificar uma classe `CalculadorDeArea` cada vez que uma nova forma é adicionada, você pode criar uma interface `Forma` com um método `calcularArea()`. Cada forma concreta implementaria essa interface. A classe `CalculadorDeArea` trabalharia com a interface `Forma`, estando aberta para extensão (novas formas) sem precisar ser modificada.

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

- Definição: Subtipos devem ser substituíveis por seus tipos base sem alterar a correção do programa. Em outras palavras, se você tem uma classe base e uma classe derivada, um objeto da classe derivada deve poder ser usado em qualquer lugar onde um objeto da classe base é esperado, sem que o programa "quebre".
- Objetivo: Garantir a coesão e a integridade da hierarquia de herança.
- Exemplo: Considere uma classe base `Pato` com um método `voar()`. Se você tem uma subclasse `PatoMudo` que não pode voar, violar o LSP. Um código que espera que todo `Pato` possa `voar()` falharia ao receber um `PatoMudo`. Uma solução seria repensar a hierarquia ou criar interfaces separadas para comportamentos voadores e não voadores.

I - Interface Segregation Principle (Princípio da Segregação da Interface)

- Definição: Nenhum cliente (classe que utiliza uma interface) deve ser forçado a depender de métodos que não usa. Interfaces grandes devem ser divididas

em interfaces menores e mais específicas, para que os clientes só precisem conhecer os métodos que são relevantes para eles.
- Objetivo: Reduzir o acoplamento e tornar o sistema mais flexível.
- Exemplo: Imagine uma interface `Trabalhador` com métodos como `trabalhar()`, `comer()` e `descansar()`. Se você tem um tipo de trabalhador que não come (por exemplo, um robô), ele seria forçado a implementar um método `comer()` que não faz sentido para ele. A solução seria segregar a interface `Trabalhador` em interfaces menores, como `Trabalhador`, `Alimentavel` e `Descansavel`, permitindo que cada classe implemente apenas as interfaces relevantes.

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

- Definição:
    1. Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
    2. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.[3]
- Objetivo: Reduzir o acoplamento entre módulos, tornando o sistema mais flexível, testável e fácil de manter. A dependência em abstrações (interfaces ou classes abstratas) em vez de implementações concretas permite que diferentes implementações sejam facilmente trocadas.
- Exemplo: Considere uma classe `ServicoDeEmail` que depende diretamente de uma classe concreta `ImplementacaoDeEnvioDeEmail`. Isso cria um alto acoplamento. O DIP sugere que ambos deveriam depender de uma abstração, como uma interface `EnviadorDeEmail`. A classe `ServicoDeEmail` dependeria da interface, e diferentes implementações de envio de email (como `ImplementacaoDeEnvioDeEmail`, `OutraImplementacaoDeEnvioDeEmail`) dependeriam dessa interface. Isso permite trocar a implementação de envio de email sem modificar a classe `ServicoDeEmail`.

Em resumo, os princípios SOLID visam criar um código:

- Mais fácil de entender: Cada classe tem um propósito claro.
- Mais fácil de manter: Alterações em uma parte do sistema têm menos probabilidade de afetar outras partes.
- Mais fácil de estender: Novas funcionalidades podem ser adicionadas sem modificar o código existente.
- Mais fácil de testar: Classes com responsabilidades únicas são mais fáceis de isolar e testar.
- Mais reutilizável: Classes bem definidas e com baixo acoplamento podem ser reutilizadas em diferentes partes do sistema ou em outros projetos.

A aplicação dos princípios SOLID, embora possa exigir um planejamento inicial mais cuidadoso, resulta em um código mais robusto, adaptável e sustentável a longo prazo.