

DESCRIÇÃO

Padrões de projeto de software e visão geral dos padrões GRASP, SOLID e GoF.

PROPÓSITO

Compreender o conceito de padrões de projeto e ser capaz de identificar oportunidades para a aplicação dos principais padrões em projetos de sistemas são habilidades importantes para a sua formação, pois elas possibilitam a criação de programas bem estruturados por meio do uso de soluções reconhecidamente bem-sucedidas para problemas recorrentes no desenvolvimento de software.

PREPARAÇÃO

Este conteúdo não requer nenhum pré-requisito operacional obrigatório. Porém, antes de iniciar, é recomendado ter instalado em seu computador um programa que lhe permita elaborar modelos sob a forma de diagramas da UML (Linguagem Unificada de Modelagem). Nossa sugestão inicial é o Astah Free Student License, usado nos exemplos deste estudo, e, para isso, será necessário lançar mão do seu e-mail institucional para ativar a licença. Preencha os dados do formulário no site do software, envie e aguarde a liberação de sua licença em seu e-mail institucional. Ao receber a licença, siga as instruções do e-mail e instale o produto em seu computador. Os arquivos-fonte dos diagramas Astah UML mostrados neste conteúdo podem ser baixados aqui.

Sugestões de links adicionais de programas livres para modelagem de sistemas em UML (UML Tools) podem ser encontradas em buscas na internet.

OBJETIVOS

MÓDULO 1

Descrever os conceitos gerais de padrões de projeto, seus elementos e suas características

MÓDULO 2

Reconhecer o propósito dos padrões GRASP e as situações nas quais eles podem ser aplicados

MÓDULO 3

Descrever as características dos princípios SOLID

MÓDULO 4

Reconhecer o propósito dos principais padrões GoF e as situações nas quais eles podem ser aplicados

INTRODUÇÃO

Desenvolvedores pouco experientes frequentemente têm dificuldade em estruturar um software de forma que ele satisfaça os requisitos de qualidade demandados.

Por que é importante estruturar um software adequadamente? Não basta programar as suas instruções de forma que ele realize corretamente as funções demandadas pelo cliente?

Realizar as funções corretamente é apenas um dos requisitos de qualidade de um sistema. Um sistema bem-sucedido passa por diversos ciclos de evolução ao longo da sua existência, motivados por mudanças tanto no negócio como na tecnologia. Para complicar o cenário, as pessoas que participam dos ciclos iniciais de desenvolvimento podem ir para outros projetos ou outras empresas, e novos desenvolvedores precisarão dar continuidade à evolução do sistema. Portanto, não basta que ele funcione corretamente; deve ser estruturado de modo que sua manutenção seja possível a custos e prazos aceitáveis.

Neste estudo, você aprenderá como os conceitos de padrões de projeto, padrões GRASP, GoF e princípios SOLID podem ser empregados para a produção de sistemas bem estruturados.

MÓDULO 1

🕒 Descrever os conceitos gerais de padrões de projeto, seus elementos e suas características

Você já ouviu falar em padrões de projeto (***design patterns***)? Sabe por que eles foram criados?

DESIGN PATTERNS

Design patterns ou padrões de projeto são soluções gerais para problemas recorrentes durante o desenvolvimento de um software.

Neste módulo, você vai aprender o que são padrões de projeto, como eles são definidos e entender as vantagens e desvantagens da sua utilização.

MOTIVAÇÃO E ORIGEM DOS PADRÕES DE PROJETO

Imagine que você foi alocado em um novo trabalho, ficando responsável por evoluir um sistema desenvolvido por outras pessoas ao longo de anos.

Você não se sentiria mais confortável se o software tivesse sido projetado com estruturas de solução que você já conhecesse?

Ao aprendermos a programar em uma linguagem orientada a objetos, por exemplo, somos capazes de definir classes, atributos, operações, interfaces e subclasses com essa linguagem. Esse conhecimento nos permite responder a perguntas operacionais como, por exemplo:

“Como eu escrevo a definição de uma classe nesta linguagem de programação?”.

Entretanto, quando temos que estruturar um sistema, a pergunta passa a ser bem mais complexa:

“Quais classes, interfaces, relacionamentos, atributos e operações devemos definir para resolvermos este problema de forma adequada?”.

Ao desenvolvermos um sistema, nos deparamos inúmeras vezes com essa pergunta. É comum um desenvolvedor iniciante se sentir perdido diante da diversidade de respostas possíveis, sendo que algumas podem facilitar, enquanto outras podem dificultar bastante as futuras evoluções do sistema.

Que tipo de conhecimento, então, um desenvolvedor experiente utiliza para estruturar um sistema?

Fazendo uma analogia com o jogo de xadrez, um desenvolvedor inexperiente guarda semelhanças com um jogador iniciante, que sabe colocar as peças no tabuleiro, conhece os movimentos permitidos para cada peça e domina alguns princípios elementares do jogo, como a importância relativa das peças, e o centro do tabuleiro.



Um desenvolvedor inexperiente conhece as construções básicas de programação (ex.: sequência, decisão, iteração, classes, objetos, atributos, operações) e alguns princípios elementares de estruturação de um sistema, tais como dividi-lo em módulos e evitar construir módulos longos e complexos.

O que diferencia os grandes enxadristas dos iniciantes?

Os grandes enxadristas estudaram ou experimentaram inúmeras situações de jogo e acumularam experiência sobre as consequências que determinado lance pode gerar no desenrolar da partida.

Quanto maior for o arsenal de situações conhecidas por um jogador, maiores as chances que ele terá de fazer uma boa jogada em pouco tempo.

É isso mesmo: os jogadores de xadrez também têm um prazo para cumprir. Existem campeonatos que definem um tempo máximo de duas horas para que cada jogador realize seus primeiros quarenta lances.

Um jogador iniciante não aprende apenas jogando e vivenciando essas situações. Ele pode estudar inúmeras situações catalogadas pelas quais outros jogadores já passaram.

COMENTÁRIO

Existem catálogos de situações e estratégias para o **início**, para o **meio** e para a **finalização** de um jogo de xadrez.

As estratégias para o início de jogo são conhecidas como **aberturas**. Cada abertura tem um nome e corresponde a um conjunto de jogadas que geram consequências positivas e, eventualmente, negativas para cada lado (peças brancas ou pretas). Uma dessas aberturas é o nome de uma das séries originais mais vistas na história da Netflix: ***O gambito da rainha***.

E o que diferencia os desenvolvedores experientes dos iniciantes?

RESPOSTA

Desenvolvedores experientes acumularam conhecimento sobre as consequências de resolver um problema aplicando determinada estrutura de solução, isto é, dividindo a solução em um conjunto específico de módulos e estabelecendo uma estrutura particular de interação entre eles. Essa experiência acumulada permite-lhes reusar, no presente, soluções que funcionaram bem no passado em problemas similares, produzindo sistemas flexíveis, elegantes e em menor tempo.

Muitos problemas em estruturação de software são recorrentes, então que tal registrá-los com as respectivas soluções para que, de forma análoga aos livros de abertura do xadrez, possamos compartilhar esse conhecimento com desenvolvedores que estejam encarando um problema análogo pela primeira vez?

Foi o que pensaram quatro engenheiros de software (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides) em 1994, quando publicaram o livro ***Design patterns: elements of reusable object-oriented software***, descrevendo 23 padrões que eles utilizaram no desenvolvimento de diversos sistemas ao longo de suas carreiras. Os autores ficaram mais conhecidos como a “**gangue dos quatro**” (tradução de ***gang of four***), e os padrões publicados são conhecidos como os “**padrões GoF**”. Desde então, diversas publicações surgiram relatando novos padrões, críticas a padrões existentes, padrões específicos para determinadas linguagens e paradigmas de programação, padrões arquiteturais e até mesmo antipadrões – construções que tipicamente trazem consequências negativas para a estrutura de um sistema.

O QUE É UM PADRÃO DE PROJETO

Um padrão de projeto (***design pattern***) descreve um problema recorrente e a estrutura fundamental da sua solução definida por módulos e pelas comunicações entre eles.

Em uma estrutura de solução orientada a objetos, a solução é descrita por classes, interfaces e mecanismos de colaboração entre objetos.

Porém, o conceito de padrões de projeto é mais abrangente, uma vez que existem padrões voltados para outros paradigmas de programação, como, por exemplo, **programação funcional** e **programação reativa**.

Um padrão de projeto apresenta quatro elementos fundamentais:

NOME

O nome possibilita a comunicação entre os desenvolvedores em um nível mais abstrato. Em uma discussão de projeto, em vez de descrevermos todos os elementos participantes da solução para um problema, a forma de comunicação entre eles e as consequências do seu uso, basta mencionarmos o nome do padrão apropriado.

Um desenvolvedor experiente pode recomendar algo como: “use o Observer para resolver esse problema!”, e isso será suficiente, pois conhecendo o padrão Observer, você já saberá como estruturar a solução. Portanto, os nomes dos padrões foram um vocabulário compartilhado pelos desenvolvedores, para agilizar a discussão de soluções de projeto.

PROBLEMA

Um padrão deve descrever as situações nas quais a sua utilização é apropriada, explicando o problema e o contexto. Os problemas podem ser específicos, como, por exemplo: “como podemos representar diferentes algoritmos com o mesmo propósito na forma de classes?” ou “como podemos instanciar uma classe específica, dentre várias similares, sem criar uma dependência rígida com a implementação escolhida?”.

Este elemento também pode descrever e discutir soluções inadequadas ou pouco flexíveis para o contexto apresentado, de forma que, se você estiver pensando em uma solução similar às descritas, o seu problema passará a ser: “ok, estou vendo que a estrutura que estava pensando em aplicar não é adequada. Como devo, então, estruturar a solução?”.

SOLUÇÃO

O padrão deve descrever os elementos recomendados para a implementação, suas responsabilidades, seus relacionamentos e colaborações. A solução é uma descrição abstrata que pode ser replicada em diferentes situações e sistemas. O desenvolvedor deve traduzir

essa descrição abstrata em uma implementação específica no problema particular que ele estiver resolvendo. Note que a solução é uma ideia que precisa ser construída e implementada pelo desenvolvedor. Um catálogo de padrões de projeto é bem diferente de uma biblioteca de código, pois um padrão de projeto não tem o propósito de promover reutilização de código, mas sim de conhecimento, *know-how*.

CONSEQUÊNCIAS

Um padrão precisa discutir os custos e benefícios da solução proposta. Uma solução pode dar maior flexibilidade ao projeto, ao mesmo tempo em que pode trazer maior complexidade ou até mesmo resultar em problemas de performance. Antes de utilizarmos um padrão, devemos sempre avaliar os custos e benefícios em relação ao problema particular que estamos querendo resolver.

VANTAGENS E DESVANTAGENS DO USO DE PADRÕES DE PROJETO

Por que você deve conhecer padrões de projeto? Podemos enumerar alguns motivos:

Padrões facilitam o desenvolvimento, pois permitem a reutilização de soluções bem-sucedidas em problemas similares. Um padrão não pode ser simplesmente uma ideia; ao contrário, só pode ser considerado como tal se passar pela regra dos três: ele deve ter sido utilizado em pelo menos três diferentes situações ou aplicações.

Padrões possibilitam maior produtividade ao desenvolvimento, uma vez que não desperdiçamos tempo pensando, fazendo e refinando soluções até encontrarmos a mais adequada. Padrões nos fornecem um atalho para uma boa solução.

Padrões reforçam práticas de reutilização de código com estruturas que acomodam mudanças por meio do uso de mecanismos como delegação, composição e outras técnicas que não sejam baseadas em estruturas de herança.

Padrões fornecem uma linguagem comum para os desenvolvedores, agilizando a troca de ideias e experiências.

E quais são as desvantagens ou os pontos de atenção na utilização de padrões de projeto?

Desde que o conceito foi apresentado na década de 1990, surgiu uma grande quantidade de padrões. Portanto, a curva de aprendizado pode ser grande.

Decidir se um padrão pode ser empregado em um problema específico nem sempre é uma tarefa fácil e requer alguma experiência.

Às vezes é necessário adaptar ou mudar alguma característica da solução proposta por um padrão para tornar sua utilização adequada a um problema específico, o que também requer experiência.

É comum um iniciante achar que os padrões devem estar por toda a implementação e acabar fazendo uso inadequado deles.

Não existe consenso sobre a qualidade de todos os padrões. Singleton, por exemplo, é um padrão GoF que gera grande controvérsia, sendo considerado um antipadrão por muitos.

PRINCIPAIS PADRÕES EM LINHAS GERAIS

Os 23 padrões de projeto GoF são classificados em três grandes categorias:

Criação	<p>Os padrões de projeto desta categoria têm como objetivo tornar a implementação independente da forma com que os objetos são criados, compostos ou representados.</p> <p>Os cinco padrões GoF desta categoria são: Abstract Factory, Builder, Factory Method, Prototype e Singleton.</p>
Estrutura	<p>Os padrões de projeto desta categoria tratam de formas de combinar classes e objetos para formar estruturas maiores.</p> <p>Os sete padrões GoF desta categoria são: Adapter, Bridge, Composite, Decorator, Facade, Flyweight e Proxy.</p>
Comportamento	<p>Os padrões de projeto da categoria comportamental tratam de algoritmos, da distribuição de responsabilidades pelos objetos e da comunicação entre eles.</p> <p>Os onze padrões GoF desta categoria são: Chain of Responsibility, Command, Interpreter, Iterator, Mediator,</p>

	Memento, Observer, State, Strategy, Template Method e Visitor.
--	---



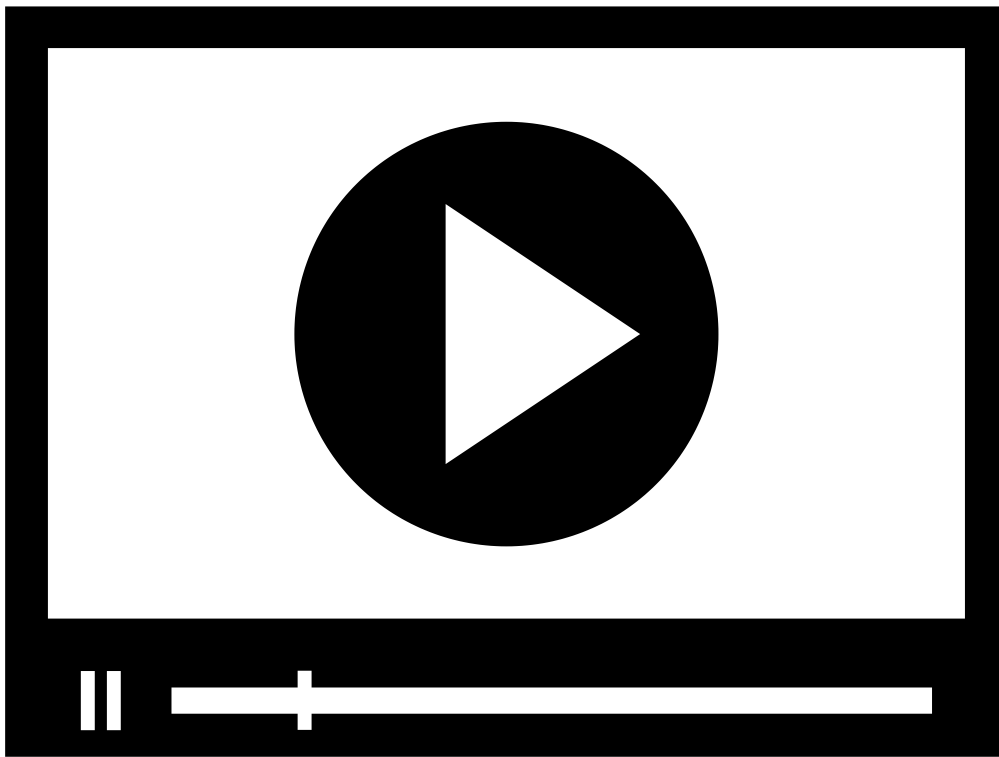
Atenção! Para visualização completa da tabela utilize a rolagem horizontal

SAIBA MAIS

Existem inúmeros padrões publicados para diferentes problemas: arquitetura e projeto detalhado de software, padrões específicos para uma linguagem de programação, testes, gerência de projeto, gerência de configuração, entre outros. Você não precisa estudar todos os padrões disponíveis, mas é importante saber que eles existem, onde estão descritos e conhecer os problemas que eles resolvem. Assim, no dia em que tiver que resolver um problema similar, você poderá aproveitar uma solução utilizada com sucesso por outras pessoas.

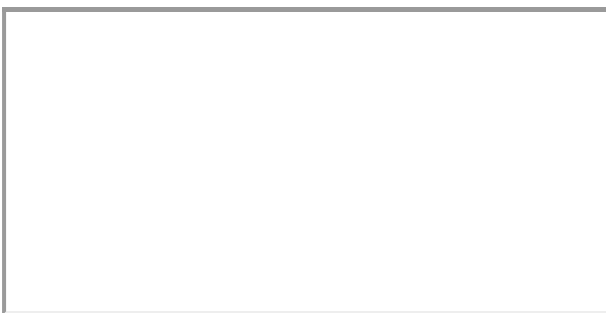
Como achá-los? Experimente fazer uma busca na internet com termos gerais, tais como “pattern books”, “patterns catalog”, ou mais específicos, como “java design patterns”, “cloud architecture patterns”, “node js patterns”.

Você encontrará dicas mais específicas na seção Explore +.



CONCEITOS GERAIS DE PADRÕES DE PROJETO

No vídeo a seguir, conheça as principais características, vantagens e problemas de um padrão de projeto.



VERIFICANDO O APRENDIZADO

MÓDULO 2

- ⦿ Reconhecer o propósito dos padrões GRASP e as situações nas quais eles podem ser aplicados

PADRÕES GRASP

GRASP é o acrônimo para o termo em inglês ***General Responsibility Assignment Software Patterns***, termo definido por Craig Larman no livro intitulado ***Applying UML and patterns***, que define padrões gerais para a atribuição de responsabilidades em software.

Os padrões GoF tratam de problemas específicos em projeto de software.



Os padrões GRASP podem ser vistos como princípios gerais de projeto de software orientado a objetos aplicáveis a diversos problemas específicos.

Neste módulo, você vai aprender seis padrões GRASP:

Especialista

Criador

Baixo acoplamento

Alta coesão

Controlador

Polimorfismo

ESPECIALISTA

Problema:

Quando estamos elaborando a solução técnica de um projeto utilizando objetos, a nossa principal tarefa consiste em definir as responsabilidades de cada classe e as interações necessárias entre os objetos dessas classes, para cumprir as funcionalidades esperadas com uma estrutura fácil de entender, manter e estender.

Solução:

Este padrão trata do princípio geral de atribuição de responsabilidades aos objetos de um sistema: atribua a responsabilidade ao especialista, isto é, ao módulo que traga o conhecimento necessário para realizá-la.

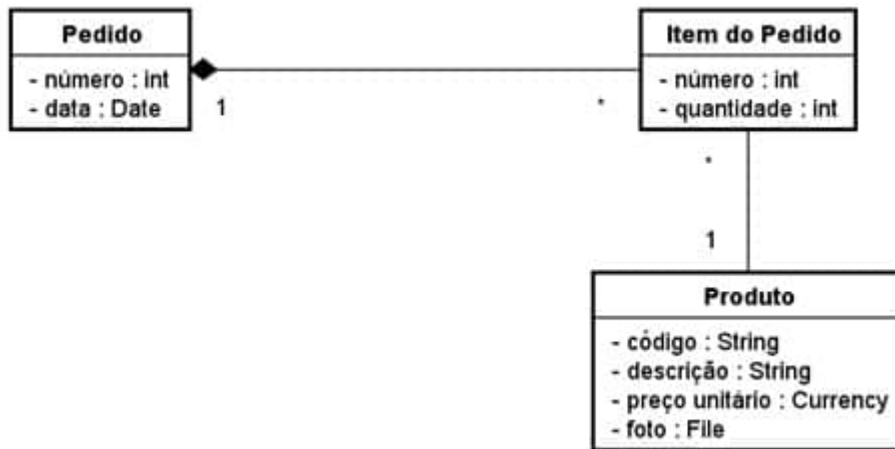
COMENTÁRIO

Atribuir a responsabilidade ao especialista é uma heurística intuitiva que utilizamos no nosso cotidiano. A quem você atribuiria a responsabilidade de trocar a parte elétrica da sua casa? Possivelmente a alguém com o conhecimento necessário para realizar essa atividade (um especialista), ou seja, um eletricista.

Suponha que você esteja desenvolvendo um site de vendas de produtos pela internet.

A figura a seguir apresenta um modelo simplificado de classes desse domínio. Digamos que o site deva apresentar o pedido do cliente e o valor total do pedido.

Como você organizaria as responsabilidades entre as classes, para fornecer essa informação?



📷 Padrão Especialista – classes de domínio.

O valor total do pedido pode ser definido como a soma do valor de cada um de seus itens.

Segundo o padrão **Especialista**, a responsabilidade deve ficar com o detentor da informação.

Nesse caso, quem conhece todos os itens que compõem um pedido? O próprio pedido, não é mesmo? Então, vamos definir a operação **obterValorTotal** na classe Pedido.

E onde ficaria o cálculo do preço de um item do pedido?

Quais informações são necessárias para esse cálculo?

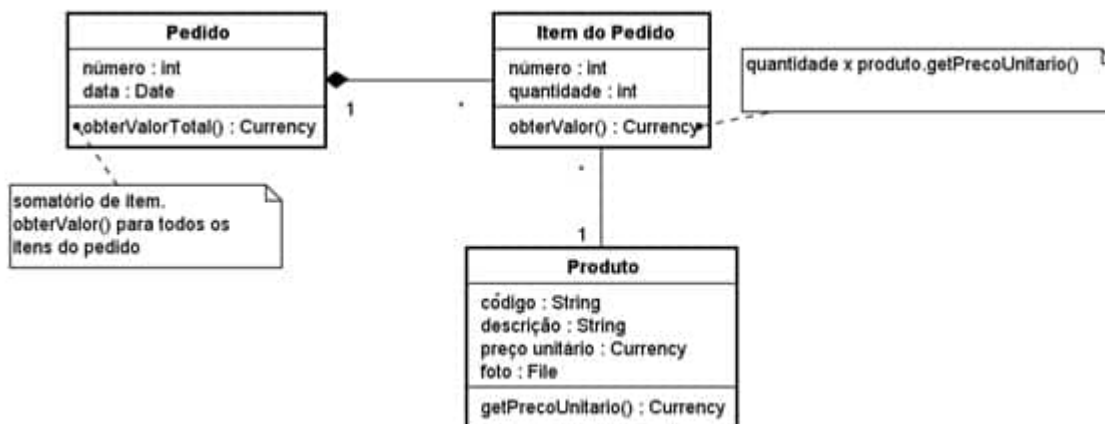
A quantidade e o preço do produto. Quem conhece essas informações?

A classe Item do Pedido conhece a quantidade e o produto associado. Vamos definir, então, uma operação **obterValor** na classe Item do Pedido.

E o preço do produto?

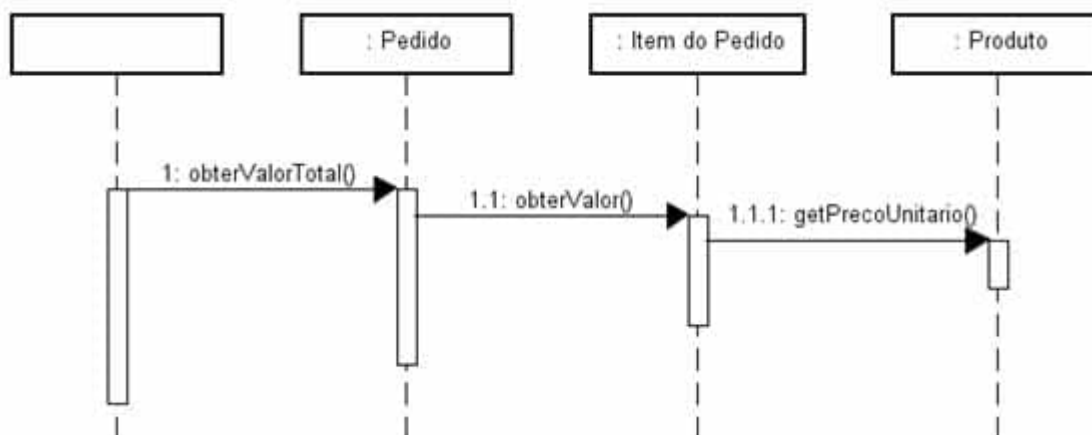
Basta o objeto item do pedido pedir essa informação para o produto, por meio da operação de acesso **getPrecoUnitario**.

A figura seguinte apresenta o diagrama de classes com a alocação de responsabilidades resultante.



📷 Padrão Especialista – alocação de responsabilidades.

O diagrama de sequência da próxima figura ilustra a colaboração definida para implementar a obtenção do valor total de um pedido.



📷 Padrão Especialista – colaboração (opção 1).

Consequências:

Quando o padrão Especialista não é seguido, é comum encontrarmos uma solução deficiente (antipadrão) conhecida como “**God Class**” – que consiste em definir apenas operações de acesso (conhecidas como **getters** e **setters**) nas classes de domínio – e concentrar toda a lógica de determinada funcionalidade do sistema em uma única classe, usualmente definida na forma de uma classe de controle ou de serviço. Essa classe implementa procedimentos utilizando as operações de acesso das diversas classes de domínio, que, nesse estilo de solução, são conhecidas como classes “idiotas”.

Existem, entretanto, situações em que a utilização desse padrão pode comprometer conceitos como coesão e acoplamento.

★ EXEMPLO

Qual classe deveria ser responsável por implementar o armazenamento dos dados de um Pedido no banco de dados?

Pelo princípio do Especialista, deveria ser a própria classe Pedido, uma vez que ela detém todas as informações que serão armazenadas. Porém, essa solução acoplaria a classe de negócio com conceitos relativos à tecnologia de armazenamento (e.g. SQL, NoSQL, arquivos etc.), ferindo o princípio fundamental da coesão, pois a classe Pedido ficaria sujeita a dois tipos de mudança: mudanças no negócio e mudanças na tecnologia de armazenamento, o que é absolutamente inadequado.

CRIADOR

Problema

A instanciação de objetos é uma das instruções mais presentes em um programa orientado a objetos. Embora um comando simples, como *new ClasseX* em Java, resolva a questão, a instanciação indiscriminada – e sem critérios bem definidos – de objetos por todo o sistema tende a gerar uma estrutura pouco flexível, difícil de modificar e com alto acoplamento entre os módulos.

Portanto, a pergunta que este padrão tenta responder é:

Quem deve ser responsável pela instanciação de um objeto de determinada classe?

Solução

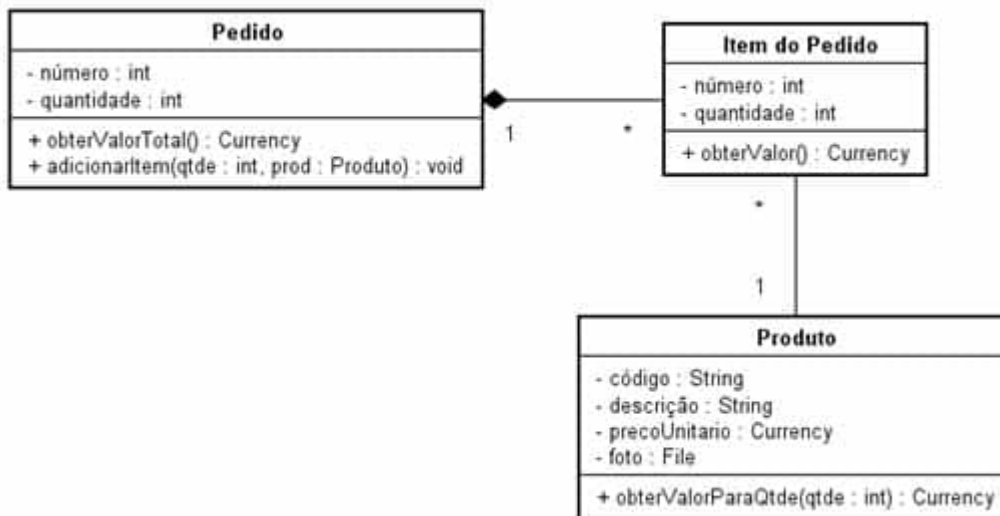
Coloque em uma classe X a responsabilidade de criar uma instância da classe Y se X for um agregado formado por instâncias de Y, ou se X possuir os dados de inicialização necessários para a criação de uma instância de Y.


No exemplo do site de vendas de produtos pela internet, considerando o modelo de classes da figura **Especialista – classes de domínio**, qual classe deveria ser responsável pela criação das instâncias de Item do Pedido?

RESPOSTA

Uma abordagem comum, mas inadequada, é instanciar o item em uma classe de serviço e apenas acumulá-lo no Pedido. Entretanto, quando se trata de uma relação entre um agregado e suas partes, a responsabilidade pela criação das partes deve ser alocada ao agregado, responsável por todo o ciclo de vida das suas partes (criação e destruição).

A figura a seguir apresenta o diagrama de classes após definirmos a operação **adicionarItem** na classe Pedido. Perceba que as operações vão sendo definidas nas diversas classes à medida que estabelecemos os mecanismos de colaboração para cada funcionalidade do sistema.



 Padrão Creator – classes de domínio.

Consequências

O padrão Criador é especialmente indicado para a criação de instâncias que formam parte de um agregado, por promover uma solução de menor acoplamento.



Por outro lado, o padrão Criador não é apropriado em algumas situações especiais, como, por exemplo, a criação condicional de uma instância dentro de uma família de classes similares.

BAIXO ACOPLAMENTO

Problema

Acoplamento corresponde ao grau de dependência de um módulo em relação a outros módulos do sistema. Um módulo com alto acoplamento depende de vários outros módulos e tipicamente apresenta problemas como propagação de mudanças pelas relações de dependência, dificuldade de entendê-lo isoladamente e dificuldade de reusá-lo em outro contexto, por exigir a presença dos diversos módulos que formam a sua cadeia de dependências.

Outra questão importante com relação ao acoplamento é a natureza das dependências.

Se uma classe A depende de uma classe B, dizemos que A depende de uma implementação concreta presente em B.



Por outro lado, se uma classe A depende de uma interface I, dizemos que A depende de uma abstração, pois A poderia trabalhar com diferentes implementações concretas de I, sem depender de nenhuma específica.

De forma geral, sistemas mais flexíveis são construídos quando fazemos implementações (classes) dependerem de abstrações (interfaces), especialmente quando a interface abstrair diferentes possibilidades de implementação, seja por envolver diferentes soluções tecnológicas (ex.: soluções de armazenamento e recuperação de dados), seja por envolver diferentes questões de negócio (ex.: diferentes regras de negócio, diferentes fornecedores de uma mesma solução de pagamento etc.).

Portanto, a pergunta que este padrão tenta responder é:

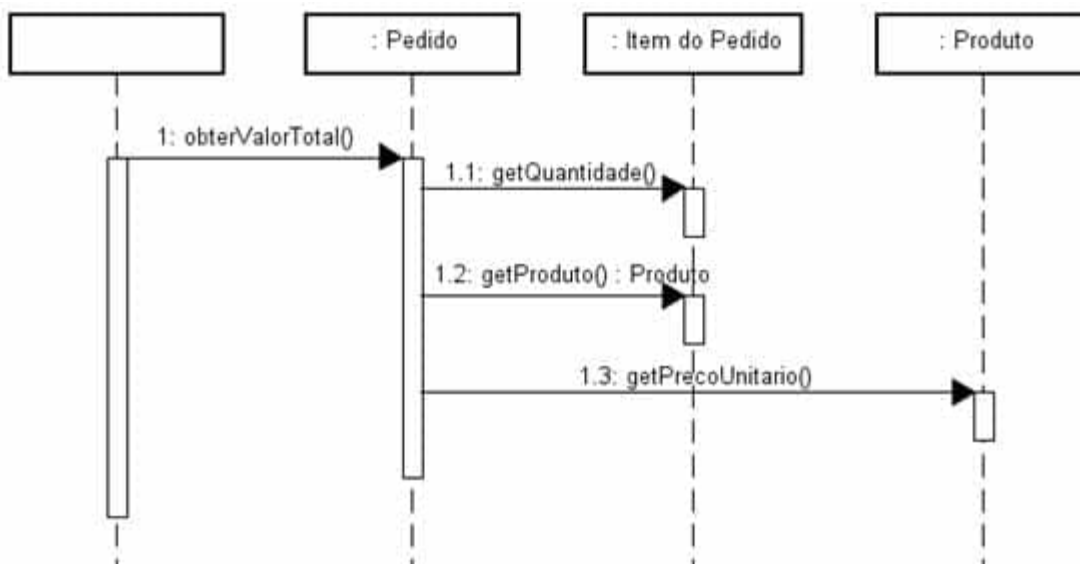
Como definir relações de dependência entre as classes de um sistema de forma a manter o acoplamento baixo, minimizar o impacto de mudanças e facilitar o reuso?

Solução

Distribuir as responsabilidades de forma a gerar um baixo acoplamento entre os módulos.

Para você entender o conceito de baixo acoplamento, vamos apresentar um contraexemplo, isto é, uma situação em que foi criado um acoplamento maior do que o desejado. A figura seguinte apresenta outra solução para o problema mostrado no padrão Especialista, na qual fazemos o cálculo do preço do item do pedido dentro da operação obterValorTotal da classe Pedido. Para isso, percorremos todos os itens do pedido, obtemos a quantidade e o produto de cada item, o preço unitário do produto e multiplicamos a quantidade pelo preço unitário.

Essa solução gerou um acoplamento entre Pedido e Produto que não está presente na solução dada pelo padrão Especialista.



📷 Padrão Baixo Acoplamento – exemplo de alto acoplamento.

Consequências

Acoplamento é um princípio fundamental da estruturação de software que deve ser considerado em qualquer decisão de projeto de software. Portanto, avalie com cuidado se cada acoplamento definido no projeto é realmente necessário ou se existem alternativas que levariam a um menor acoplamento, ou, ainda, se alguma abstração poderia ser criada para não gerar dependência com uma implementação específica.

Cuidado para não gerar soluções excessivamente complexas em que não haja motivação real para criar soluções mais flexíveis. Em geral, manter as classes de domínio isoladas e não dependentes de tecnologia (ex.: persistência, **GUI**, integração entre sistemas) é uma política geral de acoplamento que deve ser seguida, recomendada por diversas proposições de arquitetura.

GUI

Interface Gráfica do Usuário, do inglês *Graphical User Interface* .

ALTA COESÃO

Problema

Coesão é uma forma de avaliar se as responsabilidades de um módulo estão fortemente relacionadas e têm o mesmo propósito. Buscamos criar módulos com alta coesão, isto é, elementos com responsabilidades focadas e com um tamanho aceitável.

COMENTÁRIO

Módulos ou classes com baixa coesão realizam muitas operações pouco correlacionadas, gerando sistemas de difícil entendimento, reuso, manutenção e muito sensíveis às mudanças.

Portanto, a pergunta que este padrão tenta responder é:

Como definir as responsabilidades dos módulos de forma que a complexidade resultante seja gerenciável?

Solução

A solução consiste em definir módulos de alta coesão. Mas como se mede a coesão?

Coesão está ligada ao critério utilizado para reunir um conjunto de elementos em um mesmo módulo.

Coesão está ligada ao critério utilizado para reunir um conjunto de elementos em um mesmo módulo.

1

COESÃO DE UM MÉTODO DE UMA CLASSE

Um método reúne um conjunto de instruções.

COESÃO DE UMA CLASSE

Uma classe reúne um conjunto de atributos e operações.

2

3

COESÃO DE UM PACOTE

Um pacote reúne um conjunto de classes e interfaces.

COESÃO DE UM SUBSISTEMA

Um subsistema reúne um conjunto de pacotes.

4

A coesão de um módulo, seja ele classe, pacote ou subsistema, pode ser classificada de acordo com o critério utilizado para reunir o conjunto dos elementos que o compõem.

Devemos estruturar os módulos de forma que eles apresentem coesão funcional, isto é, os elementos são agrupados porque, juntos, cumprem um único propósito bem definido.

As classes do pacote `java.io` da linguagem Java, por exemplo, estão reunidas por serem responsáveis pela entrada e saída de um programa. Nesse pacote, encontramos classes com responsabilidades bem específicas, como:

FILEOUTPUTSTREAM

(para escrita de arquivos binários)

FILEINPUTSTREAM

(para leitura de arquivos binários)

FILEREADER

(para leitura de arquivos texto)

FILEWRITER

(para escrita de arquivos texto)

Consequências

Coesão e acoplamento são princípios fundamentais em projetos de software. A base da modularidade de um software está na definição de módulos com alta coesão e baixo acoplamento.

COMENTÁRIO

Sistemas construídos com módulos apresentando alta coesão tendem a ser mais flexíveis, mais fáceis de serem entendidos e evoluídos, além de proporcionarem mais possibilidades de reutilização e de um projeto com baixo acoplamento.

Entretanto, em sistemas distribuídos é preciso balancear a elaboração de módulos com responsabilidades específicas com o princípio fundamental de sistemas distribuídos, que consiste em minimizar as chamadas entre processos.

CONTROLADOR

Problema

Um sistema interage com elementos externos, também conhecidos como atores. Muitos elementos externos geram eventos que devem ser capturados pelo sistema, processados e produzir alguma resposta, interna ou externa.

Por exemplo, quando o cliente solicita o fechamento de um pedido na nossa loja online, esse evento precisa ser capturado e processado pelo sistema. Este padrão procura resolver o seguinte problema:

A quem devemos atribuir a responsabilidade de tratar os eventos que correspondam a requisições de operações para o sistema?

Solução

Atribuir a responsabilidade de receber um evento do sistema e coordenar a produção da sua resposta a uma classe que represente uma das seguintes opções:

Opção 1

Uma classe correspondente ao sistema ou a um subsistema específico, solução conhecida pelo nome **Controlador Fachada**. Normalmente é utilizada em sistemas com poucos eventos.



Opção 2

Uma classe correspondente a um caso de uso onde o evento ocorra. Normalmente essa classe tem o nome formado pelo nome do caso de uso com um sufixo *Processador*, *Controlador*, *Sessão* ou algo similar.

Essa classe deve reunir o tratamento de todos os eventos que o sistema receba no contexto deste caso de uso. Esta solução evita a concentração das responsabilidades de tratamento de eventos de diferentes funcionalidades em um único *Controlador Fachada*, evitando a criação de um módulo com baixa coesão.

ATENÇÃO

Note que esta classe não cumpre responsabilidades de interface com o usuário. Em um sistema de *home banking*, por exemplo, o usuário informa todos os dados de uma transação de transferência em um componente de interface com o usuário e, ao pressionar o botão transferir, esse componente delega a requisição para o controlador realizar o processamento lógico da transferência. Assim, o mesmo controlador pode atender a solicitações realizadas por diferentes interfaces com o usuário (web, dispositivo móvel, totem 24 horas).

Consequências

Normalmente, um módulo Controlador, ao receber uma requisição, coordena e controla os elementos que são responsáveis pela produção da resposta.

Em uma orquestra, um maestro comanda o momento em que cada músico deve entrar em ação, mas ele mesmo não toca nenhum instrumento.

Da mesma forma, um módulo Controlador é o grande orquestrador de um conjunto de objetos, cada qual com sua responsabilidade específica na produção da resposta ao evento.

Um problema que pode ocorrer com este padrão é alocar ao Controlador responsabilidades além da orquestração, como se o maestro, além de comandar os músicos, ficasse responsável também por tocar piano, flauta e outros instrumentos. Essa concentração de responsabilidades

no Controlador gera um módulo grande, complexo e que ninguém se sente confortável em evoluir.

POLIMORFISMO

Problema

Como evitar construções condicionais complexas no código?

Suponha que você esteja implementando a parte de pagamento em cartão de uma loja virtual. Para realizar um pagamento interagindo diretamente com uma administradora de cartão, temos que passar por um processo longo e complexo de homologação junto a ela. Imagine realizar esse processo com cada administradora!

Para simplificar o problema, existem diferentes **brokers** de pagamento que já são homologados com as diversas administradoras e fornecem uma **API** para integração com a nossa aplicação. Cada **broker** tem uma política de preços e volume de transações e, eventualmente, podem surgir novos **broker** com políticas mais atrativas no mercado.

BROKER

Broker é o software intermediário que permite que programadores façam chamadas a programas de um computador para outro.

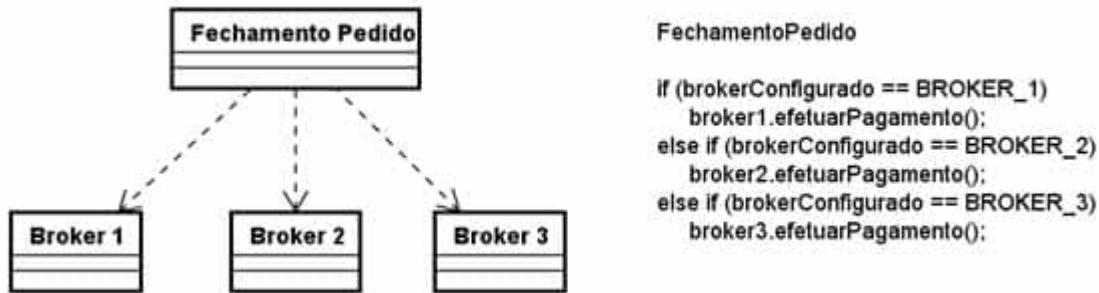
API

Interface de Programação de Aplicação, do inglês **Application Programming Interface**

Agora imagine que fornecemos uma solução de software de loja virtual para diferentes lojas, que podem demandar diferentes **brokers** de pagamento em função das suas exigências de segurança, preço e volume de transações. Isso significa que o nosso software tem que ser capaz de funcionar com diferentes **brokers**, cada um com a sua API.

Sem polimorfismo, esse problema poderia ser resolvido com uma solução baseada em ***if-then-else*** ou ***switch-case***, sendo cada alternativa de ***brokers*** mapeada para um comando case no switch ou em uma condição no if-then-else (figura a seguir).

Pense como ficaria esse código se houvesse vinte ***brokers*** diferentes.



📷 Solução sem polimorfismo.

O problema que esse padrão resolve é o seguinte:

Como escrever uma solução genérica para alternativas baseadas no tipo de um elemento, criando módulos plugáveis?

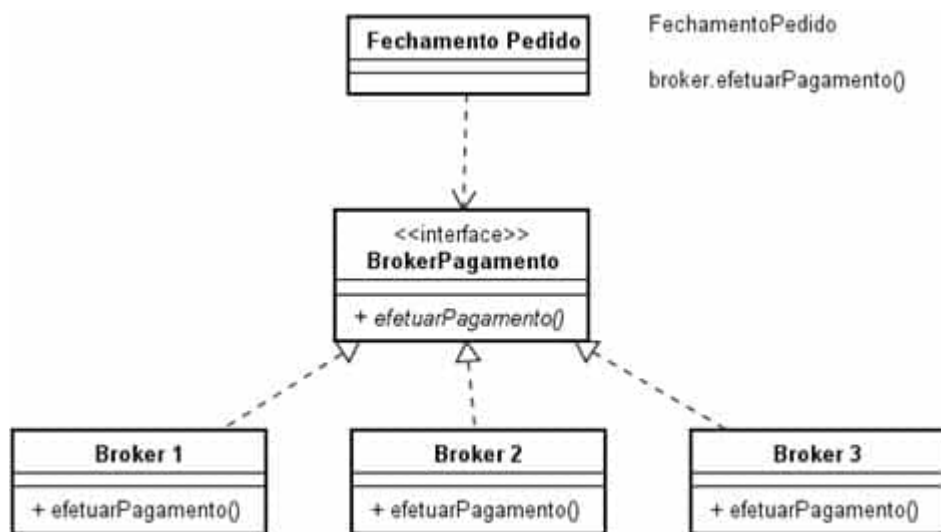
No nosso exemplo, as alternativas são todos os diferentes tipos de *brokers* com as suas respectivas APIs.

Solução

Percebeu como a solução baseada em estruturas condicionais do tipo *if-then-else* ou *switch-case*, além de serem mais complexas, criam um acoplamento do módulo chamador com cada implementação específica? Note como a classe **Fechamento Pedido** depende diretamente de todas as implementações de *broker* de pagamento.

A solução via polimorfismo consiste em criar uma interface genérica para a qual podem existir diversas implementações específicas (veja na figura seguinte).

A estrutura condicional é substituída por uma única chamada utilizando essa interface genérica. O chamador, portanto, não precisa saber quem está do outro lado da interface concretamente provendo a implementação. Essa capacidade de um elemento (a interface genérica) poder assumir diferentes formas concretas (*broker1*, *broker2* ou *broker3*, no nosso exemplo) é conhecida como polimorfismo.



📷 Solução com polimorfismo.

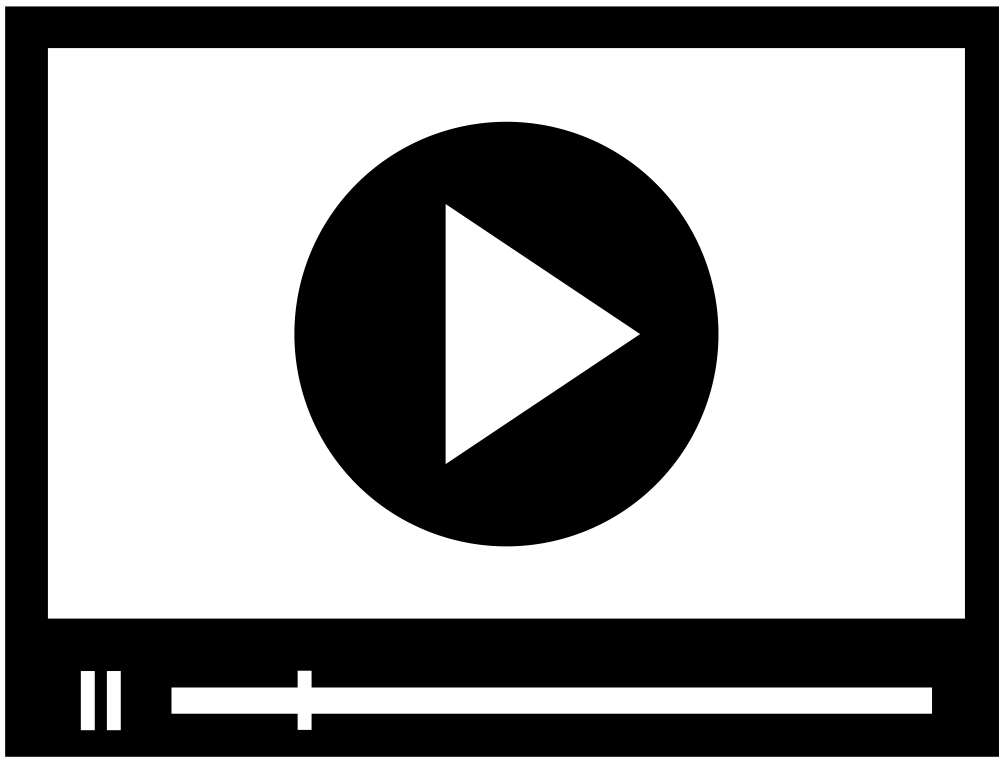
Consequências

Polimorfismo é um princípio fundamental em projetos de software orientados a objetos que nos ajuda a resolver, de forma sintética, elegante e flexível, o problema de lidar com variantes de implementação de uma mesma operação conceitual.

💡 DICA

Esse princípio geral é utilizado para a definição de diversos padrões GoF, tais como: Adapter, Command, Composite, Proxy, State e Strategy.

Entretanto, é preciso ter cuidado para não implementar estruturas genéricas em situações em que não haja possibilidade de variação. Uma solução genérica é mais flexível, mas é preciso estar atento para não investir esforço na produção de soluções genéricas para problemas que sejam específicos por natureza, isto é, que não apresentem variantes de implementação.



COESÃO E ACOPLAMENTO, OS PILARES DA ESTRUTURAÇÃO DE SOFTWARE

Os conceitos de coesão e acoplamento são apresentados de maneira detalhada no vídeo a seguir.



VERIFICANDO O APRENDIZADO

MÓDULO 3

🕒 Descrever as características dos princípios SOLID

PRINCÍPIOS SOLID

Você já ouviu falar em **SOLID**?

Esse termo é frequentemente encontrado como um conhecimento requisitado em perfis de vagas para desenvolvimento de software.

SOLID é um acrônimo para cinco princípios de projeto orientado a objetos, definidos por Robert C. Martin, que devem ser seguidos para nos ajudar a produzir software com uma estrutura sólida, isto é, uma estrutura que permita sua evolução e manutenção com menor esforço.

Cada letra do acrônimo corresponde a um dos princípios:

S

Single-Responsibility Principle (SRP)

Open-closed Principle (OCP)

O

L

Liskov Substitution Principle (LSP)

Interface Segregation Principle (ISP)

I

D

Dependency Inversion Principle (DIP)

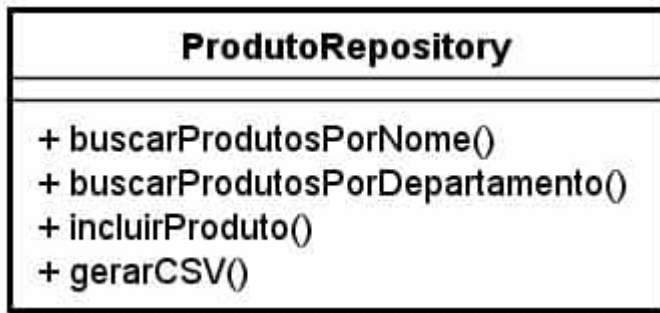
PRINCÍPIO DA RESPONSABILIDADE ÚNICA (SRP)

O princípio **SRP** (do inglês ***Single Responsibility Principle***) estabelece que o módulo deve ser responsável por um – e apenas um – ator, representando o papel desempenhado por alguém envolvido com o software. Nesse contexto, um módulo corresponde a um arquivo-fonte.

★ EXEMPLO

Em Java, um módulo corresponde a uma classe.

Suponha que a nossa loja virtual tenha requisitos de busca de produtos por nome e por departamento, cadastro de um novo produto e exportação de dados dos produtos em formato CSV. Se pensarmos conforme o padrão Especialista, por exemplo, todas essas funcionalidades trabalham com os dados dos produtos e, portanto, poderiam ser alocadas ao módulo `ProdutoRepository`, conforme ilustrado na figura seguinte.



📷 Violação do princípio SRP.

📢 ATENÇÃO

Essa alocação de responsabilidades viola o princípio SRP, pois define um módulo que atende a diferentes atores.

Vamos analisar os atores envolvidos com cada responsabilidade do módulo:

Pesquisar produtos por nome e por departamento são demandas do cliente da loja.

Incluir produto é uma demanda da área de vendas.

GerarCSV é uma demanda da área de integração de sistemas.

Portanto, o módulo ProdutoRepository apresenta diferentes razões para mudar: a área de integração pode demandar um novo formato para a exportação dos dados dos produtos, enquanto os clientes podem demandar novas formas de pesquisar os produtos.

De acordo com o princípio SRP, devemos separar essas responsabilidades em três módulos, o que nos daria a flexibilidade de, por exemplo, separar a busca de produtos e a inclusão de produtos em serviços fisicamente independentes, o que possibilitaria atender diferentes demandas de escalabilidade de forma mais racional.

★ EXEMPLO

A busca de produtos pode ter que atender a milhares de clientes simultâneos, enquanto esse número, no caso da inclusão de produtos, não deve passar de algumas dezenas de usuários.

PRINCÍPIO ABERTO FECHADO (OCP)

O princípio OCP (do inglês ***Open Closed Principle***) estabelece que um módulo deve estar aberto para extensões, mas fechado para modificações, isto é, seu comportamento deve ser extensível sem que seja necessário alterá-lo para acomodar as novas extensões.

Quando uma mudança em um módulo causa uma cascata de modificações em módulos dependentes, isso é sinal de que a estrutura do software não acomoda mudanças adequadamente. Se aplicado de forma adequada, este princípio permite que futuras mudanças desse tipo sejam feitas pela adição de novos módulos, e não pela modificação de módulos já existentes.

Você deve estar pensando como é possível fazer com que um módulo estenda seu comportamento sem que seja necessário mudar uma linha do seu código. A chave para esse enigma chama-se **abstração**. Em uma linguagem orientada a objetos, é possível criar uma interface abstrata que apresente diferentes implementações concretas e, portanto, um novo comportamento, em conformidade com essa abstração, pode ser adicionado simplesmente criando-se um módulo.

O exemplo a seguir apresenta um código que não segue o princípio ***Open Closed***. A classe **CalculadoraGeometrica** contém operações para calcular a área de triângulos e quadrados.

Violação do princípio OCP (clonagem).

```
public class Triangulo {  
    private double base;  
    private double altura;  
  
    public double getBase() {  
        return base;  
    }  
  
    public void setBase(double base) {  
        this.base = base;  
    }  
  
    public double getAltura() {  
        return altura;  
    }  
  
    public void setAltura(double altura) {
```



```

        this.altura = altura;
    }
}

public class Quadrado {
    double lado;

    public double getLado() {
        return lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }
}

public class CalculadoraGeometrica {
    public double obterArea(Quadrado quadrado) {
        return quadrado.getLado() * quadrado.getLado();
    }

    public double obterArea(Triangulo triangulo) {
        return triangulo.getBase() * triangulo.getAltura() / 2;
    }
}

```

Imagine que o programa tenha que calcular a área de um círculo, por exemplo.

O módulo CalculadoraGeometrica poderá calcular a área de um círculo sem que o seu código seja alterado?

Não, pois teremos que adicionar uma nova operação obterArea à CalculadoraGeometrica.

Essa é uma forma comum de violação, em que uma classe concentra diversas operações da mesma natureza que operam sobre tipos diferentes.

O exemplo a seguir ilustra outra forma comum de **violação do princípio OCP**.

```

public class FiguraGeometrica {
}

public class Triangulo extends FiguraGeometrica {
    private double base;
    private double altura;

    public double getBase() {

```

```

        return base;
    }

    public void setBase(double base) {
        this.base = base;
    }

    public double getAltura() {
        return altura;
    }

    public void setAltura(double altura) {
        this.altura = altura;
    }
}

public class Quadrado extends FiguraGeometrica {
    double lado;

    public double getLado() {
        return lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }
}

public class CalculadoraGeometrica {
    public double obterArea(FiguraGeometrica figura) {
        if (figura instanceof Quadrado) {
            return obterAreaQuadrado((Quadrado) figura);
        } else if (figura instanceof Triangulo) {
            return obterAreaTriangulo((Triangulo) figura);
        } else {
            return 0.0;
        }
    }

    private double obterAreaQuadrado(Quadrado quadrado) {
        return quadrado.getLado() * quadrado.getLado();
    }

    private double obterAreaTriangulo(Triangulo triangulo) {
        return triangulo.getBase() * triangulo.getAltura() / 2;
    }
}

```

```
}  
}
```

Neste caso, a classe `CalculadoraGeometrica` possui uma operação `obterArea`, que recebe um tipo genérico `FiguraGeometrica`, mas continua concentrando a lógica de cálculo para todos os tipos de figuras. Para incluir um círculo, precisaríamos adicionar a operação `obterAreaCirculo` e modificar a implementação da operação `obterArea`, inserindo uma nova condição. Imagine como ficaria o código dessa operação se ela tivesse que calcular a área de cinquenta tipos diferentes de figuras geométricas!

O que devemos fazer para que o módulo `CalculadoraGeometrica` possa estar aberto para trabalhar com novos tipos de figuras sem precisarmos alterar o seu código?

A resposta está no emprego da abstração e, neste exemplo, dos princípios do Especialista e do Polimorfismo.

Para isso, definimos que toda figura geométrica é capaz de calcular sua área, sendo que a fórmula de cálculo de cada figura específica deve ser implementada nas realizações concretas de `FiguraGeometrica`.

Dessa forma, para adicionar uma nova figura geométrica, basta adicionar um novo módulo com uma implementação específica para essa operação.

O próximo exemplo ilustra a nova estrutura do projeto.

A classe `FiguraGeometrica` define uma operação abstrata `obterArea`. Aplicando o princípio do Especialista, cada subclasse fica responsável pelo cálculo da sua área. Aplicando o princípio do Polimorfismo, o módulo `CalculadoraGeometrica` passa a depender apenas da abstração `FiguraGeometrica`, que define que todo elemento desse tipo é capaz de calcular a sua área. Dessa forma, a calculadora geométrica pode funcionar com novos tipos de figuras, sem que seja necessário alterar o seu código, uma vez que ela deixou de depender da forma específica da figura.

Reestruturação adequada ao princípio OCP.

```
public abstract class FiguraGeometrica {  
    public abstract double obterArea();  
}  
  
public class Triangulo extends FiguraGeometrica {  
    private double base;  
    private double altura;
```

```

public double obterArea() {
    return this.getBase() * this.getAltura() / 2;
}

public double getBase() {
    return base;
}

public void setBase(double base) {
    this.base = base;
}

public double getAltura() {
    return altura;
}

public void setAltura(double altura) {
    this.altura = altura;
}
}

```

```

public class Quadrado extends FiguraGeometrica {
    double lado;

    public double obterArea() {
        return this.getLado() * this.getLado();
    }

    public double getLado() {
        return lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }
}

```

```

public class CalculadoraGeometrica {
    public double obterArea(FiguraGeometrica figura) {
        return figura.obterArea();
    }
}

```

PRINCÍPIO DA SUBSTITUIÇÃO DE LISKOV (LSP)

O princípio LSP (do inglês ***Liskov Substitution Principle***) foi definido em 1988 por Barbara Liskov e estabelece que um tipo deve poder ser substituído por qualquer um de seus subtipos sem alterar o correto funcionamento do sistema.

O exemplo seguinte apresenta um caso clássico de violação do princípio de Liskov. Na geometria, um quadrado pode ser visto como um caso particular de um retângulo. Se levarmos essa propriedade para a implementação em software, podemos definir uma classe Quadrado como uma extensão da classe Retangulo.

ATENÇÃO

Note que, como a largura do quadrado é igual ao comprimento, a implementação dos métodos de acesso (setLargura e setComprimento) do quadrado deve se preocupar em preservar essa propriedade.

Princípio LSP (Quadrado)

```
public class Retangulo {
    private double largura;
    private double comprimento;

    public double area() {
        return largura * comprimento;
    }

    public double getLargura() {
        return largura;
    }
    public void setLargura(double largura) {
        this.largura = largura;
    }
    public double getComprimento() {
        return comprimento;
    }
    public void setComprimento(double comprimento) {
        this.comprimento = comprimento;
    }
}

public class Quadrado extends Retangulo {
    public void setLargura(double largura) {
        super.setLargura(largura);
        super.setComprimento(largura);
    }
    public void setComprimento(double largura) {
```

```
        super.setLargura(largura);
        super.setComprimento(largura);
    }
}
```

Porém, o próximo exemplo mostra como essa solução viola o princípio da substituição de Liskov. O método `verificarArea` da classe `ClienteRetangulo` recebe um objeto `r` do tipo `Retangulo`.

De acordo com esse princípio, o código desse método deveria funcionar com qualquer objeto de um tipo derivado de `Retangulo`. Entretanto, no caso de um objeto do tipo `Quadrado`, a execução da chamada `setComprimento` fará com que os dois atributos do quadrado passem a ter o valor 10. Em seguida, a execução da chamada `setLargura` fará com que os dois atributos do quadrado passem a ter o valor 8 e, portanto, o valor retornado pelo método `area` definido na classe `Retangulo` será 64 e não 80, violando o comportamento esperado para um retângulo.

Violação do princípio LSP

```
public class ClienteRetangulo {
    public void verificarArea(Retangulo r) throws Exception {
        r.setComprimento(10);
        r.setLargura(8);
        if (r.area() != 80) {
            throw new Exception("area incorreta");
        }
    }
}
```

Esse caso ilustra que, embora um quadrado possa ser classificado como um caso particular de um retângulo, o princípio de Liskov estabelece que um subtipo não pode estabelecer restrições adicionais que violem o comportamento genérico do supertipo. Nesse caso, o fato de o quadrado exigir que comprimento e largura sejam iguais o torna mais restrito que o retângulo. Portanto, definir a classe `Quadrado` como um subtipo de `Retangulo` é uma solução inapropriada para este problema, do ponto de vista da orientação a objetos, por violar o princípio de Liskov.

PRINCÍPIO DA SEGREGAÇÃO DE INTERFACES (ISP)

O princípio ISP (do inglês ***Interface Segregation Principle***) estabelece que clientes de uma classe não devem ser forçados a depender de operações que eles não utilizem.

O código do exemplo a seguir apresenta uma interface que viola esse princípio por reunir operações que dificilmente serão utilizadas em conjunto. As operações login e registrar estão ligadas ao registro e à autorização de acesso de um usuário, enquanto a operação logErro registra mensagens de erro ocorridas durante algum processamento, e a operação enviarEmail permite enviar um e-mail para o usuário logado.

Violação do princípio ISP

```
public interface IUsuario {  
    boolean login(String login, String senha);  
    boolean registrar(String login, String senha, String email);  
    void logErro(String msgErro);  
    boolean enviarEmail(String assunto, String conteudo);  
}
```

Segundo este princípio, devemos manter a coesão funcional das interfaces, evitando colocar operações de diferentes naturezas em uma única interface. Aplicando ao exemplo, poderíamos segregar a interface original em três interfaces (exemplo a seguir):

IAutorizacao, com as operações de login e registro de usuários;

IMensagem, com as operações de registro de erro, aviso e informações;

Email para o envio de e-mail.

Segregação das interfaces

```
public interface IAutorizacao {  
    boolean login(String login, String senha);  
    boolean registrar(String login, String senha, String email);  
}
```

```
public interface IEmail {  
    boolean enviarEmail(String assunto, String conteudo);  
}
```

```
public interface IMensagem {  
    void logErro(String msgErro);  
    void logAviso(String msgAviso);  
    void logInfo(String msgInfo);  
}
```

PRINCÍPIO DA INVERSÃO DE DEPENDÊNCIAS (DIP)

O princípio DIP (do inglês ***Dependency Inversion Principle***) estabelece que as entidades concretas devem depender de abstrações e não de outras entidades concretas. Isso é especialmente importante quando estabelecemos dependências entre entidades de camadas diferentes do sistema.

Segundo este princípio, um módulo de alto nível não deve depender de um módulo de baixo nível.

Módulos de alto nível

São aqueles ligados estritamente ao domínio da aplicação (ex.: Loja, Produto, ServiçoVenda etc.).



Módulos de baixo nível

São ligados a alguma tecnologia específica (ex.: acesso a banco de dados, interface com o usuário etc.) e, portanto, mais voláteis.

O próximo exemplo apresenta um caso de dependência de um módulo de alto nível em relação a um módulo de baixo nível. A classe `ServicoConsultaProduto` depende diretamente da implementação de um repositório de produtos que acessa um banco de dados relacional Oracle.

Essa dependência é estabelecida na operação `obterPrecoProduto` da classe `ServicoConsultaProduto` pelo comando `new ProdutoRepositoryOracle()`.

Violação do princípio da Inversão de Dependências

```
public class ProdutoRepositoryOracle {  
    public Produto obterProduto(String codigo) {  
        Produto p = new Produto();  
        // implementacao SQL de recuperacao do produto  
        return p;  
    }  
}
```



```

    public void salvarProduto(Produto produto) {
        // implementacao SQL de insert ou update do produto
    }
}

public class ServicoConsultaProduto {
    public double obterPrecoProduto(String codigo) {
        ProdutoRepositoryOracle repositório
            = new ProdutoRepositoryOracle();
        Produto produto = repositório.obterProduto(codigo);
        return produto.getPreco();
    }
}

```

A figura a seguir apresenta, em um diagrama UML, a relação de dependência existente entre a classe concreta `ServicoConsultaProduto` e a classe concreta `ProdutoRepositoryOracle`.



📷 Violação do DIP (Diagrama UML).

Por que essa dependência é inadequada?

Como aspectos tecnológicos são voláteis, devemos isolar a lógica de negócio, permitindo que a implementação concreta desses aspectos possa variar sem afetar as classes que implementam a lógica de negócio.

O exemplo a seguir apresenta uma solução resultante da aplicação deste princípio. A interface abstrata (`ProdutoRepository`) define um contrato abstrato entre cliente (`ServicoConsultaProduto`) e fornecedor (`ProdutoRepositoryOracle`). Em vez de o cliente depender de um fornecedor específico, ambos passam a depender dessa interface abstrata.

Assim, a classe `ServicoConsultaProduto` pode trabalhar com qualquer implementação concreta da interface `ProdutoRepository`, sem que seu código precise ser alterado.

```

public interface ProdutoRepository {
    Produto obterProduto(String codigo);
    void salvarProduto(Produto produto);
}

public class ProdutoRepositoryOracle implements ProdutoRepository {

```

```

public Produto obterProduto(String codigo) {
    Produto p = new Produto();
    // implementacao SQL de recuperacao do produto
    return p;
}

public void salvarProduto(Produto produto) {
    // implementacao SQL de insert ou update do produto
}
}

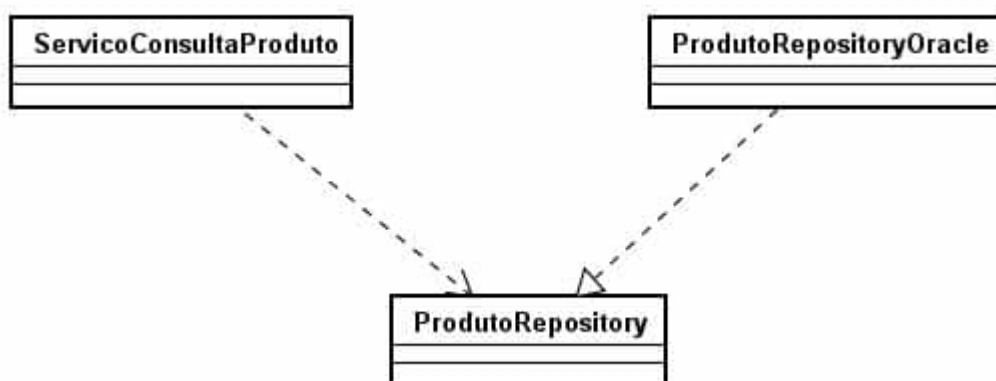
public class ServicoConsultaProduto {
    ProdutoRepository repositorio;

    public ServicoConsultaProduto(ProdutoRepository repositorio) {
        this.repositorio = repositorio;
    }

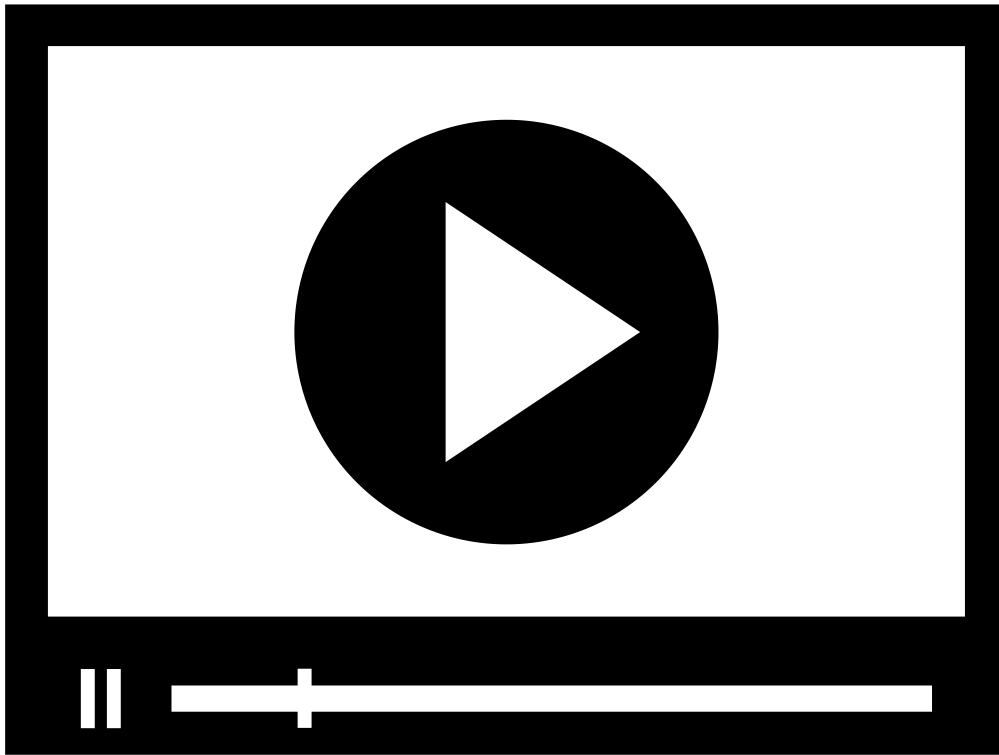
    public double obterPrecoProduto(String codigo) {
        Produto produto = repositorio.obterProduto(codigo);
        return produto.getPreco();
    }
}

```

A figura a seguir apresenta o diagrama UML correspondente à nova estrutura da solução após a aplicação desse princípio. A classe `ServicoConsultaProduto` depende apenas da interface `ProdutoRepository`, enquanto a classe `ProdutoRepositoryOracle` é uma das possíveis implementações concretas dessa interface.



📷 Diagrama UML da solução de acordo com o DIP.



PRINCÍPIOS SOLID X PADRÕES GRASP

No vídeo a seguir, é apresentada a relação entre os princípios SOLID e os padrões GRASP de alta coesão, baixo acoplamento e polimorfismo.



VERIFICANDO O APRENDIZADO

MÓDULO 4

🕒 Reconhecer o propósito dos principais padrões GoF e as situações nas quais eles podem ser aplicados

Este módulo apresenta uma seleção de alguns dos padrões GoF mais utilizados no desenvolvimento de sistemas.

FACTORY METHOD

Problema

Este padrão define uma interface genérica de criação de um objeto, deixando a decisão da classe específica a ser instanciada para as implementações concretas dessa interface. Este padrão é muito utilizado no desenvolvimento de **frameworks**.

Os **frameworks** definem pontos de extensão (**hot spots**) nos quais devem ser feitas adaptações do código para um sistema específico. Tipicamente, os **frameworks** definem classes e interfaces abstratas que devem ser implementadas nas aplicações que os utilizem. Um exemplo de *framework* muito usado em Java é o Spring.

FRAMEWORKS

Frameworks são estruturas de software parcialmente completas projetadas para serem instanciadas.

Por que este padrão é importante?

Em aplicações desenvolvidas em Java, por exemplo, podemos criar um objeto de uma classe simplesmente utilizando o operador *new*.

💬 COMENTÁRIO

Embora seja uma solução aceitável para pequenos programas, à medida que o sistema cresce, a quantidade de códigos para criar objetos também cresce, espalhando-se por todo o sistema.

Como a criação de um objeto determina dependência entre a classe onde a instanciação está sendo feita e a classe instanciada, é preciso muito cuidado para não criar dependências que dificultem futuras evoluções do sistema.

Solução

A figura a seguir apresenta a estrutura de solução proposta pelo padrão **Factory Method**.

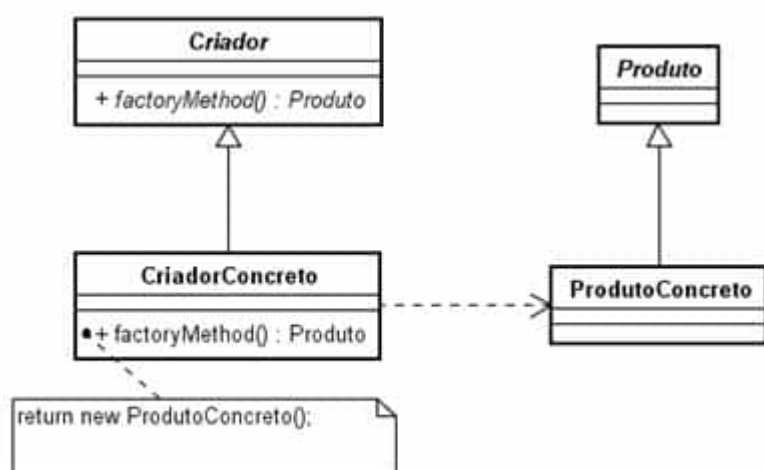
Os participantes são:

Produto: define o tipo abstrato dos objetos criados pelo padrão. Todos os objetos concretamente instanciados serão derivados deste tipo.

ProdutoConcreto: corresponde a qualquer classe concreta que implementa a definição abstrata do tipo Produto. Representa as classes específicas em uma aplicação desenvolvida com um **framework**.

Criador: declara um **Factory Method** que define uma interface abstrata que retorna um objeto genérico do tipo Produto.

CriadorConcreto: corresponde a qualquer classe concreta que implemente o **Factory Method** definido abstratamente no Criador.



📷 Estrutura do padrão **Factory Method**.

A figura seguinte apresenta um exemplo de aplicação do padrão.

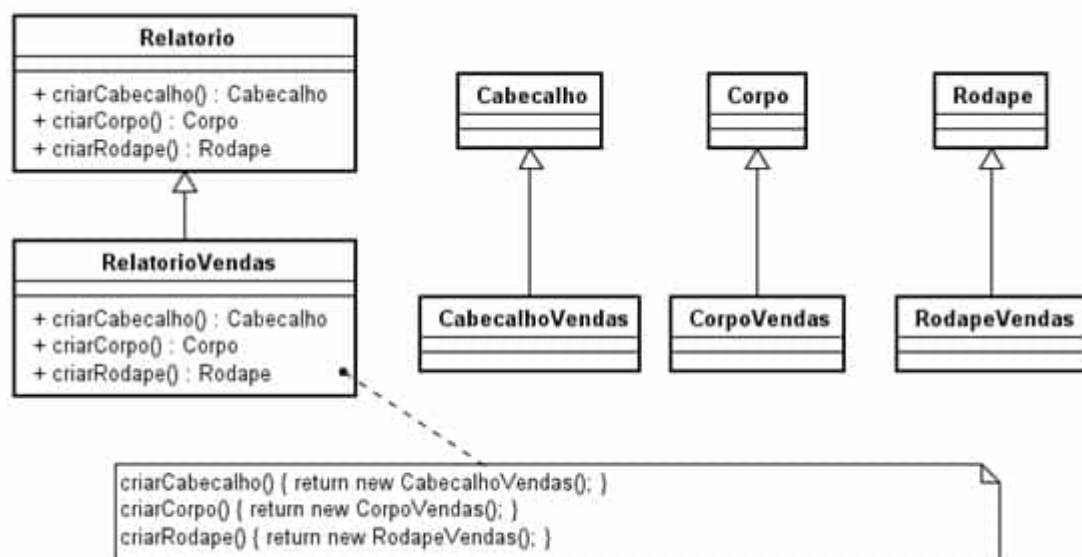
Na parte **superior**, estão as classes que pertencem a um **framework** genérico de geração de relatório.



Na parte **inferior**, estão as classes concretas de uma aplicação que utiliza o **framework** para gerar um relatório de vendas.

As operações `criarCabecalho`, `criarCorpo` e `criarRodape`, definidas na classe genérica, são implementadas concretamente na classe `RelatorioVendas`. Cada método dessa subclasse cria um objeto específico que implementa os tipos abstratos `Cabecalho`, `Corpo` e `Rodape` definidos no **framework**.

A classe `Relatorio` desempenha o papel de Criador, a classe `RelatorioVendas` corresponde ao participante `CriadorConcreto`, enquanto as classes `Cabecalho`, `Corpo` e `Rodape` desempenham o papel de Produto; finalmente, as classes `CabecalhoVendas`, `CorpoVendas` e `RodapeVendas` correspondem ao participante `ProdutoConcreto`.



📷 Exemplo do padrão Factory Method.

Consequências

O padrão **Factory Method** permite criar estruturas arquiteturais genéricas (**frameworks**), provendo pontos de extensão por meio de operações que instanciam os objetos específicos da aplicação. Dessa forma, todo código genérico pode ser escrito no **framework** e reutilizado em diferentes aplicações, evitando soluções baseadas em clonagem ou em estruturas condicionais complexas.

ADAPTER

Problema

O problema resolvido por este padrão é análogo ao da utilização de tomadas em diferentes partes do mundo.

★ EXEMPLO

Quando um turista americano chega em um hotel no Brasil, ele provavelmente não conseguirá colocar o seu carregador de celular na tomada que está na parede do quarto.



Como ele não pode mudar a tomada que está na parede, pois ela é propriedade do hotel, a solução é utilizar um adaptador que converta os pinos do carregador americano em uma saída de três pinos compatível com as tomadas brasileiras.

Como esse problema ocorre em um software?

Imagine que estamos desenvolvendo um software para a operação de vendas de diferentes lojas de departamentos, e que cada loja possa utilizar uma solução de pagamento dentre as diversas disponíveis no mercado.

O problema é que cada fornecedor permite a realização de um pagamento em cartão de crédito por meio de uma API específica fornecida por ele.

Portanto, o software de vendas deve ser capaz de ser plugado a diferentes API de pagamento que oferecem basicamente o mesmo serviço: intermediar as diversas formas de pagamento existentes no mercado.

Gostaríamos de poder fazer um software de vendas aderente ao princípio **Open Closed**, isto é, um software que pudesse trabalhar com novos fornecedores que apareçam no mercado, sem termos que mexer em módulos já existentes.

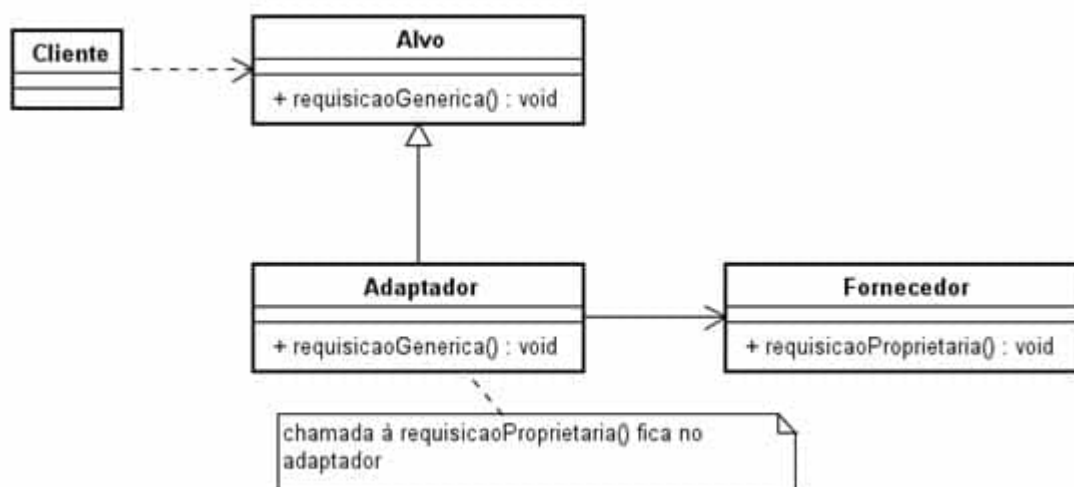
Solução

O Adapter é um padrão estrutural, cuja estrutura é apresentada na figura a seguir.

O participante Fornecedor define uma API proprietária e que não pode ser alterada. O participante Alvo representa a generalização dos serviços oferecidos pelos diferentes fornecedores. Os demais módulos do sistema utilizarão apenas este participante, ficando isolados do fornecedor concreto da implementação. O participante adaptador transforma uma requisição genérica feita pelo cliente em uma chamada à API específica do fornecedor com o qual ele está conectado.

📢 ATENÇÃO

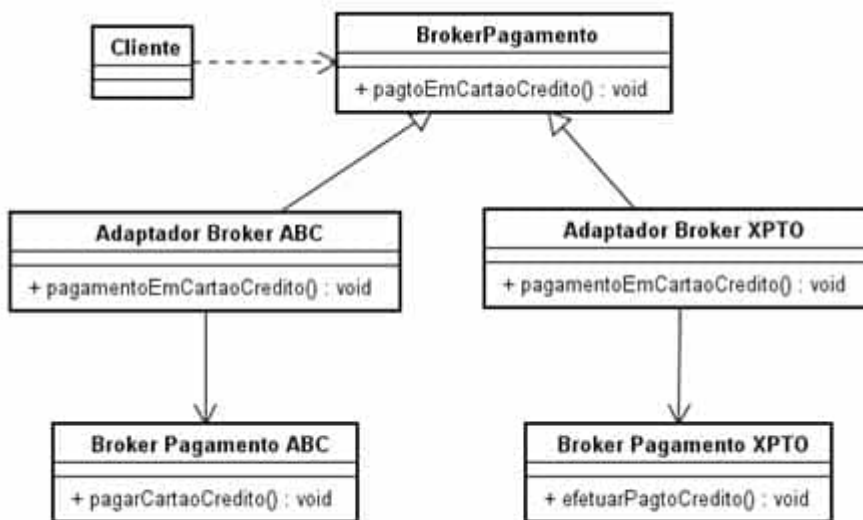
Note que cada fornecedor possuirá um adaptador específico, da mesma forma que existem adaptadores específicos para cada padrão de tomada.



📷 Estrutura do padrão Adapter

A figura a seguir apresenta como este padrão poderia ser aplicado no problema do software para a loja de departamentos. Suponha que existam dois **brokers** de pagamento (ABC e XPTO), sendo as suas respectivas API representadas pelas classes BrokerPagamentoABC e BrokerPagamentoXPTO. Note que as operações dessas classes, embora similares, têm nomes diferentes e poderiam também ter parâmetros de chamada e retorno diferentes. Essas classes são fornecidas pelos fabricantes e não podem ser alteradas.

A aplicação deste padrão consiste em definir uma interface genérica (BrokerPagamento) que será utilizada em todos os módulos do sistema que precisem interagir com o **brokers** de pagamento (representados genericamente pela classe Cliente do diagrama). Para cada API específica, criamos um adaptador que implementa a interface genérica BrokerPagamento, traduzindo a chamada da operação genérica pagtoEmCartaoCredito para o protocolo específico da API. Dessa forma, a operação do AdaptadorBrokerABC chamará a operação pagarCartaoCredito do BrokerPagamentoABC, enquanto a operação do AdaptadorBrokerXPTO chamará a operação efetuarPagtoCredito de BrokerPagamentoXPTO.



📷 Exemplo do padrão Adapter.

Consequências

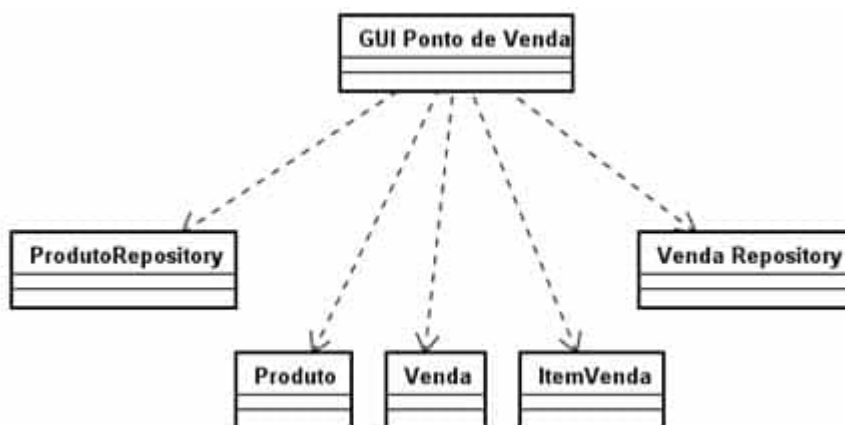
O padrão Adapter permite incorporar módulos previamente desenvolvidos, modificando sua interface original sem que haja necessidade de alterá-los, possibilitando a utilização de diferentes implementações proprietárias por meio de uma única interface, sem criar dependências dos módulos clientes com as implementações proprietárias.

FACADE

Problema

O problema resolvido pelo padrão estrutural **Facade** (fachada) é reduzir a complexidade de implementação de uma operação do sistema, fornecendo uma interface simples de uso, ao mesmo tempo em que evita que a implementação precise lidar com diferentes tipos de objetos e chamadas de operações específicas.

A próxima figura apresenta uma situação típica em que o padrão **Facade** pode ser utilizado. A classe GUI Ponto de Venda representa um elemento da camada de interface com o usuário responsável por registrar itens vendidos no caixa da loja. Para isso, quando os dados do item são entrados pelo operador, a implementação busca o produto chamando uma operação da classe ProdutoRepository, cria um ItemVenda adicionando-o a um objeto Venda, e salva a nova configuração da Venda chamando uma operação da classe VendaRepository. Essa solução é inadequada, pois cria inúmeras dependências e gera maior complexidade em uma classe de interface com o usuário.

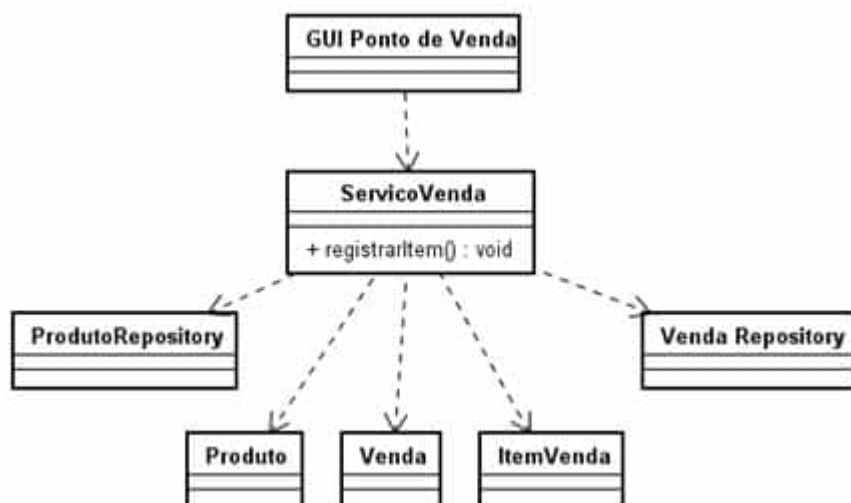


📷 Padrão Facade – problema.

Solução

A figura a seguir mostra como essas dependências entre um elemento de interface com o usuário e os elementos de negócio e armazenamento podem ser simplificadas pela introdução da classe ServicoVenda. Ela passa a servir de fachada para a execução de uma operação complexa do sistema, oferecendo uma interface única e simplificada para a classe GUI Ponto de Venda. Em vez de chamar várias operações de diferentes objetos, basta ela chamar a operação registrarItem definida na classe fachada. Dessa forma, os elementos da camada de

interface com o usuário ficam isolados da complexidade de implementação e da estrutura interna dos objetos pertencentes à lógica de negócio envolvida na operação.



📷 Padrão Facade – solução.

Consequências

O padrão *Facade* é bastante utilizado na estruturação dos serviços lógicos que um sistema oferece, isolando a camada de interface com o usuário da estrutura da lógica de negócio do sistema. Portanto, este padrão nada mais é do que a aplicação do princípio da abstração, em que isolamos um cliente de detalhes irrelevantes de implementação, promovendo uma estrutura com menor acoplamento entre as camadas.

STRATEGY

Problema

O problema resolvido pelo padrão **Strategy** ocorre quando temos diferentes algoritmos para realizar determinada tarefa.

★ EXEMPLO

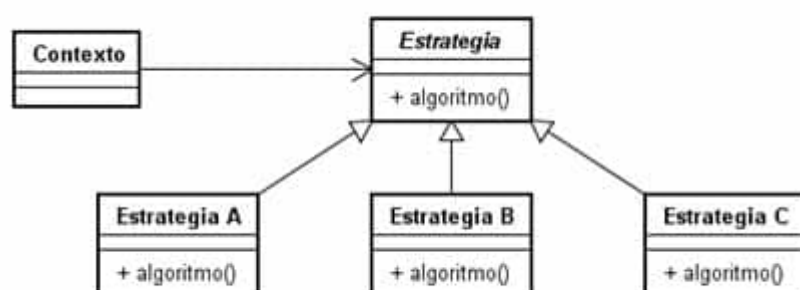
Uma loja de departamentos pode ter diferentes políticas de desconto aplicáveis em função da época do ano (ex.: Natal, Páscoa, Dia das Mães, Dia das Crianças).

Como organizar esses algoritmos de forma que possamos respeitar o princípio *Open Closed*, fazendo com que o sistema de vendas possa aplicar novas políticas de desconto sem que seja necessário modificar o que já está implementado?

Solução

O padrão Strategy define uma família de algoritmos, onde cada um é encapsulado em sua própria classe. Os diferentes algoritmos compõem uma família que pode ser utilizada de forma intercambiável, e novos algoritmos podem ser adicionados à família sem afetar o código existente.

A figura a seguir apresenta a estrutura do padrão. O tipo *Estrategia* define uma interface comum a todos os algoritmos, podendo ser implementado como uma classe abstrata ou como uma interface. Essa interface genérica é utilizada pelo participante *Contexto* quando este necessitar que o algoritmo seja executado. *Estrategia A*, *B* e *C* são implementações específicas do algoritmo genérico que o participante *Contexto* pode disparar. A estratégia específica deve ser injetada no módulo *Contexto*, que pode alimentar a estratégia com seus dados.



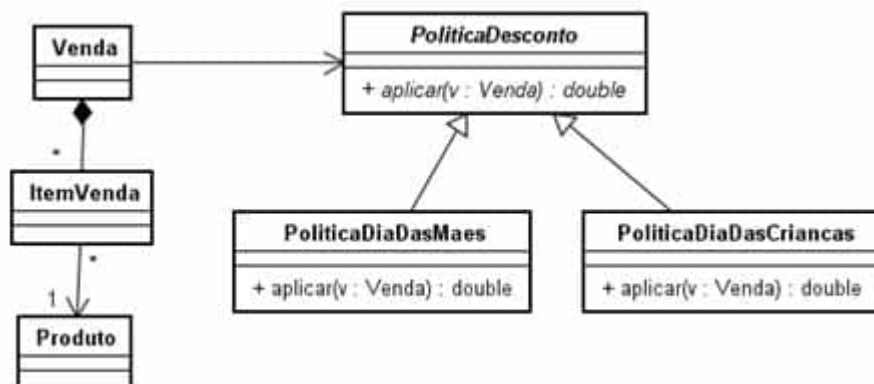
📷 Padrão Strategy – estrutura.

A próxima figura apresenta um exemplo de aplicação do padrão Strategy. A classe *Venda* representa uma venda da loja de departamentos. Uma política de desconto pode ser associada a uma venda no momento de sua instânciação e utilizada para cálculo do valor a ser pago pelo cliente.

Para obter o valor a pagar da *Venda*, basta obter o seu valor total, somando o valor dos seus itens, e subtrair o valor retornado pela operação *aplicar* da instância de *PoliticaDesconto*, associada ao objeto *venda*.

Note que novas políticas podem ser agregadas à solução, bastando adicionar uma nova implementação da interface *PoliticaDesconto*. Nenhuma alteração na classe *Venda* é

necessária, pois ela faz uso apenas da interface genérica, não dependendo de nenhuma política de descontos específica.



📷 Exemplo do padrão Strategy.

Consequências

O padrão Strategy permite separar algoritmos de diferentes naturezas dos elementos necessários para sua execução. No exemplo da loja de departamentos, podemos aplicar à Venda diferentes tipos de algoritmo, tais como: política de desconto, cálculo de frete e prazo de entrega, entre outros. Cada tipo de algoritmo conta com especializações que podem ser implementadas em uma família própria de algoritmos.

O padrão Strategy oferece uma solução elegante e extensível para o encapsulamento de algoritmos que podem ter diferentes implementações, isolando, de suas implementações concretas, os módulos clientes desses algoritmos. Além disso, permite a remoção de estruturas condicionais complexas normalmente presentes quando essas variações não são implementadas com este padrão.

TEMPLATE METHOD

Problema

O padrão **Template Method** é aplicável quando temos diferentes implementações de uma operação em que alguns passos são idênticos e outros são específicos de cada implementação.

Para ilustrar o problema, suponha a implementação de duas máquinas de bebidas: uma de café e outra de chá. O procedimento de preparação é bem similar, com alguns passos em

comum e outros específicos:

Máquina de café	Máquina de chá
Esquentar a água	Esquentar a água
Preparar mistura (via moagem do café)	Preparar mistura (via infusão do chá)
Colocar a mistura no copo	Colocar a mistura no copo
Adicionar açúcar, se selecionado	Adicionar açúcar, se selecionado
Adicionar leite , se selecionado	Adicionar limão , se selecionado
Liberar a bebida	Liberar a bebida



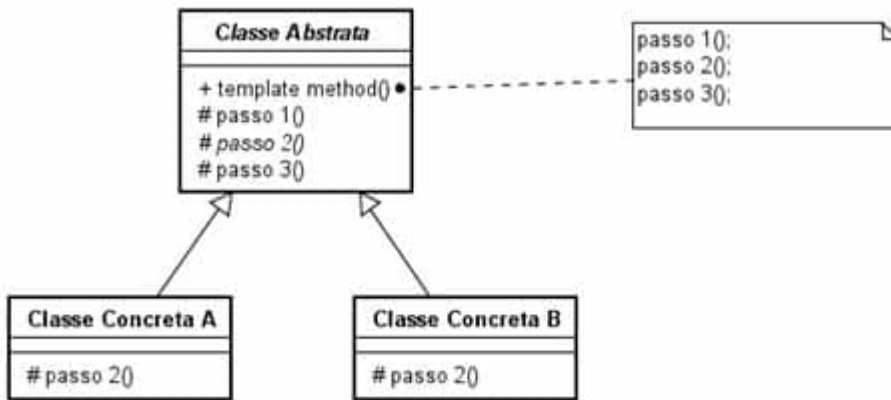
Atenção! Para visualização completa da tabela utilize a rolagem horizontal

O problema consiste em implementar diferentes formas concretas de um algoritmo padrão contendo pontos de variação, sem clonar o algoritmo em cada forma concreta, nem produzir uma única implementação contendo diversas estruturas condicionais.

Solução

A figura a seguir apresenta a estrutura geral da solução.

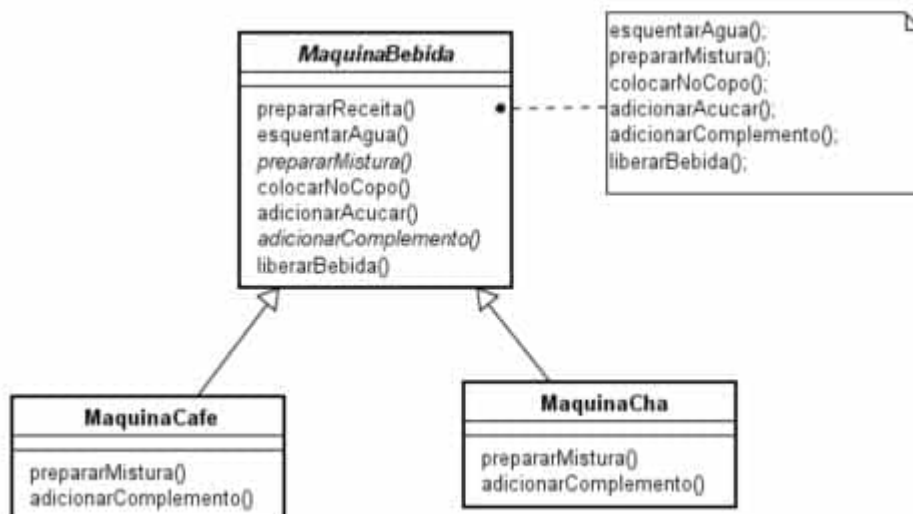
O procedimento genérico é definido em um método na classe abstrata. Cada passo do procedimento é definido como uma operação na classe abstrata. Alguns passos são comuns e, portanto, podem ser implementados na própria classe abstrata. Os passos específicos são definidos como operações abstratas na classe abstrata, sendo as implementações específicas implementadas nas subclasses, como é o caso do passo 2.



📷 *Template Method* – estrutura.

A figura seguinte apresenta uma solução para o problema da máquina de café e da máquina de chá utilizando este padrão.

O método `prepararReceita`, definido na superclasse `MaquinaBebida`, possui seis passos que são definidos como operações nesta mesma classe. Note que existem dois passos definidos como operações abstratas, pois são executados de forma específica nas duas máquinas: `prepararMistura` e `adicionarComplemento`. Cada máquina derivada da classe abstrata `MaquinaBebida` implementa apenas os passos específicos, isto é, apenas a implementação dessas duas operações abstratas.



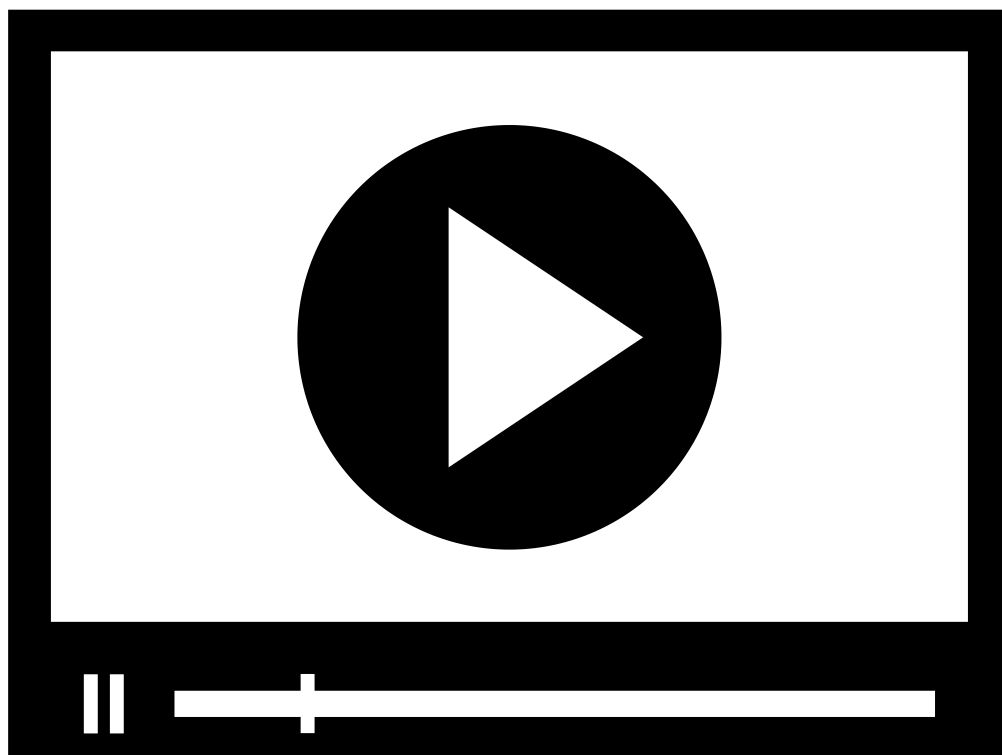
📷 *Template Method* – exemplo.

Consequências

O padrão **Template Method** evita a duplicação de algoritmos que apresentem a mesma estrutura, com alguns pontos de variação entre eles. O algoritmo é implementado apenas uma vez em uma classe abstrata onde são definidos os pontos de variação. As subclasses

específicas implementam esses pontos de variação. Dessa forma, qualquer alteração no algoritmo comum pode ser feita em apenas um módulo.

Esse padrão é muito utilizado na implementação de **frameworks**, permitindo a construção de estruturas de controle invertidas, também conhecidas como “princípio de Hollywood” ou “Não nos chame, nós o chamaremos”, referindo-se a estruturas em que a superclasse é quem chama os métodos definidos nas subclasses, e não o contrário.



PADRÕES DE PROJETO GOF

Uma visão geral dos problemas tratados pelos padrões GoF é explicitada no vídeo a seguir.



VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Neste estudo, você viu que os desenvolvedores de software devem produzir sistemas com uma estrutura adequada que permita a sua evolução com custos e prazos aceitáveis. Padrões de projeto formam um corpo de conhecimento muito útil, pois permitem a reutilização de boas soluções de projeto e a produção de sistemas bem estruturados.

Viu também como os padrões GRASP e o princípios SOLID oferecem um conjunto de conceitos fundamentais para qualquer projeto de software e, em especial, para projetos orientados a objetos. Saber estruturar sistemas em módulos com baixo acoplamento e alta coesão, aplicando conceitos como polimorfismo, inversão de dependências e módulos abertos para extensões sem precisarem ser modificados, é uma habilidade fundamental para os desenvolvedores de software atualmente.

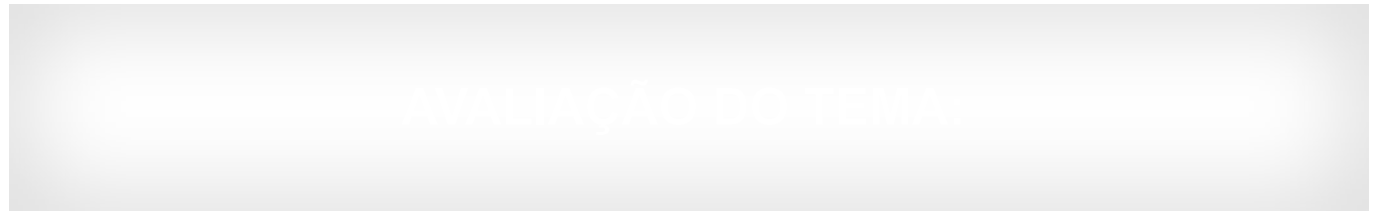
Finalmente, você conheceu os principais padrões de projeto GoF e viu como eles promovem o reuso de soluções para problemas recorrentes em projetos de software.

O assunto é vasto e de grande importância no mercado, sendo o seu conhecimento um diferencial do profissional de desenvolvimento de sistemas. Um aspecto interessante nessa área é que, dentro do espírito de reuso de padrões de projeto, a comunidade de desenvolvimento costuma compartilhar suas experiências de forma colaborativa.



PODCAST

No áudio a seguir, apresentamos um resumo sobre os padrões de projeto.



REFERÊNCIAS

BUSCHMANN, F. *et al.* **Pattern-oriented software architecture: a system of patterns**. 1. ed. River Street, NJ: John Wiley & Sons, 2000.

GAMMA, E. *et al.* **Design patterns: elements of reusable object-oriented software**. 1. ed. Boston: Addison-Wesley, 1994.

LARMAN, C. **Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development**. 3. ed. Upper Saddle River, NJ: Prentice Hall, 2004.

MARTIN, R.C. **Clean architecture: a craftsman's guide to software structure and design**. 1. ed. Upper Saddle River, NJ: Prentice Hall, 2017.

EXPLORE+

Indicamos a leitura dos livros:

Patterns of enterprise application architecture, de Martin Fowler. O livro apresenta padrões para módulos de interface com o usuário, persistência de dados, lógica de negócio e integração de sistemas.

Java EE 8 design patterns and best practices, de Rhuan Rocha e João Purificação. O livro aborda padrões para interface com o usuário, lógica do negócio, integração de sistemas, orientação a aspectos, programação reativa e microserviços.

Além de livros-texto, é possível encontrar muitos artigos técnicos por meio de buscas na internet, usando os acrônimos dos tipos de padrões e princípios abordados no conteúdo (GoF, GRASP, SOLID) ou os nomes específicos dos padrões e princípios que pretenda utilizar, aproveitando soluções adotadas com sucesso por outras pessoas.

CONTEUDISTA

Alexandre Luís Correa

 **CURRÍCULO LATTES**