Design Patterns

Apresentação dos Padrões de Projetos apresentados por GoF (Gang of Four), Craig Larman (GRASP - *General Responsibility Assignment Software Patterns*) e Martin Fowler (*Patterns of Enterprise Application Architecture*).

Alexandre A. Santicioli

SUMÁRIO

INTRODUÇÃO

ENGENHARIA DE SOFTWARE E DESIGN PATTERNS

A **Engenharia de Software** é a disciplina que se dedica à **concepção, criação, desenvolvimento e manutenção** de sistemas de software. É um campo abrangente que envolve diversas áreas, como:

- **Análise de Requisitos**
  Compreender as necessidades dos usuários e stakeholders.
- **Projeto de Software**
  Planejar a arquitetura e as funcionalidades do sistema.
- **Implementação**
  Escrever o código-fonte do software.
- **Testes**
  Verificar se o software atende aos requisitos e funciona corretamente.
- **Manutenção**
  Corrigir bugs, aprimorar funcionalidades e adaptar o software a novas necessidades.

Nesse contexto, os ***Design Patterns*** (Padrões de *Design* ou Padrões de Projeto) desempenham um papel crucial, servindo como **soluções reutilizáveis para problemas comuns** de design de software. Eles fornecem um conjunto de **melhores práticas** para criar software modular, flexível, reutilizável e fácil de manter.

Alguns benefícios de se utilizar *Design Patterns*:

- **Evitar reinventar a roda**
  Em vez de ter que criar soluções do zero para cada problema, os *Design Patterns* oferecem soluções testadas e validadas por outros desenvolvedores.
- **Melhorar a qualidade do código**
  Os *Design Patterns* promovem código mais limpo, legível, robusto e fácil de testar.
- **Facilitar a comunicação entre desenvolvedores**
  Ao usar uma linguagem comum de *Design Patterns*, os desenvolvedores podem se comunicar de forma mais eficaz sobre o design do software.
- **Acelerar o desenvolvimento**
  A utilização de *Design Patterns* pode reduzir o tempo de desenvolvimento, pois os desenvolvedores não precisam gastar tempo criando soluções do zero.

Embora os *Design Patterns* ofereçam diversos benefícios, como a criação de software modular, flexível e reutilizável, seu uso também apresenta alguns desafios que devem ser considerados:

- **Curva de aprendizado**
  Aprender e compreender os *Design Patterns* pode ser um processo desafiador, especialmente para desenvolvedores iniciantes. É necessário investir tempo e

esforço para estudar os padrões, seus princípios e como aplicá-los em diferentes situações.

- **Escolher o padrão correto**
  Com uma variedade de padrões disponíveis, pode ser difícil identificar o padrão ideal para resolver um problema específico. A escolha incorreta do padrão pode levar a um código mais complexo e menos eficiente.
- **Excesso de abstração**
  O uso excessivo de *Design Patterns* pode tornar o código mais abstrato e difícil de entender, especialmente para outros desenvolvedores que não estejam familiarizados com os padrões utilizados.
- **Impacto no desempenho**
  Alguns *Design Patterns* podem ter um impacto no desempenho do software, especialmente se não forem utilizados de forma adequada. É importante avaliar o impacto potencial do padrão no desempenho antes de aplicá-lo.
- **Resistência à mudança**
  Implementar *Design Patterns* pode exigir alterações na estrutura do código existente, o que pode gerar resistência por parte dos desenvolvedores que não estão familiarizados com os benefícios dos padrões.

Algumas estratégias para superar os desafios podem ser utilizadas:

- **Começar com padrões básicos**
  Iniciar o aprendizado com padrões básicos e fáceis de entender, como *Singleton* e *Factory Method*, pode facilitar a assimilação de conceitos mais complexos.
- **Praticar e experimentar**
  A melhor maneira de aprender a usar *Design Patterns* é através da prática. Experimente aplicar diferentes padrões em projetos pessoais ou de pequeno porte para se familiarizar com seu funcionamento e benefícios.
- **Buscar orientação e mentoria**
  Buscar orientação de desenvolvedores experientes em *Design Patterns* pode ser crucial para aprender as melhores práticas e evitar erros comuns.
- **Utilizar ferramentas e recursos**
  Existem diversas ferramentas e recursos disponíveis para auxiliar no aprendizado e na aplicação de *Design Patterns*, como livros, artigos, cursos online e ferramentas de análise de código.
- **Comunicar e colaborar**
  É importante comunicar e colaborar com outros membros da equipe de desenvolvimento sobre a utilização de *Design Patterns* para garantir que todos estejam cientes dos benefícios e estejam preparados para as mudanças necessárias.

Design Patterns são ferramentas valiosas que podem aprimorar a qualidade do software e facilitar o desenvolvimento de sistemas robustos, flexíveis e reutilizáveis. Ao compreender os benefícios e desafios de sua adoção, os desenvolvedores podem tomar decisões sobre quando e como utilizá-los de forma eficaz em seus projetos.

"*DESIGN*" E "*PATTERNS*": ETIMOLOGIA E CONCEITO

Um bom começo para se compreender algum conceito é verificar o significado das palavras nos dicionários, sendo assim, uma breve descrição do que as palavras "*Design*" e "*Patterns*" significam em dicionários americanos é bem-vinda.

Design

- Substantivo
  Plano ou esquema para a construção de algo.
  A aparência ou estilo de algo.
- Verbo
  Planejar e criar algo.
  Dar forma ou aparência a algo.

Patterns

- Substantivo
  Modelo ou exemplo recorrente.
  Sequência de ações ou comportamentos que se repetem.
- Verbo
  Seguir um modelo ou exemplo.
  Criar algo com base em um modelo ou exemplo.

Essa base etimológica é importante para compreender a ideia das palavras, contudo, é necessário compreender tais palavras o contexto da Engenharia de Software:

- No contexto da Engenharia de Software, "*design*" se refere ao processo de planejar e estruturar a arquitetura e as funcionalidades de um sistema de software. Isso envolve desde a definição de classes e objetos até a criação de interfaces de usuário e bancos de dados.
- Na Engenharia de Software, "*patterns*" se refere a soluções reutilizáveis para problemas comuns de design de software. Esses padrões fornecem modelos ou exemplos de como estruturar e implementar código de forma eficiente e eficaz.

Ao combinar os conceitos de "*design*" e "*patterns*", podemos entender ***Design Patterns*** como **modelos ou exemplos recorrentes de soluções para problemas comuns de design de software**. Eles fornecem um guia para estruturar e implementar código de forma eficiente, robusta e reutilizável.

DESENVOLVIMENTO ÁGIL E A MAGIA DOS *DESIGN PATTERNS*

O Desenvolvimento Ágil de Software é uma metodologia que valoriza a colaboração, a adaptabilidade e a entrega incremental de valor. Nesse contexto, os *Design Patterns* se tornam ainda mais valiosos, pois podem ajudar os times a:

- **Melhorar a comunicação**
  Ao usar uma linguagem comum de *Design Patterns*, os membros do time podem se comunicar de forma mais eficaz sobre o design do software. Isso pode reduzir mal-entendidos e conflitos, e levar a um desenvolvimento mais colaborativo.

- **Acelerar o desenvolvimento**
  Design Patterns podem ser usados como um ponto de partida para o design do software, o que pode ajudar a reduzir o tempo de desenvolvimento.
- **Criar soluções flexíveis**
  Design Patterns são flexíveis e adaptáveis, o que significa que podem ser usados para criar soluções que podem ser facilmente modificadas para atender às mudanças nos requisitos.
- **Escrever código de alta qualidade**
  Design Patterns promovem **código mais limpo, legível, robusto e fácil de testar**. Isso pode levar a um software mais confiável e fácil de manter.
- **Melhorar os testes**
  Design Patterns podem ser usados para escrever testes mais eficazes, pois eles fornecem uma estrutura clara para o código, o que facilita a identificação de pontos de teste.
- **Promover a reutilização de código**
  Design Patterns são soluções reutilizáveis, o que significa que podem ser usados em diferentes projetos. Isso pode reduzir a duplicação de código e levar a um desenvolvimento mais eficiente.

REFERÊNCIAS EM DESIGN PATTERNS

No mundo dos *Design Patterns*, há duas obras que se tornaram referência, pois consolidaram padrões muito utilizados:

- *"Design Patterns: Elements of Reusable Object-Oriented Software"* - de *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* – conhecidos como *GoF (Gang of Four)*. Eles identificaram 23 padrões divididos em três categorias: criacionais, estruturais e comportamentais.
- *"Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design"* - de *Craig Larman*. Enfatiza que os padrões de design não são receitas prontas, mas sim guias para a resolução de problemas de design de forma eficaz e reutilizável. Nessa obra é apresentado os padrões GRASP (*General Responsibility Assignment Software Patterns*).

Apesar de terem sido publicados na década de 1990, os padrões GoF e GRASP continuam sendo relevantes e valiosos para os desenvolvedores de software hoje em dia. As razões para sua importância duradoura incluem:

- Foco em princípios fundamentais
  Os padrões GoF e GRASP se concentram em princípios fundamentais de design de software que são aplicáveis a uma ampla variedade de linguagens de programação e tecnologias.
- Soluções testadas e validadas
  Os padrões GoF e GRASP foram testados e validados por anos de uso na indústria de software.
- Comunidade ativa
  Existe uma comunidade ativa de desenvolvedores que usam e discutem os padrões

GoF e GRASP, o que significa que há uma riqueza de conhecimento e recursos disponíveis para quem deseja aprender mais sobre eles.

Há outras obras que apresentam *Design Patterns* importantes, além de haver padrões que não estão em obras, mas em artigos acadêmicos ou ainda em blogs de desenvolvedores consagrados, como é o caso blog de Martin Fowler, autor do livro *Patterns of Enterprise Application Architecture* - considerado uma referência em padrões arquiteturais.

OS DESIGN PATTERNS MAIS UTILIZADOS

Embora a popularidade dos padrões de design possa variar ao longo do tempo, alguns padrões se destacam como sendo frequentemente utilizados:

- Padrões Criacionais: *Singleton*, *Factory Method*, *Abstract Factory*, *Builder*.
- Padrões Estruturais: *Adapter*, *Bridge*, *Composite*, *Decorator*, *Façade*, *Proxy*.
- Padrões Comportamentais: *Observer*, *Strategy*, *State*, *Template Method*, *Chain of Responsibility*, *Command*, *Memento*, *Iterator*.

PADRÕES GOF (GANG OF FOUR)

A obra *"Design Patterns: Elements of Reusable Object-Oriented Software"* divide seus padrões em Criacionais, Criacionais e Comportamentais.

PADRÕES CRIACIONAIS

Os Padrões Criacionais se concentram em encapsular a lógica de **criação de objetos**, separando-a da lógica de uso dos objetos em si.

Os Padrões Criacionais se dividem em quatro categorias principais, cada uma focada em um aspecto específico da criação de objetos.

SINGLETON

Os padrões de *Singleton* garantem que uma classe tenha apenas uma instância em todo o sistema. Essa instância única é acessível através de um ponto de acesso global, fornecendo um controle centralizado sobre a criação e o uso do objeto.

Exemplo: classes que gerenciam as configurações globais do aplicativo, como preferências do usuário e informações de conexão com o banco de dados.

> "Um *Singleton* é uma classe da qual apenas uma instância existe em um sistema. É um *design pattern* que garante que você tenha um ponto de acesso global para essa instância." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

FACTORY METHOD

Os padrões de *Factory Method* encapsulam a lógica de criação de objetos em um único método, permitindo que subclasses definam quais tipos de objetos devem ser criados. Essa abstração facilita a troca de tipos de objetos sem afetar o código que os utiliza.

Exemplo: Uma classe abstrata que define um método de fábrica para criar diferentes tipos de veículos, como carros, motos e caminhões.

> "O *Factory Method* permite que você defina uma interface para criar objetos, mas deixe que subclasses decidam quais classes instanciar. Isso pode ser usado para centralizar a lógica de criação de objetos em um ponto e para permitir que subclasses personalizem a criação de objetos." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

ABSTRACT FACTORY

Os padrões de *Abstract Factory* fornecem uma interface para criar uma família de produtos relacionados, sem especificar as classes concretas dos produtos. Isso permite que você alterne famílias de produtos sem modificar o código que os utiliza.

Exemplo: Uma interface abstrata que define métodos para criar diferentes tipos de componentes de interface do usuário, como botões, menus e caixas de diálogo.

> "O *Abstract Factory* fornece uma interface para criar uma família de produtos relacionados, mas deixa que subclasses decidam quais classes instanciar. Isso pode ser usado para centralizar a lógica de criação de produtos em um ponto e para permitir que subclasses personalizem a criação de produtos." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

BUILDER

Os padrões de *Builder* separam a construção de um objeto complexo em etapas independentes, permitindo que o objeto seja construído de forma variada e personalizada. Isso é útil para objetos que possuem muitas propriedades ou configurações.

Exemplo: Uma classe que constrói um relatório detalhado de um pedido, permitindo que diferentes seções do relatório sejam construídas independentemente.

> "O *Builder* separa a construção de um objeto complexo em etapas independentes. Isso permite que você construa diferentes representações do mesmo objeto usando a mesma interface de construção." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

PROTOTYPE

O padrão de *Prototype* permite criar objetos copiando objetos existentes, evitando a necessidade de recriar o objeto do zero. Isso pode ser útil para objetos que são caros de criar ou que possuem dados que precisam ser preservados.

Exemplo: Uma classe que representa um documento de texto, permitindo que você crie documentos copiando documentos existentes e modificando-os posteriormente.

> "O *Prototype* permite que se crie objetos copiando objetos existentes. Isso pode ser útil para objetos que são caros de criar ou que possuem dados que precisam ser preservados." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

PADRÕES ESTRUTURAIS

Os Padrões Estruturais se concentram em organizar classes e objetos em relações significativas, definindo padrões de comunicação e colaboração entre eles.

Os Padrões Estruturais se dividem em sete categorias principais, cada uma focada em um aspecto específico da estruturação de software:

ADAPTER

O padrão de *Adapter* converte a interface de uma classe em outra interface compatível com o cliente, permitindo que objetos incompatíveis trabalhem juntos. Isso é útil quando você precisa integrar componentes de diferentes sistemas ou bibliotecas.

Exemplo: Um adaptador que converte a interface de um sistema de pagamento legado em uma interface compatível com um novo sistema de e-commerce.

> "O *Adapter* permite que objetos com interfaces incompatíveis trabalhem juntos. Isso é feito convertendo a interface de um objeto em outra interface que o cliente espera." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

BRIDGE

O padrão de *Bridge* separa a interface de uma classe de sua implementação, permitindo que ambas variem independentemente. Isso possibilita a modificação da implementação de uma classe sem afetar o código que a utiliza.

Exemplo: Uma classe abstrata que define uma interface para desenhar formas, com subclasses concretas que implementam diferentes algoritmos de desenho (como linha, retângulo e círculo).

> "O Bridge separa a interface de uma classe de sua implementação. Isso permite que você altere a implementação de uma classe sem afetar o código que a utiliza." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

COMPOSITE

O padrão de *Composite* compõe objetos em uma estrutura hierárquica, permitindo que sejam tratados como um único objeto. Isso facilita o gerenciamento de grupos de objetos e a implementação de comportamentos recursivos.

Exemplo: Uma classe que representa uma estrutura de diretórios, com objetos que representam arquivos e subdiretórios.

> "O *Composite* compõe objetos em uma estrutura hierárquica, permitindo que sejam tratados como um único objeto. Isso facilita o gerenciamento de grupos de objetos e a implementação de comportamentos recursivos." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

DECORATOR

O padrão de *Decorator* adiciona responsabilidades a um objeto dinamicamente, encapsulando-o em um novo objeto. Isso permite a personalização de objetos sem alterar sua classe original.

Exemplo: Uma classe que representa uma xícara de café, com classes decoradoras que adicionam leite, açúcar e chocolate à xícara.

> "O *Decorator* adiciona responsabilidades a um objeto dinamicamente, encapsulando-o em um novo objeto. Isso permite que você personalize objetos sem alterar sua classe original." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

FAÇADE

O padrão de *Façade* fornece uma interface única para um subsistema complexo, escondendo sua complexidade interna. Isso facilita o uso do subsistema para clientes que não precisam saber sobre seus detalhes internos.

> **Curiosidade**
>
> A palavra "*façade*" vem do francês e significa "frente" ou "fachada".
>
> A pronúncia correta é `faˈsɑːd`, não "*faceide*".
>
> Mesmo o termo em inglês "*facade*" não possui essa pronúncia, pois se pronuncia `fəˈsɑːd`.

Exemplo: Uma classe que fornece uma interface única para um sistema de gerenciamento de contas bancárias, permitindo que os clientes depositem, saquem e transfiram dinheiro sem precisar saber sobre os detalhes internos do sistema.

> O padrão de *Façade* fornece uma interface única para um subsistema complexo, escondendo sua complexidade interna. Isso facilita o uso do subsistema para clientes que não precisam saber sobre seus detalhes internos." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

FLYWEIGHT

O padrão de *Flyweight* usa objetos compartilhados para representar partes do estado de um objeto, reduzindo o uso de memória. Isso é útil para objetos que possuem muitos dados redundantes.

Exemplo: Uma classe que representa um caractere em uma fonte, com objetos compartilhados para representar diferentes estilos de fonte (negrito, itálico, sublinhado).

> "O *Flyweight* usa objetos compartilhados para representar partes do estado de um objeto, reduzindo o uso de memória. Isso é útil para objetos que possuem muitos dados redundantes." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

PROXY

O padrão de *Proxy* fornece um substituto para outro objeto, controlando o acesso a esse objeto e adicionando funcionalidades adicionais. Isso é útil para proteger recursos, controlar o acesso remoto a objetos ou implementar cache.

Exemplo: Um proxy que controla o acesso a um arquivo, verificando se o usuário tem permissão para ler ou gravar no arquivo.

> "O Proxy fornece um substituto para outro objeto, controlando o acesso a esse objeto e adicionando funcionalidades adicionais. Isso é útil para proteger recursos, controlar o acesso remoto a objetos ou implementar cache." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

PADRÕES COMPORTAMENTAIS

Os Padrões Comportamentais assumem o papel de maestros experientes, orquestrando a comunicação e a colaboração entre objetos de forma elegante e eficiente. Esses padrões fornecem soluções testadas e validadas para problemas comuns de interação e comunicação entre objetos, permitindo que os desenvolvedores criem sistemas mais dinâmicos, adaptáveis e fáceis de entender.

Os Padrões Comportamentais se concentram em definir como os objetos se comunicam e interagem entre si, definindo responsabilidades, sequências de ações e protocolos de comunicação.

Os Padrões Comportamentais se dividem em onze categorias principais, cada uma focada em um aspecto específico da comunicação e interação entre objetos.

CHAIN OF RESPONSIBILITY

O padrão de *Chain of Responsibility* permite que vários objetos lidem com uma solicitação em sequência, passando a solicitação de um para outro até que um deles a resolva. Isso é útil para distribuir responsabilidades entre diferentes objetos e evitar a necessidade de um objeto centralizado tomar todas as decisões.

Exemplo: Um sistema de aprovação de pedidos que passa a solicitação de aprovação por uma sequência de gerentes, cada um com diferentes níveis de autoridade.

> "O *Chain of Responsibility* permite que você passe uma solicitação por uma cadeia de objetos. Cada objeto na cadeia tem a chance de lidar com a solicitação. Se um objeto não puder lidar com a solicitação, ele a passa para o próximo objeto na cadeia." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

COMMAND

O padrão de *Command* encapsula uma solicitação como um objeto, permitindo que você enfileire, agende, registre e invoque solicitações. Isso é útil para desacoplar o solicitante do receptor e para implementar logs e desfazer/refazer funcionalidades.

Exemplo: Um sistema de edição de texto que armazena comandos de edição (como inserir, excluir e formatar) em uma fila para serem executados posteriormente.

> "O Command encapsula uma solicitação como um objeto. Isso permite que você desacople o solicitante do receptor e para implementar logs e desfazer/refazer funcionalidades." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

CQRS

O padrão *Command* e a arquitetura CQRS (*Command Query Responsibility Segregation*) se unem como peças de um quebra-cabeça para criar sistemas robustos, escaláveis e flexíveis. Essa combinação poderosa oferece diversos benefícios, desde a simplificação do código até a otimização do desempenho.

SIMPLIFICANDO O CÓDIGO COM COMANDOS

O padrão *Command* encapsula uma solicitação como um objeto, permitindo seu enfileiramento, agende, registre e invoque com facilidade. Imagine um sistema de edição de texto: cada comando de edição (inserir, excluir, formatar) é um objeto autônomo, facilitando o gerenciamento e a execução das ações do usuário.

CQRS: SEPARANDO RESPONSABILIDADES PARA MAIOR CLAREZA:

A arquitetura CQRS separa as operações de leitura e atualização de um armazenamento de dados, definindo comandos (*Commands*) para modificar o estado do sistema e consultas (*Queries*) para recuperar informações sem alterá-lo. Essa separação de responsabilidades torna o código mais claro, organizado e fácil de manter.

A UNIÃO DOS TITÃS: COMMAND E CQRS EM SINTONIA:

O padrão *Command* se encaixa perfeitamente na arquitetura CQRS. Os comandos, encapsulando as solicitações do usuário, podem ser facilmente roteados para os componentes adequados do sistema, seja para atualizar o estado do banco de dados ou para recuperar informações.

BENEFÍCIOS DA DUPLA DINÂMICA:

- Código mais simples e organizado
  O padrão *Command* encapsula as solicitações em objetos autônomos, enquanto o CQRS separa as responsabilidades de leitura e atualização, resultando em um código mais fácil de entender e modificar.
- Maior escalabilidade
  O CQRS permite que os componentes de leitura e atualização sejam escalados de forma independente, otimizando o desempenho do sistema sob carga elevada.
- Flexibilidade aprimorada
  A separação de responsabilidades do CQRS facilita a adição de novas

funcionalidades e a modificação do comportamento do sistema sem afetar outros componentes.

- Testes mais fáceis
  O padrão *Command* e o CQRS facilitam a criação de testes unitários e de integração, garantindo a confiabilidade do sistema.

EXEMPLO PRÁTICO: GERENCIANDO PEDIDOS EM UM E-COMMERCE:

Imagine um sistema de e-commerce que utiliza CQRS para gerenciar pedidos. Os comandos, como "Criar Pedido", "Adicionar Item" e "Confirmar Pedido", podem ser facilmente roteados para os componentes adequados do sistema. As consultas, como "Obter Detalhes do Pedido" e "Verificar Status do Pedido", podem ser otimizadas para acesso rápido às informações.

INTERPRETER

O padrão de *Interpreter* interpreta uma linguagem específica, definindo uma gramática para a linguagem e um interpretador que analisa e executa sentenças na linguagem. Isso é útil para implementar linguagens de *scripting*, expressões matemáticas e regras de validação.

Exemplo: Um interpretador de linguagem de expressões matemáticas que avalia expressões como `2 + 3 * 4` e retorna o resultado.

> "O *Interpreter* define uma gramática para uma linguagem específica e um interpretador que analisa e executa sentenças na linguagem. Isso é útil para implementar linguagens de *scripting*, expressões matemáticas e regras de validação." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

ITERATOR

O padrão de *Iterator* permite percorrer uma coleção de elementos de forma sequencial, sem expor sua representação interna. Isso é útil para implementar loops e iterar sobre diferentes tipos de coleções, como listas, arrays e árvores.

Exemplo: Um iterador que percorre uma lista de produtos e imprime o nome de cada produto.

> "O *Iterator* permite que você percorra uma coleção de elementos de forma sequencial, sem expor sua representação interna. Isso é útil para implementar loops e iterar sobre diferentes tipos de coleções, como listas, arrays e árvores." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

MEDIATOR

O padrão de *Mediator* define uma comunicação indireta entre objetos, centralizando a comunicação em um único objeto. Isso facilita a comunicação entre objetos que não estão diretamente relacionados e ajuda a reduzir o acoplamento entre os objetos.

Exemplo: Um mediador que controla a comunicação entre os jogadores em um jogo multiplayer, garantindo que as regras do jogo sejam seguidas e que os jogadores possam interagir entre si.

> "O *Mediator* define uma comunicação indireta entre objetos, centralizando a comunicação em um único objeto. Isso facilita a comunicação entre objetos que não estão diretamente relacionados e ajuda a reduzir o acoplamento entre os objetos." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

MEMENTO

O padrão de *Memento* captura e armazena o estado interno de um objeto, permitindo que você restaure o objeto a um estado anterior. Isso é útil para implementar funcionalidades de desfazer/refazer, checkpoints e logs de estado.

Exemplo: Um editor de texto que usa o padrão *Memento* para armazenar o estado do documento em diferentes pontos no tempo, permitindo que o usuário desfaça ou refaça as alterações feitas no documento.

> "O *Memento* captura e armazena o estado interno de um objeto, permitindo que você restaure o objeto a um estado anterior. Isso é útil para implementar funcionalidades de desfazer/refazer, checkpoints e logs de estado." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

OBSERVER

O padrão de *Observer* define uma dependência um-para-muitos entre objetos, onde um objeto (sujeito) notifica seus dependentes (observadores) sobre mudanças em seu estado. Isso é útil para implementar sistemas de eventos, notificações e atualizações em tempo real.

Exemplo: Um sistema de monitoramento de temperatura que notifica observadores (como interfaces de usuário e aplicativos) quando a temperatura excede um limite predefinido.

> "O *Observer* define uma dependência um-para-muitos entre objetos, onde um objeto (sujeito) notifica seus dependentes (observadores) sobre mudanças em seu estado. Isso é útil para implementar sistemas de eventos, notificações e atualizações em tempo real." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

Sistemas Críticos

No universo dos sistemas críticos, onde falhas podem ter consequências desastrosas, o Padrão *Observer* assume o papel de maestro experiente, orquestrando o monitoramento e a comunicação entre componentes de forma confiável e robusta.

DESVENDANDO OS SISTEMAS CRÍTICOS

Sistemas críticos são aqueles em que falhas podem levar a perdas financeiras significativas, danos à propriedade, lesões corporais ou até mesmo à perda de vidas.

Exemplos incluem sistemas de controle de tráfego aéreo, sistemas de geração de energia, sistemas bancários e sistemas médicos.

A IMPORTÂNCIA DO MONITORAMENTO EM SISTEMAS CRÍTICOS

O monitoramento contínuo e preciso do estado de saúde desses sistemas é crucial para garantir sua operação segura e confiável. O Padrão *Observer* fornece uma estrutura robusta para implementar o monitoramento em sistemas críticos, permitindo que:

- **Eventos sejam detectados e notificados de forma eficiente**
  O padrão *Observer* permite que componentes do sistema se registrem como observadores de eventos específicos. Quando um evento ocorre, ele é notificado a todos os observadores registrados, permitindo que eles tomem as medidas cabíveis.

- **Falhas sejam identificadas e isoladas rapidamente**
  Ao monitorar eventos em tempo real, o padrão *Observer* facilita a identificação de falhas em seus estágios iniciais, permitindo que medidas corretivas sejam tomadas antes que as falhas causem danos maiores.

- **A disponibilidade do sistema seja otimizada**
  O monitoramento contínuo do estado do sistema permite identificar gargalos e problemas de desempenho antes que afetem a disponibilidade do sistema, permitindo medidas preventivas para garantir a operação contínua.

EXEMPLOS PRÁTICOS DO PADRÃO *OBSERVER* EM SISTEMAS CRÍTICOS

Imagine um sistema de monitoramento de temperatura em uma usina nuclear. Sensores de temperatura espalhados pela usina enviam eventos de temperatura para um sistema central que atua como observador. Se um sensor detectar um aumento repentino na temperatura, ele notificará o sistema central, que acionará alarmes e iniciará os procedimentos de segurança.

BENEFÍCIOS DO PADRÃO *OBSERVER* EM SISTEMAS CRÍTICOS

- **Flexibilidade e Escalabilidade**
  O padrão *Observer* permite que você adicione ou remova observadores facilmente, sem afetar o código central. Isso torna o sistema mais flexível e escalável, permitindo que você adapte o monitoramento às necessidades específicas do sistema.

- **Desacoplamento**
  O padrão *Observer* promove o desacoplamento entre os componentes do sistema, pois os observadores não precisam saber nada sobre a implementação do componente que gera os eventos. Isso torna o código mais modular e fácil de manter.

- **Reusabilidade**
  O padrão *Observer* é um padrão geral que pode ser aplicado em diversos contextos, desde sistemas críticos até aplicações web. Isso torna o código mais reutilizável e facilita o desenvolvimento de sistemas robustos e confiáveis.

PONTOS DE ATENÇÃO

- A implementação do Padrão *Observer* em sistemas críticos deve levar em consideração requisitos de desempenho, confiabilidade e segurança específicos do sistema.

- É importante escolher ferramentas e bibliotecas adequadas para implementar o padrão *Observer* de forma eficiente e segura.

- A documentação clara e concisa do código é crucial para facilitar a manutenção e o aprimoramento do sistema ao longo do tempo.

CONCLUSÃO

O Padrão *Observer* é uma ferramenta essencial para implementar o monitoramento em sistemas críticos, proporcionando uma arquitetura robusta, flexível e escalável. Ao utilizar esse padrão, os desenvolvedores podem garantir que seus sistemas operem de forma segura e confiável, minimizando o risco de falhas e seus impactos negativos.

Internet das Coisas (IoT) em Automação Industrial

A Internet das Coisas (IoT) está revolucionando a automação industrial, conectando máquinas, sensores e dispositivos em um vasto ecossistema digital. Nesse contexto, o padrão *Observer* emerge como uma ferramenta crucial para gerenciar a comunicação e a interação eficiente entre esses elementos.

BENEFÍCIOS DO PADRÃO *OBSERVER* NA AUTOMAÇÃO INDUSTRIAL COM IOT

- **Escalabilidade Aprimorada**
  O padrão *Observer* permite que vários observadores sejam conectados a um único sujeito, sem comprometer o desempenho do sistema. Isso é fundamental para lidar com a grande quantidade de dispositivos presentes em ambientes IoT.

- **Desacoplamento Robusto**
  O padrão *Observer* promove o desacoplamento entre o sujeito e seus observadores, tornando o sistema mais flexível e adaptável. Essa característica facilita a adição, remoção ou modificação de observadores sem afetar o funcionamento do sujeito.

- **Comunicação Eficiente**
  O padrão *Observer* garante que os observadores sejam notificados apenas sobre as mudanças relevantes em seu estado, evitando a necessidade de processar informações irrelevantes. Isso otimiza o uso da largura de banda e reduz o consumo de recursos.

EXEMPLOS PRÁTICOS NA AUTOMAÇÃO INDUSTRIAL COM IOT

- **Monitoramento de sensores**
  Um sensor de temperatura em uma máquina industrial pode ser o sujeito, notificando observadores (como um sistema de controle ou um painel de monitoramento) sobre mudanças na temperatura.

- **Gerenciamento de alarmes**
  Um sistema de alarme pode ser o sujeito, notificando observadores (como operadores ou técnicos de manutenção) sobre eventos críticos, como falhas de equipamentos ou violações de segurança.

- **Automação de processos**
  Um sistema de automação pode ser o sujeito, notificando observadores (como robôs ou válvulas) sobre etapas específicas do processo que requerem ações automatizadas.

CONSIDERAÇÕES ADICIONAIS

- A escolha da implementação do padrão *Observer* (como *publish/subscribe* ou *pull/push*) depende das características específicas do sistema IoT.

- A segurança e a confiabilidade da comunicação entre o sujeito e os observadores são aspectos cruciais em sistemas de automação industrial.

CONCLUSÃO

O padrão *Observer* se destaca como uma ferramenta essencial para a construção de sistemas de automação industrial robustos, escaláveis e eficientes na era da IoT. A adoção dessa estrutura garante uma comunicação fluida e confiável entre dispositivos conectados, otimizando o desempenho e a confiabilidade dos processos industriais.

STATE

O padrão de *State* permite a alteração do comportamento de um objeto em resposta a mudanças internas ou externas. Isso é útil para implementar objetos que passam por diferentes estados, como um semáforo que muda de verde para amarelo para vermelho.

Exemplo: Um objeto que representa um pedido em um sistema de e-commerce, com estados como "pendente", "processando", "enviado" e "entregue".

> "O *State* permite que um objeto altere seu comportamento em resposta a mudanças internas ou externas. Isso é útil para implementar objetos que passam por diferentes estados, como um semáforo que muda de verde para amarelo para vermelho." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

STRATEGY

O padrão de *Strategy* permite que um algoritmo varie sem alterar o código cliente, definindo famílias de algoritmos, classes de algoritmos e objetos de algoritmos. Isso é

útil para implementar algoritmos intercambiáveis e para tornar o código mais flexível e adaptável.

Exemplo: Um sistema de compressão de arquivos que suporta diferentes algoritmos de compressão, como **ZIP**, **RAR** e **7-Zip**.

> "O *Strategy* permite que você varie um algoritmo sem alterar o código cliente. Isso é feito definindo famílias de algoritmos, classes de algoritmos e objetos de algoritmos." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

TEMPLATE METHOD

O padrão de *Template Method* define um esqueleto de um algoritmo em uma classe abstrata, permitindo que subclasses definam os passos específicos do algoritmo. Isso é útil para implementar algoritmos com passos comuns e variáveis, como a ordenação de dados.

Exemplo: Uma classe abstrata que define um algoritmo de ordenação de dados, com subclasses que implementam diferentes algoritmos de ordenação (como ordenação por bolha, ordenação por seleção e ordenação rápida).

> "O *Template Method* define um esqueleto de um algoritmo em uma classe abstrata, permitindo que subclasses definam os passos específicos do algoritmo. Isso é útil para implementar algoritmos com passos comuns e variáveis, como a ordenação de dados - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

VISITOR

O padrão de *Visitor* permite adicionar novas operações a uma classe sem modificar sua classe, definindo uma classe visitante que visita as classes do objeto e executa as operações desejadas. Isso é útil para implementar novas funcionalidades sem afetar o código existente.

Exemplo: Um sistema de análise de código que usa um visitante para analisar diferentes tipos de elementos de código, como classes, métodos e variáveis.

> "O *Visitor* permite que você adicione novas operações a uma classe sem modificar sua classe. Isso é feito definindo uma classe visitante que visita as classes do objeto e executa as operações desejadas." - *Erich Gamma*, *Richard Helm*, *Ralph Johnson* e *John Vlissides* (*Design Patterns: Elements of Reusable Object-Oriented Software*)

PADRÕES GRASP

Os Padrões GRASP (*General Responsibility Assignment Software Patterns*) direcionam a responsabilidades e a coesão em classes e objetos. Essa jornada em direção a sistemas robustos, flexíveis e fáceis de manter se inicia com a compreensão profunda da essência dos princípios GRASP, presentes na obra seminal de *Craig Larman*.

Os Padrões GRASP se dividem em nove categorias principais, cada uma focada em um aspecto específico da atribuição de responsabilidades:

CRIADOR (*CREATOR*)

Define a responsabilidade de criar objetos de uma classe específica, ou seja, define um objeto responsável pela criação de outros objetos.

Exemplo: Uma classe `ContaBancaria` que possui um método `criarConta()` responsável por criar objetos `ContaBancaria`.

> "O padrão Criador define a responsabilidade de criar objetos de uma classe específica." - *Craig Larman* (*Applying UML and Patterns*)

CONTROLADOR (*CONTROLLER*)

Define a responsabilidade de coordenar o fluxo de eventos em um sistema, ou seja, define um objeto responsável por gerenciar a interação entre outros objetos.

Exemplo: Uma classe `ControladorTelaLogin` que controla a interação do usuário com a tela de login, gerenciando eventos como digitação de login e senha e validação de acesso.

> "O padrão Controlador define a responsabilidade de coordenar o fluxo de eventos em um sistema." - *Craig Larman* (*Applying UML and Patterns*)

FABRICAÇÃO PURA (*PURE FABRICATION*)

Define a criação de objetos artificiais que não possuem representação direta no mundo real, ou seja, cria objetos que não possuem correspondência direta no mundo real.

Exemplo: Uma classe `GeradorDeId` que gera identificadores únicos para objetos, como números de série ou códigos de barras.

> "O padrão Fabricação Pura define a criação de objetos artificiais que não possuem representação direta no mundo real." - *Craig Larman* (*Applying UML and Patterns*)

ESPECIALISTA NA INFORMAÇÃO (*INFORMATION EXPERT*)

Define a responsabilidade de armazenar e gerenciar informações específicas, ou seja, atribui responsabilidades a objetos que possuem o conhecimento necessário para executá-las.

Exemplo: Uma classe `Cliente` que armazena informações sobre um cliente, como nome, endereço, dados de contato e histórico de compras.

> "O padrão Especialista na Informação define a responsabilidade de armazenar e gerenciar informações específicas." - *Craig Larman* (*Applying UML and Patterns*)

ALTA COESÃO (*HIGH COHESION*)

Define a concentração de responsabilidades relacionadas em uma única classe, ou seja, agrupa responsabilidades relacionadas em uma única classe.

Exemplo: Uma classe **Calculadora** que encapsula todas as funcionalidades relacionadas a cálculos matemáticos, como adição, subtração, multiplicação e divisão.

> "O padrão Coesão Alta define a concentração de responsabilidades relacionadas em uma única classe." - *Craig Larman* (*Applying UML and Patterns*)

BAIXO ACOPLAMENTO (*LOW COUPLING*)

Define a redução da dependência entre classes e módulos do sistema, ou seja, minimiza a dependência entre classes, tornando-as mais fáceis de modificar e testar.

Exemplo: A utilização de interfaces e abstrações para definir contratos entre classes, permitindo que diferentes classes implementem as mesmas interfaces sem afetar o restante do sistema.

> "O padrão Acoplamento Baixo define a minimização da dependência entre classes e módulos do sistema." - *Craig Larman* (*Applying UML and Patterns*)

INDIREÇÃO (*INDIRECTION*)

Define a utilização de uma camada intermediária para desacoplar elementos do sistema, ou seja, substitui ligações diretas entre objetos por indireções, como interfaces ou abstrações.

Exemplo: Uma classe **InterfaceUsuario** que define uma interface abstrata para interação com o usuário, permitindo a utilização de diferentes tipos de interfaces (como interface gráfica ou interface de linha de comando) sem afetar o restante do sistema.

> "O padrão Indireção define a utilização de uma camada intermediária para desacoplar elementos do sistema." - *Craig Larman* (*Applying UML and Patterns*)

POLIMORFISMO (*POLYMORPHISM*)

Define a capacidade de um objeto assumir diferentes formas ou comportamentos em diferentes contextos, ou seja, permite que objetos de diferentes classes respondam à mesma mensagem de maneiras diferentes.

Exemplo: A utilização de herança para criar classes que herdam funcionalidades de classes base, permitindo que diferentes classes respondam à mesma mensagem de formas distintas.

> "O padrão Polimorfismo define a capacidade de um objeto assumir diferentes formas ou comportamentos em diferentes contextos." - *Craig Larman* (*Applying UML and Patterns*)

VARIAÇÕES PROTEGIDAS (*PROTECTED VARIATIONS*)

Define a proteção de elementos do sistema contra variações em outros elementos. Isso é alcançado encapsulando o foco de instabilidade com uma interface e utilizando polimorfismo para criar diversas implementações dessa interface.

Resumindo: permite a modificação do comportamento de um objeto sem alterar sua interface.

Exemplo: Uma interface **Impressora** que define um método **imprimir()**. Diferentes classes concretas, como **ImpressoraLaser** e **ImpressoraJatoDeTinta**, implementam a interface **Impressora**, encapsulando a variação específica de cada tipo de impressora.

> "O padrão Variações Protegidas define a proteção de elementos do sistema contra variações em outros elementos." - *Craig Larman* (*Applying UML and Patterns*)

PADRÕES DE APLICATIVOS EMPRESARIAIS (MARTIN FOWLER)

O livro "*Patterns of Enterprise Application Architecture*" de Martin Fowler apresenta um rico conjunto de padrões para a construção de softwares empresariais complexos.

Antes de apresentar os padrões, é importante ressaltar as diferenças entre esses padrões e os padrões GoF (Gang of Four) e GRASP (*General Responsibility Assignment Software Patterns*).

- **GoF**
  Padrões de design focados em solucionar problemas recorrentes de design orientado a objetos (OO).
- **GRASP**
  Princípios para a atribuição de responsabilidades em classes e objetos, promovendo coesão e reduzindo acoplamento.
- **Fowler**
  Padrões de arquitetura para projetar a estrutura geral de aplicações empresariais, incluindo camadas, comunicação entre componentes, persistência de dados e apresentação.

PADRÕES DE LÓGICA DE DOMÍNIO (*DOMAIN LOGIC PATTERNS*)

TRANSACTION SCRIPT (ROTEIRO DE TRANSAÇÃO)

Define uma sequência de operações atômicas que modificam o estado do sistema.

Exemplo: Um roteiro de transação para processar um pedido de compra, envolvendo validação de dados, atualização de estoque e registro do pedido.

> "Um roteiro de transação define uma sequência de operações atômicas que modificam o estado do sistema." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

DOMAIN MODEL (MODELO DE DOMÍNIO)

Representa os conceitos centrais do negócio e suas relações.

Exemplo: Um modelo de domínio que inclui classes como Cliente, Produto e Pedido, representando as entidades do negócio e seus relacionamentos.

> "Um modelo de domínio representa os conceitos centrais do negócio e suas relações." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

TABLE MODULE (MÓDULO TABELA)

Encapsula o acesso a uma tabela específica do banco de dados. (Fowler desencoraja o uso excessivo desse padrão)

Exemplo: Um módulo de tabela que encapsula o acesso à tabela Clientes do banco de dados.

> "Um módulo de tabela encapsula o acesso a uma tabela específica do banco de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

SERVICE LAYER (CAMADA DE SERVIÇO)

Define uma camada intermediária que encapsula a lógica de negócio e provê interfaces para outros componentes do sistema.

Exemplo: Uma camada de serviço que

PADRÕES DE ARQUITETURA DE FONTE DE DADOS (DATA SOURCE ARCHITECTURAL PATTERNS)

TABLE DATA GATEWAY (GATEWAY DE DADOS DE TABELA)

Fornece acesso a dados de uma tabela específica do banco de dados utilizando SQL.

Exemplo: Um gateway de dados de tabela que permite realizar operações CRUD (*Create*, *Read*, *Update*, *Delete*) em uma tabela de clientes.

> "Um gateway de dados de tabela fornece acesso a dados de uma tabela específica do banco de dados utilizando SQL." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

ROW DATA GATEWAY (GATEWAY DE DADOS DE LINHA)

Fornece acesso a linhas individuais de uma tabela do banco de dados. (Fowler desencoraja o uso excessivo desse padrão)

Exemplo: Um gateway de dados de linha que permite recuperar e atualizar dados de uma linha específica na tabela de clientes.

> "Um gateway de dados de linha fornece acesso a linhas individuais de uma tabela do banco de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

ACTIVE RECORD (REGISTRO ATIVO)

Padrão onde as classes do modelo de domínio possuem métodos para persistir seu estado no banco de dados.

Exemplo: Uma classe Cliente que possui métodos para salvar e recuperar seus dados do banco de dados.

> "Um registro ativo é um padrão onde as classes do modelo de domínio possuem métodos para persistir seu estado no banco de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

DATA MAPPER (MAPEADOR DE DADOS)

Mapeia objetos do modelo de domínio para linhas em um banco de dados relacional.

Exemplo: Um mapeador de dados que converte um objeto Cliente em uma linha na tabela de clientes do banco de dados.

> "Um mapeador de dados mapeia objetos do modelo de domínio para linhas em um banco de dados relacional." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PADRÕES COMPORTAMENTAIS OBJETO-RELACIONAIS (*OBJECT-RELATIONAL BEHAVIORAL PATTERNS*)

UNIT OF WORK (UNIDADE DE TRABALHO)

Gerencia um conjunto de operações de persistência como uma única transação.

Exemplo: Uma unidade de trabalho que coordena a inserção de um novo pedido e a atualização do estoque, garantindo a consistência dos dados.

> "Uma unidade de trabalho gerencia um conjunto de operações de persistência como uma única transação." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

IDENTITY MAP (MAPA DE IDENTIDADE)

Mantém um cache de objetos recuperados do banco de dados para evitar acessos redundantes.

Exemplo: Um mapa de identidade que armazena objetos Cliente recuperados do banco de dados, evitando consultas repetidas para o mesmo cliente.

> "Um mapa de identidade mantém um cache de objetos recuperados do banco de dados para evitar acessos redundantes." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

LAZY LOAD (CARREGAMENTO *LAZY*)

Carrega dados relacionados a um objeto somente quando necessário.

Exemplo: Um objeto Pedido que carrega os itens associados somente quando o método `getItens()` é chamado.

> "*Lazy load* é um padrão que carrega dados relacionados a um objeto somente quando necessário." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PADRÕES ESTRUTURAIS OBJETO-RELACIONAIS (*OBJECT-RELATIONAL STRUCTURAL PATTERNS*)

IDENTITY FIELD (CAMPO DE IDENTIDADE)

Utiliza um campo único na tabela do banco de dados para identificar objetos.

Exemplo: Um campo `id_cliente` na tabela Clientes utilizado como identificador único para objetos Cliente.

> "Um campo de identidade utiliza um campo único na tabela do banco de dados para identificar objetos." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

FOREIGN KEY MAPPING (MAPEAMENTO DE CHAVE ESTRANGEIRA)

Utiliza uma chave estrangeira na tabela do banco de dados para representar relacionamentos entre entidades.

Exemplo: Um campo `id_pedido` na tabela `ItensPedido` como chave estrangeira que referencia o campo `id_pedido` na tabela Pedidos, representando o relacionamento entre pedidos e itens de pedido.

> "Um mapeamento de chave estrangeira utiliza uma chave estrangeira na tabela do banco de dados para representar relacionamentos entre entidades." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

ASSOCIATION TABLE MAPPING (MAPEAMENTO DE TABELA DE ASSOCIAÇÃO)

Utiliza uma tabela separada para representar relacionamentos muitos-para-muitos entre entidades.

Exemplo: Uma tabela `ClientesProjetos` que relaciona clientes com projetos, representando o relacionamento muitos-para-muitos entre clientes e projetos em que um cliente pode participar de vários projetos e um projeto pode ter vários clientes.

> "Um mapeamento de tabela de associação utiliza uma tabela separada para representar relacionamentos muitos-para-muitos entre entidades." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

DEPENDENT MAPPING (MAPEAMENTO DEPENDENTE)

Um objeto é mapeado para uma tabela do banco de dados que depende de outro objeto pai.

Exemplo: Um objeto `ItemPedido` que é mapeado para uma tabela `ItensPedido` dependente do objeto Pedido pai.

> "Um mapeamento dependente é um objeto mapeado para uma tabela do banco de dados que depende de outro objeto pai." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

EMBEDDED VALUE (VALOR EMBUTIDO)

Armazena dados primitivos como atributos de um objeto do modelo de domínio.

Exemplo: Um objeto Cliente que possui atributos como nome e endereco armazenados diretamente no objeto, em vez de serem mapeados para uma tabela separada.

> "Um valor embutido é um padrão onde dados primitivos são armazenados como atributos de um objeto do modelo de domínio." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

SERIALIZED LOB (LOB SERIALIZADO)

Armazena dados binários grandes (LOB) em um campo separado na tabela do banco de dados.

Exemplo: Um campo foto na tabela Clientes utilizado para armazenar a foto do cliente como um LOB serializado.

> "Um LOB serializado é um padrão onde dados binários grandes (LOB) são armazenados em um campo separado na tabela do banco de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

SINGLE TABLE INHERITANCE (HERANÇA DE TABELA ÚNICA)

Utiliza uma única tabela do banco de dados para armazenar classes herdadas em um modelo de herança.

Exemplo: Uma tabela Pessoas que armazena dados de classes herdadas como `Cliente` e `Funcionario`, utilizando mecanismos para diferenciar os tipos de objetos.

> "Herança de tabela única é um padrão onde utiliza uma única tabela do banco de dados para armazenar classes herdadas em um modelo de herança." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

CLASS TABLE INHERITANCE (HERANÇA DE TABELA POR CLASSE)

Utiliza tabelas separadas para cada classe em um modelo de herança.

Exemplo: Tabelas separadas `Clientes` e `Funcionarios` para armazenar dados de classes herdadas `Cliente` e `Funcionario`, respectivamente.

> "Herança de tabela por classe é um padrão onde utiliza tabelas separadas para cada classe em um modelo de herança." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

CONCRETE TABLE INHERITANCE (HERANÇA DE TABELA CONCRETA)

Similar à herança de tabela por classe, porém as tabelas filhas possuem chaves primárias separadas.

Exemplo: Tabelas separadas `Clientes` e `Funcionarios` com chaves primárias distintas, para armazenar dados de classes herdadas `Cliente` e `Funcionario`, respectivamente.

INHERITANCE MAPPERS (MAPEADORES DE HERANÇA)

Classes responsáveis por mapear hierarquias de classes do modelo de domínio para tabelas do banco de dados, considerando diferentes estratégias de herança.

Exemplo: Um mapeador de herança que implementa a estratégia de herança de tabela por classe, mapeando a classe Cliente para a tabela Clientes e a classe `Funcionario` para a tabela `Funcionarios`.

> "Mapeadores de herança são classes responsáveis por mapear hierarquias de classes do modelo de domínio para tabelas do banco de dados, considerando diferentes estratégias de herança." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PADRÕES DE MAPEAMENTO DE METADADOS OBJETO-RELACIONAIS (*OBJECT-RELATIONAL METADATA MAPPING PATTERNS*)

METADATA MAPPING (MAPEAMENTO DE METADADOS)

Utiliza metadados para definir o mapeamento entre objetos do modelo de domínio e tabelas do banco de dados.

Exemplo: Anotações em classes do modelo de domínio que definem o nome da tabela correspondente e o mapeamento de atributos para colunas.

> "Mapeamento de metadados utiliza metadados para definir o mapeamento entre objetos do modelo de domínio e tabelas do banco de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

QUERY OBJECT (OBJETO DE CONSULTA)

Encapsula uma consulta específica ao banco de dados.

Exemplo: Um objeto `ConsultaClientesPorNome` que encapsula a lógica para buscar clientes pelo nome.

> "Objeto de consulta é um padrão que encapsula uma consulta específica ao banco de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

REPOSITORY (REPOSITÓRIO)

Fornece uma interface para acessar e persistir dados de um tipo específico de objeto.

Exemplo: Um repositório `ClienteRepository` que provê métodos para buscar, salvar e deletar objetos Cliente.

> "Repositório é um padrão que fornece uma interface para acessar e persistir dados de um tipo específico de objeto." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PADRÕES DE APRESENTAÇÃO WEB (*WEB PRESENTATION PATTERNS*)

MODEL VIEW CONTROLLER (MVC) (MODELO-VISÃO-CONTROLADOR)

Separa a lógica de negócio (modelo), a apresentação (visão) e o tratamento de requisições (controlador).

Exemplo: Uma aplicação web que utiliza MVC, onde o modelo representa os dados do cliente, a visão gera a página HTML e o controlador processa os formulários enviados pelo usuário.

> "MVC é um padrão que separa a lógica de negócio (modelo), a apresentação (visão) e o tratamento de requisições (controlador)." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PAGE CONTROLLER (CONTROLADOR DE PÁGINA)

Controla o fluxo de uma única página web. (Considerado um padrão legado por Fowler)

Exemplo: Um controlador de página que processa os eventos de uma página de cadastro de cliente.

> "Controlador de página é um padrão que controla o fluxo de uma única página web." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

FRONT CONTROLLER (CONTROLADOR CENTRAL)

Um único controlador que despacha requisições para outros componentes do sistema.

Exemplo: Um *front controller* que recebe todas as requisições HTTP da aplicação web e direciona para os controladores apropriados.

> "*Front controller* é um padrão onde um único controlador despacha requisições para outros componentes do sistema." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

TEMPLATE VIEW (VISÃO DE TEMPLATE)

Utiliza templates para gerar conteúdo dinâmico da página web.

Exemplo: Um template HTML que define a estrutura básica da página e utiliza marcadores para inserir dados dinâmicos recuperados do modelo.

> "Visão de template é um padrão que utiliza templates para gerar conteúdo dinâmico da página web." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

TRANSFORM VIEW (VISÃO DE TRANSFORMAÇÃO)

Separa a apresentação da lógica de negócio utilizando um mecanismo de transformação de dados. (Menos comum atualmente)

Exemplo: Uma visão de transformação que recebe dados do modelo e os transforma em XML antes de serem apresentados na página web.

> "Visão de transformação é um padrão que separa a apresentação da lógica de negócio utilizando um mecanismo de transforma

PADRÕES DE DISTRIBUIÇÃO (*DISTRIBUTION PATTERNS*)

REMOTE FACADE (FACHADA REMOTA)

Fornece uma interface remota para acessar a funcionalidade de negócio em um sistema distribuído.

Exemplo: Uma fachada remota que expõe métodos para serviços de negócio como autenticação de usuários, acessíveis por aplicações clientes remotas.

> "Fachada remota fornece uma interface remota para acessar a funcionalidade de negócio em um sistema distribuído." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

DATA TRANSFER OBJECT (DTO) (OBJETO DE TRANSFERÊNCIA DE DADOS)

Objeto simples utilizado para transferir dados entre camadas ou sistemas distribuídos.

Exemplo: Um DTO `PedidoDTO` contendo apenas os dados necessários para representar um pedido em um sistema de processamento de pedidos, utilizado para transferência entre a camada de apresentação e a camada de serviço.

> "DTO é um objeto simples utilizado para transferir dados entre camadas ou sistemas distribuídos." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PADRÕES DE CONCORRÊNCIA OFFLINE (*OFFLINE CONCURRENCY PATTERNS*)

OPTIMISTIC OFFLINE LOCK (BLOQUEIO OTIMISTA OFFLINE)

Permite a edição de dados offline com validação no momento da sincronização.

Exemplo: Um aplicativo móvel que permite a edição de dados de clientes offline, validando as alterações e verificando por conflitos no momento da sincronização com o servidor.

> "Bloqueio otimista offline permite a edição de dados offline com validação no momento da sincronização." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PESSIMISTIC OFFLINE LOCK (BLOQUEIO PESSIMISTA OFFLINE)

Bloqueia dados para edição offline, evitando conflitos de concorrência.

Exemplo: Um aplicativo móvel que bloqueia dados de clientes para edição offline, impedindo que outros usuários editem os mesmos dados simultaneamente.

> "Bloqueio pessimista offline bloqueia dados para edição offline, evitando conflitos de concorrência." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

COARSE-GRAINED LOCK (BLOQUEIO GRANULAR)

Bloqueia grupos de dados relacionados para garantir consistência.

Exemplo: Um aplicativo de edição colaborativa que bloqueia um documento inteiro para edição, evitando conflitos de edição em partes do documento.

> "Bloqueio granular bloqueia grupos de dados relacionados para garantir consistência." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

IMPLICIT LOCK (BLOQUEIO IMPLÍCITO)

O sistema gerencia automaticamente o bloqueio de dados baseado em operações de acesso.

Exemplo: Um banco de dados relacional que implementa bloqueios automáticos em tabelas para garantir a consistência dos dados durante as transações.

> "Bloqueio implícito é onde o sistema gerencia automaticamente o bloqueio de dados baseado em operações de acesso." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PADRÕES DE ESTADO DA SESSÃO (*SESSION STATE PATTERNS*)

CLIENT SESSION STATE (ESTADO DA SESSÃO DO CLIENTE)

Armazena o estado da sessão no dispositivo do cliente (cookies, local storage).

Exemplo: Um carrinho de compras em uma loja virtual que utiliza cookies para armazenar os produtos selecionados pelo usuário durante a sessão de navegação.

> "Estado da sessão do cliente armazena o estado da sessão no dispositivo do cliente (cookies, local storage)." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

SERVER SESSION STATE (ESTADO DA SESSÃO DO SERVIDOR)

Armazena o estado da sessão no servidor da aplicação.

Exemplo: Um servidor web que utiliza sessões para armazenar informações do usuário logado (nome, preferências) durante a sessão.

> "Estado da sessão do servidor armazena o estado da sessão no servidor da aplicação." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

DATABASE SESSION STATE (ESTADO DA SESSÃO DO BANCO DE DADOS)

Armazena o estado da sessão em um banco de dados. (Menos comum devido ao custo de acesso)

Exemplo: Um sistema de gerenciamento de conteúdo que utiliza o banco de dados para armazenar o estado da sessão do usuário, permitindo que a sessão seja acessível de diferentes servidores web.

> "Estado da sessão do banco de dados armazena o estado da sessão em um banco de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PADRÕES BÁSICOS (*BASE PATTERNS*)

Esta categoria apresenta padrões genéricos e reutilizáveis para diversas finalidades

GATEWAY (PORTAL)

Encapsula o acesso a um recurso externo, como outro sistema ou subsistema.

Exemplo: Um gateway de pagamento que encapsula a comunicação com um serviço de pagamento externo para processar transações financeiras.

> "Gateway encapsula o acesso a um recurso externo, como outro sistema ou subsistema." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

MAPPER (MAPEADOR)

Define uma conversão entre diferentes representações de dados.

Exemplos: Já vimos o *Data Mapper* como um exemplo. Outro exemplo é um mapeador que converte objetos do modelo de domínio em representações JSON para APIs RESTful.

> "*Mapper* define uma conversão entre diferentes representações de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

LAYER SUPERTYPE (SUPERTIPO DE CAMADA)

Define uma classe base para todas as classes em uma camada específica da arquitetura. (Fowler desencoraja o uso excessivo desse padrão)

Exemplo: Uma classe abstrata `ComponenteServico` que define a interface base para todas as classes de serviço da aplicação.

> "Supertipo de camada define uma classe base para todas as classes em uma camada específica da arquitetura." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

SEPARATED INTERFACE (INTERFACE SEPARADA)

Separa a definição de uma interface de sua implementação.

Exemplo: Uma interface `ClienteService` que define os métodos para acessar dados de clientes, separada de sua implementação concreta.

> "Interface separada separa a definição de uma interface de sua implementação." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

REGISTRY (REGISTRO)

Fornece um mecanismo para localizar objetos pelo nome.

Exemplo: Um registro de serviços que armazena referências a serviços da aplicação e permite a recuperação pelo nome do serviço.

> "Registro fornece um mecanismo para localizar objetos pelo nome." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

VALUE OBJECT (OBJETO DE VALOR)

Objeto imutável que encapsula dados complexos.

Exemplo: Um objeto Endereco que encapsula dados como rua, cidade e CEP, sendo imutável para garantir a consistência dos dados.

> "Objeto de valor é um objeto imutável que encapsula dados complexos." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

MONEY (DINHEIRO)

Representa um valor monetário com precisão e regras de negócio associadas.

Exemplo: Uma classe Moeda que representa valores monetários, podendo incluir regras de arredondamento e conversão de moeda.

> "Dinheiro representa um valor monetário com precisão e regras de negócio associadas." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

SPECIAL CASE (CASO ESPECIAL)

Solução específica para um problema incomum que não se encaixa em um padrão genérico.

Exemplo: Um algoritmo de validação complexo para um tipo específico de dado que não se encaixa em um padrão de validação genérico.

> "Caso especial é uma solução específica para um problema incomum que não se encaixa em um padrão genérico." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

PLUGIN (PLUGIN)

Extensão que adiciona funcionalidades a uma aplicação hospedeira.

Exemplo: Um plugin de estatísticas para um sistema de gerenciamento de conteúdo que adiciona funcionalidades para geração de relatórios de acesso.

> "Plugin é uma extensão que adiciona funcionalidades a uma aplicação hospedeira." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

SERVICE STUB (SUBSTITUTO DE SERVIÇO)

Simulação de um serviço externo para fins de teste.

Exemplo: Um *stub* de serviço de pagamento que simula o comportamento do serviço real para testar a funcionalidade de processamento de pagamentos.

> "Substituto de serviço é uma simulação de um serviço externo para fins de teste." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

RECORD SET (CONJUNTO DE REGISTROS)

Representa um conjunto de resultados de uma consulta a um banco de dados. (Menos comum com o advento de frameworks de acesso a dados mais abstratos)

Exemplo: Um objeto `ResultSet` que encapsula os resultados de uma consulta SQL ao banco de dados, contendo linhas e colunas de dados.

> "Conjunto de registros representa um conjunto de resultados de uma consulta a um banco de dados." - *Martin Fowler* (*Patterns of Enterprise Application Architecture*)

O uso do padrão Record Set se tornou menos comum com o advento de frameworks de acesso a dados mais abstratos, como ORM (*Object-Relational Mapping*), que encapsulam a lógica de acesso ao banco de dados e oferecem uma interface mais amigável para desenvolvedores.

Frameworks como Hibernate e JPA no Java, e `SQLAlchemy` no Python, são exemplos de ferramentas que facilitam o acesso a dados e diminuem a necessidade de usar o padrão Record Set diretamente.

Embora o padrão Record Set tenha sido importante no passado, frameworks modernos de acesso a dados oferecem soluções mais convenientes e abstratas para lidar com resultados de consultas em aplicações web.

OUTROS DESIGN PATTERNS

Com o objetivo de fornecer uma resposta completa e informativa, elaborarei uma descrição detalhada de 20 padrões de design de outros autores, além dos renomados Martin Fowler, Gang of Four (GoF) e Craig Larman.

DDD (DOMAIN-DRIVEN DESIGN)

O DDD propõe a organização do código em camadas que representam os diferentes níveis de abstração do domínio do problema. As camadas principais são:

- Camada de Domínio
  Encapsula os conceitos e regras de negócio do problema, utilizando um modelo de domínio rico e expressivo.
- Camada de Aplicação
  Implementa casos de uso e serviços de aplicação, orquestrando as regras de negócio da camada de domínio.
- Camada de Infraestrutura
  Fornece acesso a recursos externos, como bancos de dados, serviços web e sistemas legados, abstraindo os detalhes de implementação.

CQRS (COMMAND QUERY RESPONSIBILITY SEGREGATION)

O CQRS separa as responsabilidades de leitura (consultas) e gravação (comandos) em diferentes modelos, otimizando cada um para sua função específica.

- Modelo de Consulta
  Projetado para consultas frequentes e eficientes, com alta disponibilidade e escalabilidade.
- Modelo de Comando
  Otimizado para gravação de dados, transações robustas e consistência forte.

EVENT SOURCING (SOURCING DE EVENTOS):

Armazena o estado de um sistema como uma sequência de eventos imutáveis, permitindo reprodutibilidade, auditoria e escalabilidade.

- Eventos
  Representam mudanças no estado do sistema, encapsulando informações sobre o que aconteceu e quando.
- Projeção
  Gera uma visão atual do estado do sistema a partir dos eventos armazenados.

SAGA

Coordena uma série de transações locais como uma única transação distribuída, mesmo na ausência de suporte nativo a transações distribuídas.

- Saga Manager
  Gerencia o ciclo de vida da saga, garantindo a consistência e a execução atômica das transações locais.
- Compensação
  Permite reverter transações locais em caso de falha na saga.

PIPELINE

Estrutura o processamento de dados em uma sequência de etapas, permitindo a composição de diferentes transformações e filtros.

- Etapas
  Cada etapa realiza uma operação específica nos dados, podendo ser combinadas para formar um pipeline completo.
- Fluxo de Dados
  Os dados fluem através das etapas do pipeline, sendo transformados e processados em cada etapa.

PUBLISH-SUBSCRIBE (PUB/SUB)

Permite que vários assinantes recebam notificações de eventos publicados por um ou mais publicadores.

- Tópicos
  Agrupam eventos relacionados, permitindo que assinantes se interessem por tópicos específicos.

- Assinantes
  Implementam a lógica de processamento para os eventos recebidos, podendo ser acionados simultaneamente.

NULL OBJECT (OBJETO NULO)

Fornece um objeto substituto especial que representa a ausência de um valor real.

- Classe *Null Object*
  Classe que implementa a interface esperada, mas não realiza nenhuma operação real.
- Clientes
  Utilizam o objeto nulo como se fosse um objeto real, evitando verificações `null` frequentes.