# Padrões de Projeto

## Padrões: Criacionais, Estruturais e Comportamentais

Alunos:
Joselio Francisco da Silva Júnior
Luís Felipe Dantas Costa
Yan Lemke de Castro

# 01

# Padrões Criacionais

Exemplo: Builder

# Exemplo

Exemplo do Padrão Criacional em **Java**

# 02

# Padrões Estruturais

Exemplos: Bridge, Flyweight e Proxy

# Definição e Objetivo

Padrões estruturais lidam com a **organização de classes e objetos** para formar estruturas maiores de forma **flexível** e **eficiente**.

Facilitar a **manutenção**, **expansão** e o **reuso de código** através de composições bem definidas.

# Bridge Pattern (Ponte)

## Intenção:

- **Desacoplar** uma abstração da sua implementação, permitindo que as duas **evoluam** independentemente.

## Quando usar:

- Quando há múltiplas variações em classes.
- Para evitar a explosão de subclasses.

## Vantagens:

- Reduz o acoplamento entre código de alto e baixo nível.
- Facilita a expansão do sistema.

# Flyweight Pattern (Peso Leve)

## Intenção:

- **Minimizar** o uso de **memória** compartilhando instâncias de objetos comuns.

## Quando usar:

- Quando há **grande número** de objetos semelhantes.
- Em sistemas com **restrições de memória**.

## Vantagens:

- **Redução** significativa no uso de memória.
- Melhora o **desempenho** em larga escala.

## Conceito-chave:

- Divisão entre estado intrínseco **(compartilhado)** e extrínseco **(fornecido pelo contexto).**

# Proxy Pattern (Representante)

## Intenção:

- Fornecer um **substituto** ou **representante** para controlar o acesso a outro objeto.

## Tipos comuns de Proxy:

- **Virtual Proxy:** adia a criação de objetos pesados.
- **Protection Proxy:** controla permissões de acesso.
- **Remote Proxy:** representa um objeto em outra máquina.

## Vantagens:

- Adiciona **funcionalidades** sem alterar o **objeto real**.
- Garante **controle de acesso, economia de recursos ou integração** com sistemas externos.

# Comparativo dos Padrões

| Padrão | Intenção Principal | Benefício Principal |
|---|---|---|
| Bridge | Separar abstração da implementação | Permite expansão independente |
| Flyweitght | Compartilhar objetos para economizar memória | Eficiência no uso de recursos |
| Proxy | Controlar o acesso a um objeto | Adição de lógica de controle e otimização |

# Conclusão

- **Padrões estruturais** oferecem **soluções robustas** para estruturar o código.
- **Bridge** foca em **flexibilidade** e **expansão.**
- **Flyweight** visa **eficiência** e **economia.**
- **Proxy** adiciona **camadas de controle** e **proteção.**

- **Essenciais para sistemas escaláveis e bem projetados.**

# UML

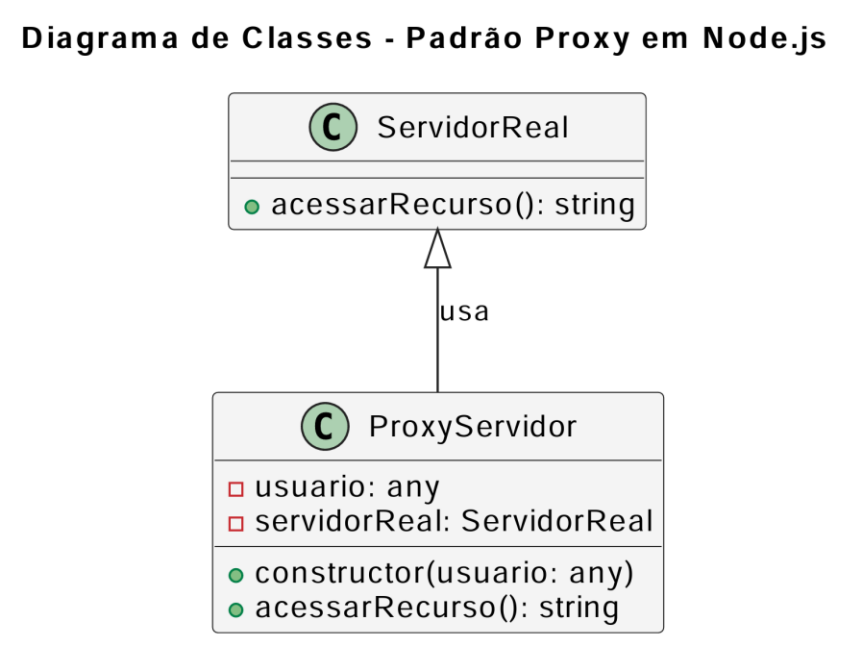
Diagrama de Classes - Padrão Proxy em Node.js
ServidorReal
acessarRecurso(): string
usa
ProxyServidor
usuario: any
servidorReal: ServidorReal
constructor(usuario: any)
acessarRecurso(): string

# Exemplo

Exemplo do Padrão Estrutural
**Proxy** em **JavaScript**

# 03

# Padrão Comportamental

Exemplos: Command, Interpreter, Mediator, Memento, State, Strategy

# Definição e Objetivo

**Padrões Comportamentais** têm como objetivo principal definir como os objetos **interagem** e se **comunicam** uns com os outros, promovendo uma **distribuição eficaz** de **responsabilidades** entre eles.

Ajudam a reduzir o **acoplamento entre os componentes**, permitindo que objetos colaborem de forma mais desacoplada e adaptável a mudanças.

# Command

## Intenção:

- **Encapsula** uma solicitação como um objeto, permitindo parametrizar clientes com diferentes **comandos**, enfileirar comandos ou desfazer operações.

## Quando usar:

- Quando quer registrar logs de ações executadas.
- Quando comandos precisam ser executados em momentos diferentes.

## Vantagens:

- Suporte a operações **undo/redo**.
- Comandos podem ser **logados** ou **agendados.**
- Facilita a extensão de **novos comandos sem modificar código existente.**

# Interpreter

## Intenção:

- Define uma representação para a gramática de uma linguagem e um interpretador para interpretar sentenças da linguagem.

## Quando usar:

- Quando você precisa interpretar expressões simples (ex: linguagens de configuração).
- Quando a gramática é pequena e bem definida.

## Vantagens:

- Boa modularidade e reutilização de regras da gramática.
- Fácil de expandir para novas regras.

# Mediator

## Intenção:

- Define um objeto que centraliza a **comunicação** entre objetos, promovendo o **baixo acoplamento.**

## Quando usar:

- Quando há muitos **objetos** interagindo entre si de forma complexa.
- Quando você quer evitar **dependências diretas** entre classes.

## Vantagens:

- Reduz o acoplamento entre componentes.
- Centraliza a lógica de comunicação.

# Memento

## Intenção:

- Permite **capturar** e **restaurar** o estado interno de um objeto sem violar seu **encapsulamento.**

## Quando usar:

- Quando você precisa de funcionalidade **undo** ou **rollback**.
- Em editores de **texto**, **jogos** ou **sistemas transacionais.**

## Vantagens:

- Preserva o **encapsulamento** do objeto.
- Permite criar **histórico de estados.**

# State

## Intenção:

- Permite alterar o comportamento de um objeto quando seu estado **muda**, **encapsulando** os estados em **classes distintas.**

## Quando usar:

- Quando o comportamento de um objeto depende do seu **estado.**
- Quando estados precisam mudar dinamicamente em tempo de execução.

## Vantagens:

- Remove estruturas de decisão **(if/switch).**
- Facilita a **manutenção** e **extensão.**

# Strategy

## Intenção:

- Define uma família de **algoritmos**, encapsula cada um e permite que sejam **intercambiáveis** em **tempo de execução.**

## Quando usar:

- Quando você tem várias **variações** de um algoritmo.
- Quando quer selecionar **algoritmos** em tempo de execução.
- Quando quer **isolar** a lógica de um **algoritmo** do resto do sistema.

## Vantagens:

- Elimina **condicionais** no código cliente.
- Facilita **testes** e **manutenção.**

# Conclusão

- Permitem que responsabilidades sejam bem **distribuídas.**
- Reduzem o **acoplamento** entre componentes.
- Aumentam a capacidade de **manutenção** e **extensão** do software.
- Desenvolvedores podem criar soluções mais **robustas**, **reutilizáveis** e **preparadas** para **mudanças.**

# UML

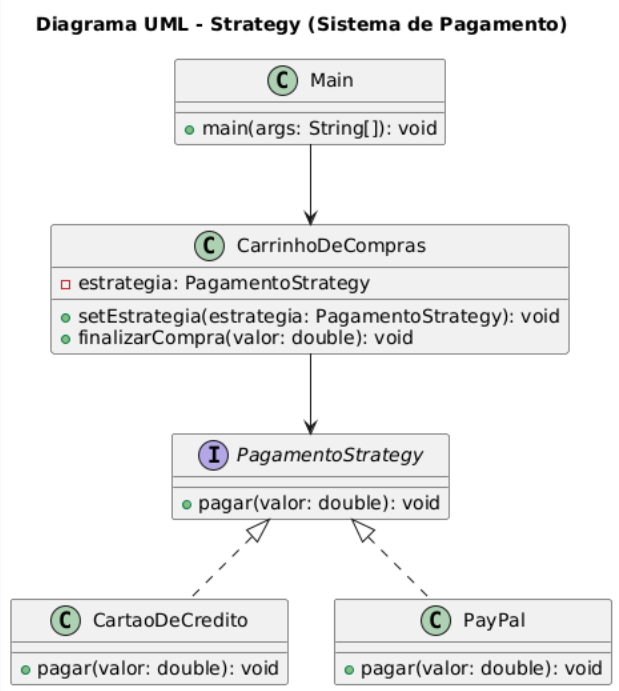
Diagrama UML - Strategy (Sistema de Pagamento)
Main
main(args: String[]): void
CarrinhoDeCompras
estrategia: PagamentoStrategy
setEstrategia(estrategia: PagamentoStrategy): void
finalizarCompra(valor: double): void
PagamentoStrategy
pagar(valor: double): void
CartaoDeCredito
pagar(valor: double): void
PayPal
pagar(valor: double): void

# Exemplo

Exemplo do Padrão Comportamental **Strategy** em **Java**