

Padrões GoF Comportamentais

Prof. Alexandre Luis Correa, Prof. Denis Cople

Apresentação

O grupo de padrões GoF comportamentais engloba Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method e Visitor. São padrões ligados a algoritmos e à atribuição de responsabilidades em projetos orientados a objetos. Buscaremos identificar oportunidades para aplicá-los. Precisamos compreender essa família de padrões e saber como utilizá-los, pois, caso contrário, nossas soluções podem se tornar pouco flexíveis e de difícil manutenção.

Propósito

Compreender os padrões ligados a algoritmos e à atribuição de responsabilidades em projetos orientados a objetos, bem como identificar oportunidades para a sua aplicação são habilidades importantes para um projetista de software, pois, sem elas, as soluções geradas podem ser pouco flexíveis e dificultar a evolução de sistemas de software em prazo e custo aceitáveis.

Preparação

Antes de iniciar o conteúdo, é recomendado instalar 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 **Free Student License for Astah UML**, usado nos

exemplos deste estudo. Os arquivos Astah com diagramas UML utilizados neste conteúdo estão disponíveis para [download](#).

Recomendamos, também, a instalação de um ambiente de programação em Java, como o **Apache Netbeans**. Porém, antes de instalar o Netbeans, é necessário ter instalado o JDK (Java Development Kit) referente à edição Java SE (Standard Edition), que pode ser encontrado no site da Oracle Technology Network: **Java SE - Downloads | Oracle Technology Network | Oracle**.

Objetivos

Módulo 1

Padrões de projeto comportamentais Chain of Responsibility, Command e Iterator

Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Chain of Responsibility, Command e Iterator.

Módulo 2

Padrões de projeto comportamentais Mediator, Memento e Strategy

Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Mediator, Memento e Strategy.

Módulo 3

Padrões de projeto comportamentais Observer, Visitor e State

Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Observer, Visitor e State.

Padrões de projeto comportamentais Interpreter e Template Method

Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Interpreter e Template Method.

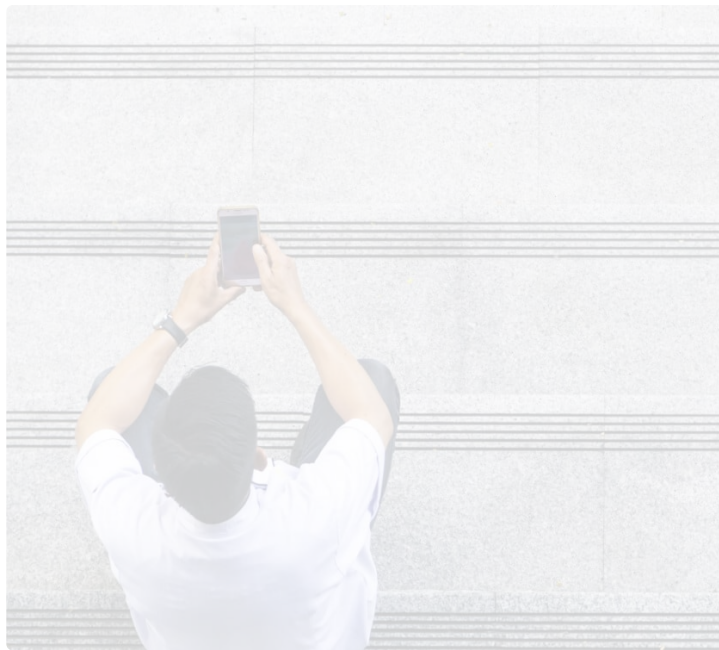


Introdução

Neste vídeo, conheça os padrões GOF comportamentais. Esses padrões são um conjunto de design patterns que lidam com a comunicação entre objetos e como os objetos interagem e se comunicam.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.





1 - Padrões de projeto comportamentais Chain of Responsibility, Command e Iterator

Ao final deste módulo, você será capaz de reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Chain of Responsibility, Command e Iterator.

Padrão Chain of Responsibility

Não são raras as situações em que uma requisição precisa passar por diferentes tratamentos, obedecendo a uma sequência específica, como em uma linha de produção. Por exemplo, ao tratar uma requisição no protocolo HTTP, pode ser necessário decriptar, descompactar e definir a codificação, antes que a informação possa ser tratada. E uma boa estratégia seria definir uma sequência de agentes para as transformações, cada um recebendo a requisição, executando sua parte, e repassando para o próximo agente. Esse é o modelo adotado no Chain of Responsibility, e aqui veremos como ele pode ser útil em nossos projetos.

Assista ao vídeo para compreender a aplicação do padrão de projeto Chain of Responsibility.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Intenção do padrão Chain of Responsibility

Suponha que você esteja em uma grande loja de departamentos querendo comprar uma televisão. O vendedor apresenta as condições para a compra, e você pede um desconto. Se o desconto estiver dentro do limite do vendedor, ele mesmo poderá aprová-lo. Caso contrário, o vendedor pedirá para você aguardar, pois ele terá que solicitar a autorização do supervisor. Se estiver no limite permitido para o supervisor, o desconto será aprovado, caso contrário, este terá que levar o pedido até o gerente, que poderá aprová-lo ou não. Ao final desse processo, o vendedor voltará para comunicar o resultado, ou seja, se o desconto foi autorizado.



Vendedor mostrando opções de televisores para um casal em uma loja de eletrônicos.

Perceba que você falou apenas com o vendedor, mas, a partir da sua requisição, uma cadeia de responsabilidade de aprovação de desconto foi acionada até a decisão final ser tomada.

A solução proposta pelo padrão Chain of Responsibility é inspirada nessa abordagem da loja de departamentos. Em vez de um módulo concentrar a responsabilidade de conhecer todos os objetos ou métodos participantes da solução, o processamento é dividido em pequenos módulos que podem ser combinados de diferentes maneiras. A ideia é construir uma cadeia de objetos, na qual cada objeto pode fazer algum processamento com a requisição ou simplesmente repassá-la para o seu sucessor na cadeia. O diagrama de classes a seguir apresenta a estrutura da solução proposta pelo padrão.

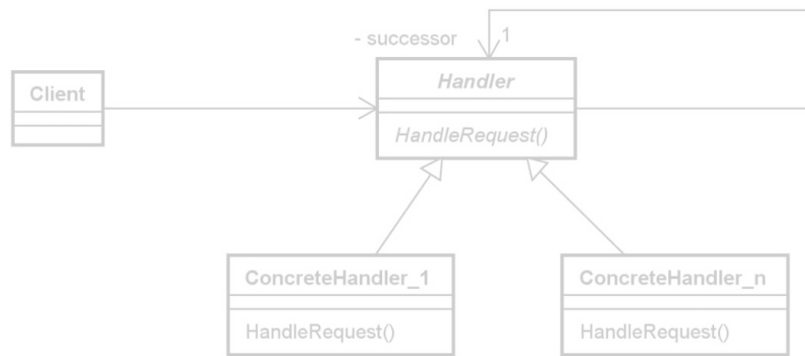


Diagrama de classes.

O participante **Client** representa um módulo cliente responsável pelo envio de uma requisição, que será tratada por uma cadeia de objetos descendentes do participante abstrato **Handler**. Cada elemento da cadeia, representada pelos participantes do tipo **ConcreteHandler**, pode tratar a requisição ou repassá-la para o seu sucessor, definido pelo autorrelacionamento presente no participante **Handler**. Portanto, todo objeto **ConcreteHandler** tem um sucessor, que é uma instância de uma classe descendente de **Handler**.

Os objetos **ConcreteHandler** formam a cadeia ilustrada no diagrama de sequência a seguir, em que a chamada do objeto **Client** à operação **HandleRequest** do primeiro **ConcreteHandler** pode ser encaminhada para o seu sucessor na cadeia, por meio de uma chamada à operação de mesmo nome.

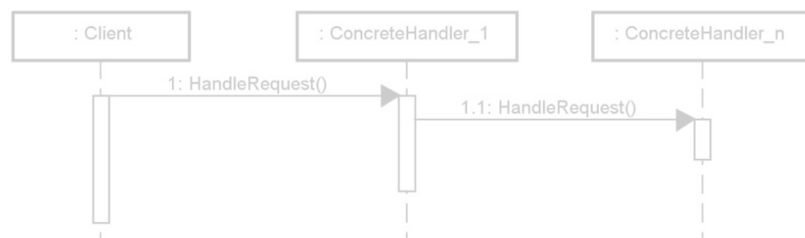


Diagrama de sequência.

Você se lembra como funciona o tratamento de exceções em Java? As chamadas de métodos em um instante de execução de um programa formam uma cadeia, também conhecida por pilha de execução. Quando um bloco `catch` em um método captura uma exceção, ele pode:



Fazer um tratamento local da exceção e encerrar o fluxo de exceção.



Fazer um tratamento local da exceção e repassá-la para o método chamador ou bloco try-catch mais externo.



Somente repassar a exceção para o método chamador ou bloco try-catch mais externo, sem fazer nenhum tratamento local.

A cadeia de objetos que forma a estrutura de solução desse padrão funciona de forma similar ao tratamento de exceções em Java, pois cada Handler pode fazer um processamento local da requisição e/ou repassá-la para o seu sucessor.

Agora, vamos reestruturar a solução apresentada anteriormente para o problema da tarifação de ligações, aplicando o padrão **Chain of Responsibility**. Primeiro, vamos definir a classe abstrata

TarifadorLigacao correspondente ao participante Handler do padrão. Ela possui um atributo correspondente ao seu sucessor e à operação tarifar, que repassa a chamada para o objeto sucessor. Note que a operação tarifar corresponde à operação HandleRequest definida na estrutura do padrão.

Java



Em seguida, vamos implementar uma classe tarifador para cada tipo de ligação. Cada tarifador específico corresponde ao participante

Consequências e padrões relacionados ao padrão Chain of Responsibility

O padrão Chain of Responsibility reduz a complexidade de uma classe que tenha que lidar com várias possibilidades de tratamento de uma requisição, transformando as diversas operações e estruturas condicionais complexas originalmente existentes em um conjunto de objetos interconectados, que podem ser combinados de diferentes formas, gerando uma solução menos acoplada e mais flexível. Por outro lado, existe o risco de uma requisição não ser respondida de forma adequada, caso a configuração da cadeia não seja corretamente realizada.

Atenção

Esse padrão é frequentemente utilizado em conjunto com o padrão Composite. Nesse caso, não é necessário implementar um sucessor, dado que podemos utilizar o relacionamento entre o agregado e as suas partes para encadear as chamadas pelos elementos da estrutura de composição.

Atividade 1

Assinale a alternativa que expressa a intenção do padrão de projeto Chain of Responsibility:

A

Reduzir o acoplamento entre o objeto que envia uma requisição e todos os possíveis objetos que podem realizar algum processamento relacionado a essa requisição.

inteiros, execução e retorno do resultado inteiro, e derivar classes concretas para cada operação? O padrão Command visa esse tipo de modelagem, encapsulando os algoritmos em uma família de classes, o que facilita muito a evolução de nossos sistemas.

Neste vídeo, você verá o padrão de projeto Command, que encapsula uma requisição em um objeto.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Intenção do padrão Command

Command é um padrão que encapsula uma requisição em um objeto. Em projetos que não utilizam esse padrão, uma requisição é normalmente realizada por meio de uma simples chamada de operação. O encapsulamento de requisições em objetos desacopla o requisitante e o objeto executor, o que possibilita:

1. Parametrizar as requisições disparadas pelos clientes;
2. Criar filas de requisições;
3. Registrar o histórico de requisições;
4. Implementar operações para desfazer (undo) ou refazer (redo) o processamento realizado para atender uma requisição.

Problema resolvido pelo padrão Command

Imagine que você esteja desenvolvendo um jogo no qual as ações associadas às teclas ou aos botões do mouse possam ser configuradas. Veja no quadro a seguir um exemplo de configuração.

Evento	Ação
Tecla W	ir para frente
Tecla S	Ir para trás
Tecla espaço	Pular

Evento	Ação
Botão 4 do mouse	Pular

Quadro: Configuração de teclas.
Elaborado por: Alexandre Luis Correa.

O módulo de configuração do jogo deve permitir que o usuário defina a correspondência desejada entre os eventos e as respectivas ações. Poderíamos resolver esse problema definindo constantes equivalentes a todos os eventos e ações do programa e, a partir dessas constantes, estabelecer uma configuração, correlacionando os tipos de eventos com os códigos das ações.

O código a seguir apresenta o esqueleto dessa solução, em que a operação **tratarEvento** é executada sempre que um evento de interface ocorrer. Essa operação obtém o código da ação associada ao evento e invoca a operação correspondente do jogo.

Java



Essa solução cria um acoplamento entre o elemento que captura o evento disparado pela interface com o usuário (InterfaceJogo) e o módulo que realiza as operações em resposta ao Jogo, além de conter uma estrutura condicional não extensível. Imagine que o jogo tenha cinquenta comandos.

Consegue visualizar como essa estrutura condicional definida pelo comando switch ficaria enorme?

Solução do padrão Command

A solução proposta pelo padrão Command consiste em transformar cada ação em um objeto. Portanto, em vez de as ações serem implementadas como operações de uma classe, cada ação é implementada individualmente em uma classe. O diagrama a seguir apresenta a estrutura do padrão.

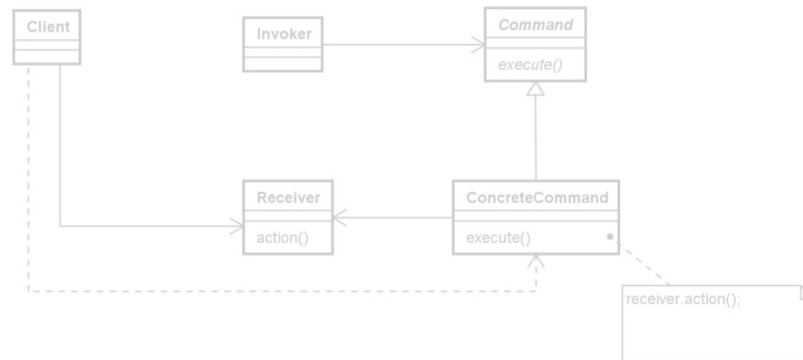


Diagrama de classes.

Cada ação é definida em uma classe correspondente ao participante **ConcreteCommand**, que implementa a interface abstrata **Command**, na qual a operação genérica `execute` está definida. O participante **Client** corresponde a um módulo cliente responsável por fazer a instanciação e a associação de cada comando ao respectivo elemento **Receiver**, módulo que efetivamente realizará as ações associadas à requisição. O participante **Invoker** corresponde ao elemento que dispara o comando, como um botão em uma interface gráfica, por exemplo.

Veja, a seguir, a estrutura do código para a configuração dos eventos do jogo com a aplicação desse padrão. Primeiro, criamos a interface genérica **Comando** e os comandos concretos correspondentes às ações do jogo.

Java



Atividade 2

Um determinado sistema de gerenciamento de produtos apresenta as funções de compra, venda, movimentação e descarte, com impacto direto sobre os estoques dos armazéns e no controle logístico. Para que o desenvolvimento ficasse mais organizado, foi especificado o padrão Command, no qual o método executar recebe um produto como parâmetro e retorna um booleano, indicando o sucesso ou falha da operação. Isso significa que, ao desenvolver o código:

- A Todas as operações devem ser concentradas em uma única classe, mediadora dos processos.
- B Devem ser definidas todas as operações necessárias, não sendo possível acrescentar novas funcionalidades posteriormente.
- C Uma classe abstrata ou interface deve definir o método executar, abstrato, e cada operação será implementada em uma classe diferente, com a sobrescrita do método.
- D Deverá ser avaliado se o melhor padrão seria Command ou Chain of Responsibility, pois não é possível combiná-los.
- E Não será possível definir fluxos de execução que combinem diferentes operações, ou comandos, causando um alto nível de replicação de código devido ao comportamento monolítico.

O participante **Aggregate** representa uma coleção genérica cujos elementos podem ser percorridos sequencialmente pelo conjunto de operações (First, Next, IsDone e CurrentItem) definido pelo participante Iterator. O participante ConcreteAggregate representa uma coleção específica, que é responsável por criar elementos do tipo ConcreteIterator capazes de percorrê-la.

O framework de estrutura de dados da linguagem Java implementa esse padrão. Coleções como ArrayList, LinkedList, HashSet e TreeSet são descendentes da classe genérica **Collection**. Nesse caso, as coleções específicas correspondem ao participante **ConcreteAggregate**, enquanto a classe Collection corresponde ao participante Aggregate. Em Java, a interface genérica Iterator define um conjunto de operações um pouco diferente daquele definido na estrutura do padrão:

hasNext

Verifica se existe um próximo elemento ou se o cursor já está posicionado no último elemento da coleção.

next

Retorna o próximo elemento da coleção. Na primeira chamada, ele retorna o primeiro elemento da coleção.

remove

Remove um elemento da coleção.

Cada coleção define uma operação Iterator que retorna um objeto ConcreteIterator capaz de percorrê-la. O diagrama de classes a seguir ilustra os iteradores correspondentes às coleções ArrayList (ArrayListIterator), LinkedList (ListIterator), HashSet (KeyIterator) e TreeSet (ValueIterator).

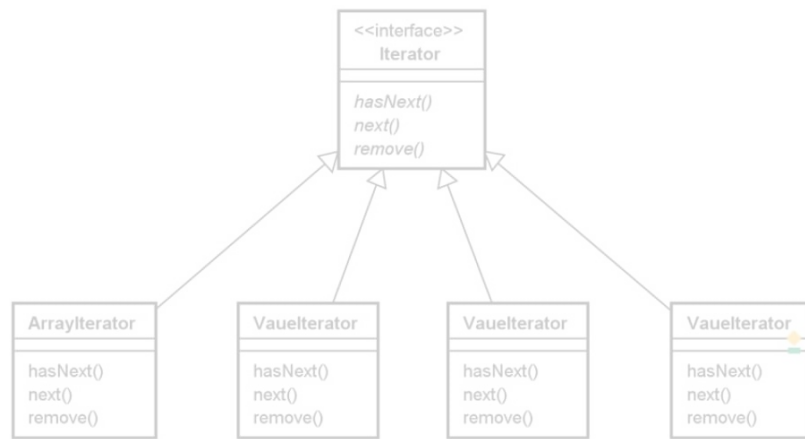


Diagrama de classes.

Veja, a seguir, um exemplo de utilização desse padrão em Java. O método **removerItensSemEstoque** recebe uma coleção de produtos. Ele solicita um **Iterator** à coleção e utiliza as operações **hasNext** e **next** para percorrê-la, removendo os produtos cuja quantidade em estoque for zero.

Java



Note que esse método funciona com qualquer coleção, isto é, **ArrayList**, **LinkedList**, **HashSet** ou **TreeSet**.

Consequências e padrões relacionados ao padrão **Iterator**

O principal benefício do padrão **Iterator** é permitir que os módulos clientes possam percorrer sequencialmente os elementos de uma coleção de forma independente da sua representação interna. Outro benefício é a possibilidade de haver diversas instâncias de **Iterator**

tarefas específicas, mas de forma que não sejam feitas chamadas diretas entre eles, evitando aumentar o acoplamento. Por exemplo, um processo de compra na Web envolve a utilização de vários módulos, nos quais o carrinho precisa de informações do estoque e o pagamento demanda ações logísticas. Para impedir que esses módulos se comuniquem diretamente, as chamadas podem ser feitas via classe Mediator.

Este vídeo apresenta o padrão de projeto Mediator, que permite encapsular a forma de interação entre um conjunto de objetos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Intenção do padrão Mediator

O padrão Mediator encapsula a forma de interação entre um conjunto de objetos, com o objetivo de evitar que eles tenham que referenciar uns aos outros explicitamente.

Problema resolvido pelo padrão Mediator

Em um sistema de comércio eletrônico, quando o cliente efetua o pagamento, a compra deve ser confirmada, o processo de logística de entrega deve ser disparado e um e-mail de confirmação do pedido deve ser enviado para o cliente.

Imagine que um desenvolvedor tenha dado a solução esquematizada no diagrama de sequência a seguir.

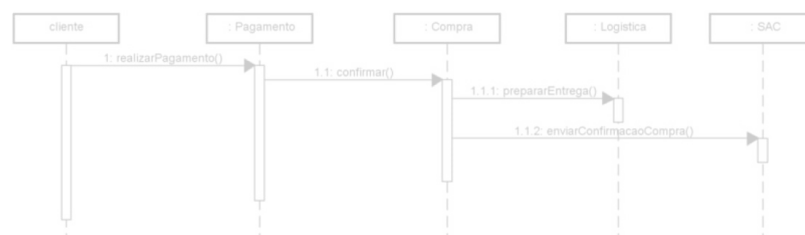


Diagrama de sequência.

Nessa solução, o módulo **Pagamento**, após realizar o pagamento da compra, chama a operação **confirmar** do módulo **Compra**, que, por sua vez, chama a operação **prepararEntrega** do módulo **Logística** e a operação **enviarConfirmacaoCompra** do módulo **SAC**, responsável por enviar um e-mail para o usuário com os dados da compra.

Você consegue visualizar o alto acoplamento entre as classes na realização do processo de fechamento da compra? Imagine que você precise inserir uma etapa nesse processo, como a baixa no estoque, por exemplo. Você teria que adicionar uma dependência no módulo Compra ou no módulo Logística, que ficaria responsável por chamar uma operação do módulo Estoque.

Como simplificar interações complexas entre os objetos, com o objetivo de reduzir o acoplamento entre eles e permitir a criação de novas interações sem que esses objetos precisem ser alterados? Essa é a essência do problema que o padrão Mediator visa solucionar.

Solução do padrão Mediator

A estrutura da solução proposta pelo padrão Mediator está representada no diagrama de classes a seguir:

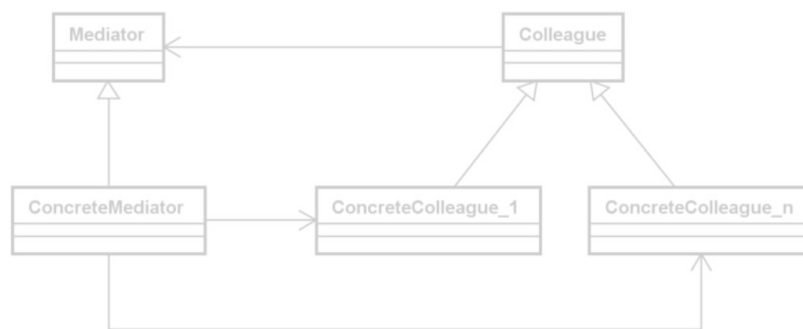


Diagrama de classes.

A solução consiste em centralizar as comunicações entre os objetos em um elemento denominado mediador. O participante **Mediator** define uma interface padrão que cada objeto utilizará para se comunicar com os demais envolvidos em uma interação. Esses objetos formam a hierarquia definida pelo participante **Colleague**. O participante **ConcreteMediator**, por sua vez, corresponde a um elemento concreto que mantém referências para todos os objetos cujas interações ele precisará mediar.

Vamos modificar a solução do problema do comércio eletrônico aplicando o padrão Mediator. As classes Pagamento, Compra, Logística e SAC correspondem ao participante **ConcreteColleague** e são especializações da classe genérica **Colleague**, na qual está definida uma referência ao objeto mediador. Note que a classe Pagamento notifica o mediador de que o pagamento foi concluído. Da mesma forma, a classe Compra notifica o mediador de que a compra foi encerrada. Portanto, esses objetos passam a se comunicar apenas com o mediador. Os

módulos Pagamento e Conta apenas notificam o evento que ocorreu, ficando a cargo do mediador definir o encaminhamento que deve ser dado a cada evento.

Java



A implementação do mediador é ilustrada esquematicamente a seguir. A interface **MediadorCompra** define todas as notificações que os componentes participantes da interação podem enviar para o objeto mediador. Essa interface é implementada pela classe **MediadorCompraSimples**, que representa uma implementação simples das interações entre os objetos. Note que outras classes de mediação de compra podem ser implementadas, representando diferentes processos de interação entre os participantes.

A classe **MediadorCompraSimples** corresponde ao participante **ConcreteMediator**. Ela possui uma referência para cada objeto participante das interações (pagamento, compra, logística e sac), recebe os eventos enviados por cada participante e dispara a execução de operações em resposta a cada evento.

Java



Atividade 1

O sistema de uma concessionária de veículos apresenta os seguintes módulos de mensagens: oficina, faturamento e venda de veículos. A cada venda ou serviço de oficina, deve ser gerado um faturamento e enviada uma mensagem para o operador do caixa. O padrão Mediator será adotado para diminuir o acoplamento entre os módulos. Para a correta implementação do padrão, será necessário:

- A Definir uma sequência de operações por onde passará a requisição, de forma que, ao final de cada operação, seja invocada a operação seguinte.
- B Instanciar os dados transacionais no módulo de mensagens, e fazer com que os demais módulos assinem o primeiro, para que sejam avisados de qualquer mudança ocorrida nos dados.
- C Definir um padrão para os formatos de entrada e saída e implementar o algoritmo de cada operação em uma classe diferente.
- D Criar uma classe gestora que centraliza o fluxo de operação, ou diálogo entre as partes, e substituir as chamadas diretas entre módulos por notificações para esse gestor.
- E Armazenar o estado das variáveis a cada operação efetuada, de forma a permitir a recuperação de um estado anterior.

Parabéns! A alternativa D está correta.

O padrão Mediator envolve a definição de um mediador do processo, ou gestor, de forma que, em vez de um módulo se comunicar diretamente com o outro, gere uma notificação para o mediador para que ele se comunique com o segundo módulo. As outras opções definem padrões diferentes, como o Chain of Responsibility, com uma requisição passando por uma sequência de operações, Observer, que notifica os clientes de qualquer modificação ocorrida em seus dados, Command, que define um formato padrão de operação e encapsula os algoritmos em classes, e Memento, que armazena o estado a cada operação executada, permitindo desfazer operações, com a recuperação de um estado anterior.

Padrão Memento

Não são raras as situações em que precisamos desfazer algo, motivo pelo qual todos adoram a combinação de teclas CTRL e Z, presente em diversos aplicativos. O padrão Memento tem como objetivo viabilizar esse tipo de comportamento, empilhando instantâneos do estado do objeto, de forma a permitir a recuperação dos estados anteriores. Como um exemplo prático, supondo que você precise definir um aplicativo de modelagem tridimensional, o Memento permitiria ao profissional de modelagem desfazer suas últimas operações, uma funcionalidade essencial para dar maior liberdade e flexibilidade ao processo criativo.

Assista ao vídeo para compreender o padrão de projeto Memento, que permite capturar o estado interno de um objeto, sem quebrar o seu encapsulamento.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Intenção do padrão Memento

Memento é um padrão que permite capturar o estado interno de um objeto, sem quebrar o seu encapsulamento, de forma que esse estado possa ser restaurado posteriormente.

O código anterior ilustra duas operações adicionadas à classe Pedido original:

Operação criarMemento

Salva o estado do pedido em uma instância de PedidoMemento.

Operação restaurarMemento

Restaura o estado do pedido a partir do objeto PedidoMemento recebido como parâmetro.

Veja no código a seguir um exemplo de implementação de uma operação undo utilizando um objeto da classe Pedido. Note que esse programa apenas guarda uma referência para o memento criado e retornado pelo objeto pedido, repassando-o quando for necessário restaurar o estado desse pedido. Em nenhum momento, o programa exemplo chama operações do objeto memento.

Java



Consequências e padrões relacionados ao padrão Memento

O padrão Memento facilita a implementação de problemas nos quais precisamos desfazer certas modificações de estado em objetos decorrentes da execução de operações ou implementar algum

mecanismo de checkpoint/restart, em que interrompemos o processamento para retomá-lo posteriormente do ponto onde paramos.

A implementação de um memento pode ser custosa em situações nas quais exista uma grande quantidade de informações para armazenar e posteriormente restaurar, especialmente quando envolver um objeto que tenha uma grande rede de objetos relacionados. Além disso, há peculiaridades na implementação do padrão Memento em algumas linguagens. A seguir, temos dois exemplos:

		
Java	×	C++
Dificulta a definição da interface do memento, de forma que somente o originador tenha acesso.		Permite uma definição mais rigorosa por meio da utilização da palavra reservada <i>friend</i> .

Esse padrão é frequentemente utilizado com o padrão Command, quando este implementar um mecanismo para desfazer um comando (undo), pois, para isso, é necessário guardar o estado anterior à sua execução, o que pode ser feito com o uso do padrão Memento.

Atividade 2

Assinale a alternativa que expressa a intenção do padrão de projeto Memento:

- ☒ A Permitir a utilização mais racional de memória, por meio do compartilhamento de objetos.

- B Interceptar o momento em que uma chamada a um objeto é executada, permitindo a execução de operações como log, auditoria, autorização, entre outras.
- C Fornecer uma interface de alto nível para um subsistema ou componente.
- D Permitir a utilização de diferentes implementações de um serviço fornecida por terceiros e que não podem ser modificadas, por meio da definição de uma interface comum e de elementos que fazem a tradução dessa interface comum para as interfaces específicas fornecidas pelos terceiros.
- E Fornecer um mecanismo para salvar e restaurar o estado de um objeto, sem quebrar o seu encapsulamento.

Parabéns! A alternativa E está correta.

As alternativas A, B, C e D descrevem a intenção dos padrões Flyweight, Proxy, Facade e Adapter, respectivamente.

Padrão Strategy

O comportamento estratégico envolve uma tomada de decisão a partir da análise de condições ambientais que favoreçam o melhor conjunto de operações. Será por meio do padrão Strategy que iremos tornar nossos sistemas mais assertivos, agrupando um conjunto de operações em cada classe de estratégia, e invocando a classe que mais se adequa ao contexto de execução do aplicativo. Por exemplo, um sistema de detecção de vírus pode decidir entre manter, remover ou colocar um arquivo em quarentena, tendo como base a análise de código efetuada por um módulo de inteligência artificial.

Assista ao vídeo para compreender o padrão de projeto Strategy, que permite definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Intenção do padrão Strategy

O padrão Strategy define uma família de algoritmos, encapsulando-os em objetos e permitindo que eles possam ser utilizados de forma intercambiável, ou seja, o algoritmo específico pode ser trocado sem que o módulo usuário desse algoritmo precise ser alterado.

Problema resolvido pelo padrão Strategy

O padrão Strategy é aplicável em situações nas quais existam diferentes algoritmos para gerar determinado resultado. Um exemplo desse tipo de situação é o cálculo dos juros de um título público. Você sabia que esse é um conhecimento fundamental para quem trabalha no mercado financeiro?

Existem diferentes métodos para calcular os juros de um título para negociação em determinada data. Alguns exemplos de métodos são:



Regressão linear múltipla



Bootstrap



Interpolação polinomial splines cúbicos

Imagine que você esteja desenvolvendo um módulo que calcula a taxa de juros de um título para negociação no dia seguinte. Uma solução frequentemente encontrada consiste em concentrar toda a lógica de

cálculo em um único módulo, como ilustrado pela estrutura de código apresentada a seguir. É importante observar que esta é uma estrutura bastante simplificada da implementação real do problema, pois abstraímos a complexidade matemática envolvida.

Java



Esse tipo de solução possui dois problemas. Você já identificou quais são eles? Vejamos:



Problema 1

Para adicionar novos algoritmos, temos que abrir o módulo `TituloPublico` para adicionar um novo método e um novo tipo ao `switch/case` do método `taxaJuros`, violando o princípio `Open Closed`, um dos princípios `SOLID` de projeto.



Problema 2

Não é possível reutilizar o algoritmo para cálculo de outros valores que não sejam taxas de juros. Esses algoritmos são métodos matemáticos aplicáveis a outros problemas. Como podemos separar os algoritmos das classes de domínio em que eles são aplicados?

Um problema similar ao problema 2 ocorre com os algoritmos de ordenação de dados, pois existem diferentes métodos de ordenação (Bubble Sort, Quick Sort, Merge Sort, Heap Sort, por exemplo) aplicáveis a diferentes tipos de dados (produtos, pessoas, números etc.).

Solução do padrão Strategy

A estrutura da solução proposta pelo padrão Strategy está representada no diagrama de classes a seguir.

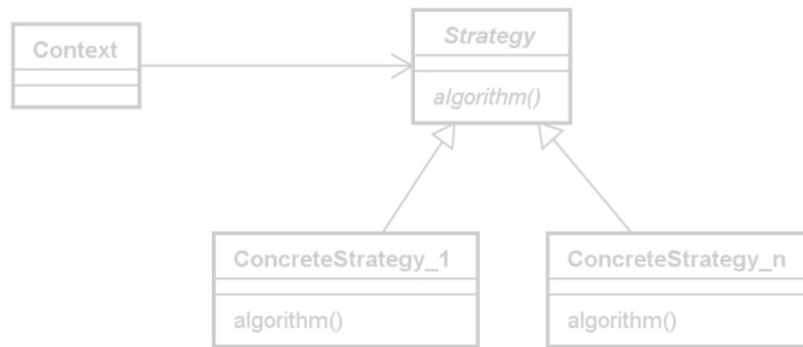


Diagrama de classes.

A ideia central consiste em separar cada algoritmo em uma classe, fazendo com que todas as classes específicas implementem uma interface comum, representada pelo participante **Strategy**. O participante **Context** define o contexto que proverá o algoritmo com os dados necessários para o processamento. Note que o contexto mantém uma referência para a interface genérica, não dependendo de qualquer implementação específica. Com isso, podemos adicionar novos algoritmos sem precisarmos modificar a classe que define o contexto para a sua aplicação.

O código a seguir mostra como podemos implementar o problema do cálculo de juros de um título com a aplicação desse padrão. A interface **CalculadoraJuros** corresponde ao participante **Strategy** e define uma interface genérica para o cálculo da taxa a partir do título, da data para a qual a taxa será calculada e dos pontos da curva. Cada algoritmo é definido em uma classe que implementa essa interface, correspondendo ao participante **ConcreteStrategy** definido na estrutura do padrão.

Java



Perceba que os algoritmos, antes implementados em operações de uma única classe, foram transformados em classes, todas implementando o mesmo conjunto de operações. A classe **CurvaJuros** passou a receber o algoritmo de cálculo como parâmetro, isto é, o algoritmo a ser utilizado passou a ser injetado dinamicamente no objeto, permitindo que ele trabalhe com qualquer algoritmo que implemente a interface genérica **CalculadoraJuros**.

Portanto, a classe CurvaJuros corresponde ao participante Context do padrão, pois ela estabelece o contexto com os dados necessários para o processamento do algoritmo. É importante observar que essa injeção pode ser feita a cada chamada da operação taxaJuros, como mostrado no exemplo, ou apenas no construtor da classe CurvaJuros, o que faria com que cada instância de curva sempre trabalhasse com o algoritmo configurado na sua criação.

Java



Veja no código a seguir como um módulo relacionado à classe CurvaJuros pode solicitar o cálculo da taxa, passando o algoritmo desejado para a sua execução.



Consequências e padrões relacionados ao padrão Strategy

O padrão Strategy oferece algumas vantagens:

Definição de família de algoritmos

Permite a definição de uma família de algoritmos, que podem ser utilizados e configurados de forma flexível.

Passos comuns na superclasse

Permite a implementação dos passos em comum dos diferentes algoritmos na superclasse, evitando a duplicação de código.

Simplificação das estruturas

Evita a criação de estruturas condicionais complexas no contexto da aplicação dos algoritmos, substituindo comandos switch/case por chamadas polimórficas de operações.

Por outro lado, o padrão Strategy expõe as diferentes opções de algoritmo para os clientes. Portanto, o uso desse padrão é mais indicado para as situações nas quais o cliente conheça e precise escolher o algoritmo mais apropriado.

O algoritmo a ser utilizado pode ser parametrizado em uma configuração da aplicação, utilizando-se um padrão de criação (Factory Method ou Abstract Factory), ou ainda o recurso de injeção de

E

Adicionar novas funcionalidades a um objeto, por meio da composição aninhada de objetos.

Parabéns! A alternativa B está correta.

As alternativas A, C, D e E estão relacionadas à intenção dos padrões Proxy, Observer, Flyweight e Decorator, respectivamente.



3 - Padrões de projeto comportamentais Observer, Visitor e State

Ao final deste módulo, você será capaz de reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Observer, Visitor e State.

Padrão Observer

Atualmente o padrão Observer é utilizado em qualquer plataforma de desenvolvimento, como no relacionamento entre modelo e visualização do Android, ou nas chamadas HTTP assíncronas do Angular via RxJS.

Segundo o padrão Observer, uma fonte de dados pode ser assinada por um conjunto de observadores, e qualquer modificação nos dados irá gerar uma notificação para os assinantes, permitindo a atualização de elementos associados em tempo real. Como exemplo simples, poderíamos construir uma planilha de gastos, em que um campo com o



ConcreteSubject

O participante ConcreteSubject corresponde a um elemento específico da aplicação que está sendo construída cujo estado, representado pelo atributo subjectState, é do interesse de um conjunto de observadores que serão notificados quando esse estado mudar.



ConcreteObserver

O participante ConcreteObserver mantém uma referência para o objeto ConcreteSubject, armazenando ou apresentando dados, representados pelo atributo observerState, que devem se manter consistentes com o estado desse objeto. Ele implementa a interface de recebimento de notificação enviada pelo Subject (operação update), sendo responsável por obter o novo estado do ConcreteSubject, por meio das operações representadas pela operação GetState do participante Subject.

O diagrama de sequência a seguir ilustra as interações entre os participantes da solução. Inicialmente, os objetos observadores devem se registrar no objeto Subject, por meio da operação attach. O estado de um objeto ConcreteSubject pode ser alterado, por meio das suas operações modificadoras, representadas genericamente pela operação setState. A implementação dessas operações modificadoras deve chamar a operação notify, definida para todo objeto do tipo Subject. A operação notify, por sua vez, deve invocar a operação update de todos os objetos observers registrados previamente. A implementação da operação update em cada ConcreteObserver deve obter o novo estado do objeto ConcreteSubject, invocando as operações de consulta representadas pela operação getState.

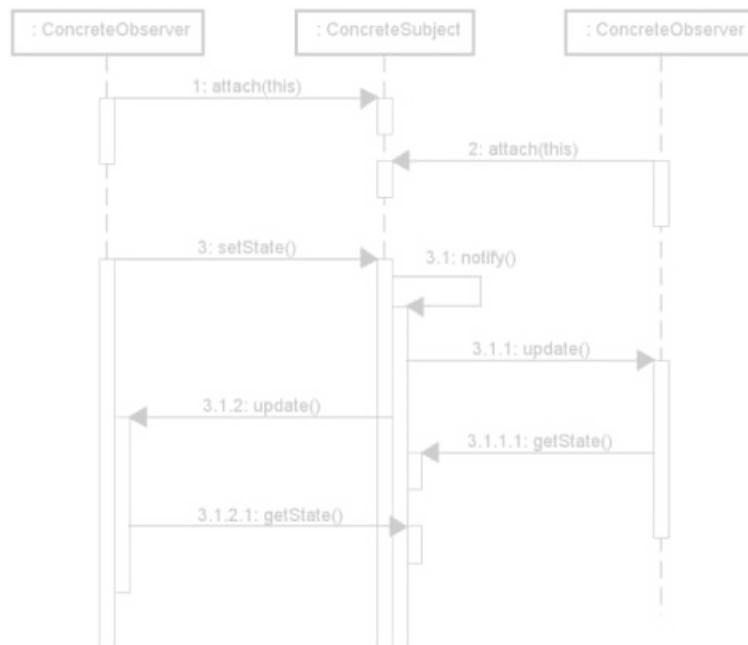


Diagrama de sequência.

Em Java, esse padrão pode ser implementado utilizando as classes disponíveis nas bibliotecas da linguagem. Com o Java 8, uma das implementações possíveis consiste em definir as classes notificadoras como subclasses de `java.util.Observable`, enquanto os observadores devem implementar a interface `java.util.Observer`.

O código a seguir apresenta um exemplo de implementação desse padrão. A classe `Ponto` desempenha o papel de `Subject`. Nesse caso, `Ponto` é uma subclasse de `Observable`, herdando a implementação da gestão da lista de observadores e o método para notificação. Observe que todas as mudanças relevantes, presentes nos métodos `setX` e `setY`, chamam duas operações da superclasse `Observable`: `setChanged` e `notifyObservers`, que notificam todos os objetos observadores de que houve uma mudança no valor de um atributo.

Java



No código, é possível perceber que o método `Observe` recebe um método para atualizar a caixa de texto a partir da modificação do valor corrente do `LiveData`, logo, corresponde ao sistema de assinatura e notificação do padrão `Observer`. Quanto às demais opções, `TextField` atua como observador; `LiveData` é o dado observável; apenas o valor do `TextField` é alterado no código apresentado, e não há qualquer impedimento para atualizar vários componentes visuais ao definir a resposta na chamada para o método `observe`.

Padrão Visitor

Segundo a definição do padrão `Visitor`, ele visa promover a separação entre uma família de objetos e os algoritmos que serão utilizados, permitindo que novas funcionalidades sejam definidas sem a necessidade de modificar os objetos. Por exemplo, você poderia definir uma interface `Visitante` que aceitasse planilhas, XML e JSON, implementando na classe `GeradorHTML`, que transformaria as informações em tabelas HTML, e na classe `GeradorPDF`, para gerar arquivos PDF a partir dos dados fornecidos, e, posteriormente, poderia adicionar a classe `Base64Transform`, que retornaria o conteúdo codificado em Base64, evoluindo o sistema sem afetar as funcionalidades definidas previamente.

Neste vídeo, compreenda a aplicação do padrão de projeto `Visitor`, que possibilita definir novas operações em uma hierarquia de objetos.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Intenção do padrão Visitor

O padrão `Visitor` permite a definição de novas operações em uma hierarquia de objetos sem que haja a necessidade de modificar as classes dessa hierarquia.

Problema resolvido pelo padrão Visitor



Padrão de projeto Visitor.

A solução consiste em separar os elementos que formam uma estrutura hierárquica e os algoritmos que realizam operações sobre essa estrutura em duas famílias de classes. Vamos analisar dois participantes da solução apresentada:

Visitor

O participante **Visitor** define uma família de operações, que podem aplicadas a cada um dos elementos concretos da estrutura. Para cada classe concreta da estrutura, correspondente ao participante **ConcreteElement**, deve ser definida uma operação visit< nome_da_classe>, que recebe como parâmetro um objeto da respectiva classe. Cada família de operações é definida em um **ConcreteVisitor** específico.

Element

O participante **Element** corresponde à classe mais genérica da estrutura hierárquica dos elementos, na qual é definida a operação accept, que recebe um visitor como parâmetro e chama a operação visitConcreteElement correspondente à sua classe, passando o próprio objeto que está sendo visitado como argumento.

O código a seguir, ilustra a implementação do problema das expressões aritméticas, utilizando o padrão Visitor. Para simplificar, vamos mostrar apenas somas de números inteiros. Cada número é uma instância da classe NumeroInteiro, enquanto cada operador de soma é uma instância

da classe OpSoma. Todo operador aritmético herda da classe OperadorAritmetico que define dois operandos, um à esquerda e o outro à direita do operador. Todo elemento de uma expressão, seja um número, seja um operador, implementa a operação accept definida na superclasse ElementoExpressao.

Java



A avaliação do resultado de uma expressão é definida em um Visitor separadamente da estrutura da expressão. A interface VisitorExpressaoAritmetica define uma operação visitor para cada elemento da estrutura, enquanto a classe VisitorCalculadora implementa as operações necessárias para calcular o valor de uma expressão.

Note que a navegação pelos elementos da estrutura é definida no visitor. Por exemplo, a operação soma precisa, em primeiro lugar, avaliar a expressão do operando à esquerda do operador, para depois avaliar a expressão à direita, e finalmente gerar o valor da expressão, somando o resultado das duas expressões.

Java



Praticar é a melhor forma de fixar o conteúdo. Nesse sentido, recomendamos as seguintes atividades:

- Implemente as operações de multiplicação, divisão e subtração, para o exemplo.
- Implemente outro visitor capaz de imprimir a expressão. Ex: $10 + 20$

Consequências e padrões relacionados ao Visitor

O padrão Visitor permite a adição de novas funcionalidades de forma ortogonal a uma estrutura de objetos. Como vimos no exemplo das expressões, diversas funcionalidades podem ser implementadas como um visitor. Veja alguns exemplos:



Cálculo



Formatação



Verificação sintática



Verificação semântica da expressão

Desse modo, a estrutura original de objetos não fica poluída com operações não relacionadas entre si. Entretanto, a adição de um novo elemento à estrutura de objetos afeta todos os visitors implementados, pois será necessário adicionar uma operação de visita em cada uma

em operações com diversas expressões condicionais, cada estado é representado em uma classe separada.

Problema resolvido pelo padrão State

O padrão State é muito útil para problemas envolvendo a implementação de entidades com uma dinâmica de estados relevante. Por exemplo, suponha um sistema de ponto de venda de uma loja. Um ponto de venda pode estar fechado, disponível para vendas ou com uma venda em curso. Eventos provocam a transição entre esses estados. Por exemplo, no início do dia, o caixa está fechado e ao receber o evento iniciar dia, ele passa para o estado disponível. Um mesmo evento pode levar para diferentes estados, como é o caso do evento iniciar sangria, que pode ocorrer quando o PDV está no estado Disponível ou Vendendo.



Diagrama de estado.

O código a seguir ilustra uma implementação convencional sem a utilização do padrão. Note como essa implementação é baseada em código condicional embasado no estado do PDV. Para uma máquina de estados mais complexa, com mais estados e eventos, esse tipo de solução torna o código muito complexo, difícil de testar e modificar.

Java



Solução do padrão State

A estrutura da solução proposta pelo padrão State é apresentada no diagrama UML a seguir. O participante Context corresponde a uma classe que possui uma dinâmica dependente de estados. Cada classe concreta ConcreteState implementa o comportamento da classe Context associado a um estado específico. A classe Context possui um atributo do tipo State, que corresponde ao estado corrente do objeto. Quando um objeto Context recebe uma requisição, a execução é passada para o estado corrente por meio da operação handle definida na interface State.

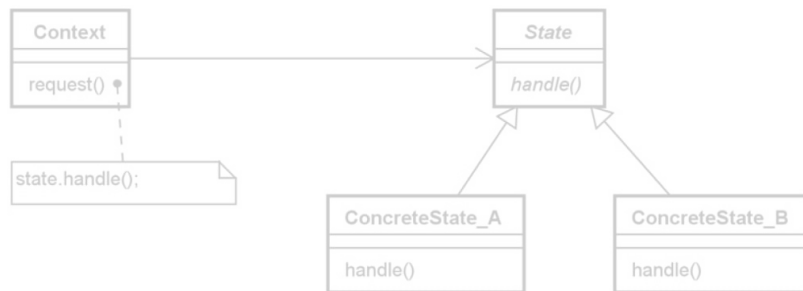


Diagrama de classes.

O exemplo a seguir mostra como o PDV descrito no problema pode ser implementado com o padrão. Essa é uma implementação parcial apenas para você entender a estrutura da solução. A classe PDV corresponde ao participante Context. Ele possui um atributo do tipo PDV_Estado, que corresponde ao participante State. PDV_Estado é uma interface que define todos os eventos que o contexto deve tratar. Cada evento é definido como uma operação distinta.

As classes PDV_Estado_Disponivel e PDV_EstadoVendendo são dois exemplos de classes concretas que correspondem a implementações distintas da interface genérica de eventos.

Java



Consequências e padrões relacionados ao State

O padrão State separa os comportamentos aplicáveis a cada estado de um objeto em classes distintas. Por outro lado, essa solução gera um número bem maior de classes, o que pode ser bom, especialmente quando existirem muitos estados e muitas operações que dependam desses estados.

Esse padrão também melhora a compreensão do código, pois além de eliminar código condicional extenso baseado no estado corrente, ele explicita as transições de estado.

Atenção

O padrão State pode ser combinado com o padrão Flyweight, permitindo o compartilhamento de objetos estados por muitos objetos contextuais, evitando a criação de um objeto estado para cada objeto contexto.

Atividade 3

Assinale a alternativa que expressa a intenção do padrão de projeto State:

- A Compor hierarquias de objetos, de modo que objetos individuais e agregados possam ser manipulados de forma genérica pelo mesmo conjunto de operações.
- B Permitir a composição do estado de um objeto complexo, a partir de diversos pequenos objetos imutáveis, por meio de uma solução baseada em compartilhamento.
- C

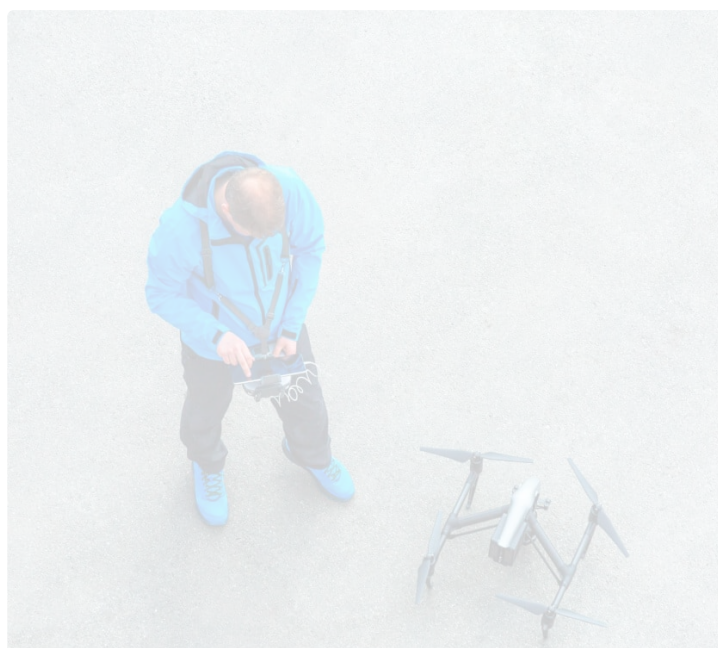
Permitir salvar e restaurar o estado de um objeto sem quebrar o seu encapsulamento.

- D Permitir a implementação de uma máquina de estados, por meio da definição de uma classe para cada estado. Cada classe implementa o comportamento do objeto naquele estado específico.

- E Instanciar um objeto de uma família de objetos relacionados, permitindo que o seu estado inicial seja definido a partir de uma configuração externa.

Parabéns! A alternativa D está correta.

A alternativa A corresponde ao propósito do padrão Composite. A alternativa B está incorreta porque nenhum padrão tem esse propósito. O padrão Flyweight permite o uso racional da memória, por meio de uma solução baseada em compartilhamento. A alternativa C corresponde ao propósito do padrão Memento. A alternativa E está incorreta porque o propósito descrito é próximo ao do padrão Abstract Factory, com a ressalva de que permitir que o seu estado seja definido a partir de uma configuração externa. não é um objetivo descrito pelo referido padrão.



4 - Padrões de projeto comportamentais Interpreter e Template Method

Ao final deste módulo, você será capaz de reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Interpreter e Template Method.

Padrão Interpreter

O padrão Interpreter, como seu nome sugere, define um interpretador de comandos, normalmente baseado em um conjunto de regras sintáticas.

Um dos melhores exemplos para adoção do padrão Interpreter é na compilação de expressões regulares, comuns na filtragem e identificação de conteúdo em sequências de texto. Outro exemplo seria os dados fornecidos por satélites meteorológicos, que são codificados na forma de texto, segundo regras bem definidas. O padrão Interpreter pode ser utilizado para controlar as regras de leitura desses dados e consequente extração das informações, permitindo alimentar outros sistemas, como na visualização de mapas meteorológicos.

Este vídeo apresenta a aplicação do padrão de projeto Interpreter, que possibilita definir uma representação para a gramática de uma linguagem.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Intenção do padrão Interpreter

O propósito do padrão Interpreter é definir uma representação para a gramática de uma linguagem e um módulo capaz de interpretar sentenças nessa linguagem.

Problema do padrão Interpreter

Em sistemas que trabalham com cálculos customizáveis, uma solução comum consiste em definir esses cálculos utilizando expressões matemáticas. Uma expressão matemática deve seguir uma gramática que estabelece as regras de formação das expressões.



Fórmulas matemáticas complexas em um computador, representando análise de dados.

O problema resolvido pelo padrão Interpreter consiste em definir uma forma de representar e interpretar uma linguagem definida por uma gramática.

Solução do padrão Interpreter

A estrutura da solução proposta pelo padrão Interpreter está representada no diagrama de classes a seguir.

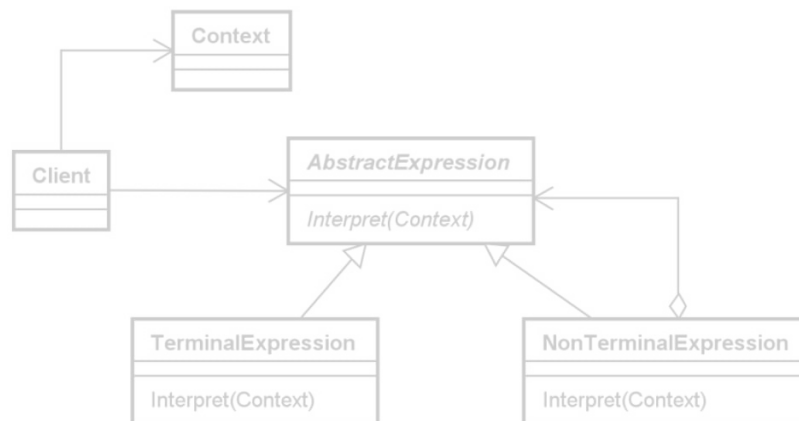
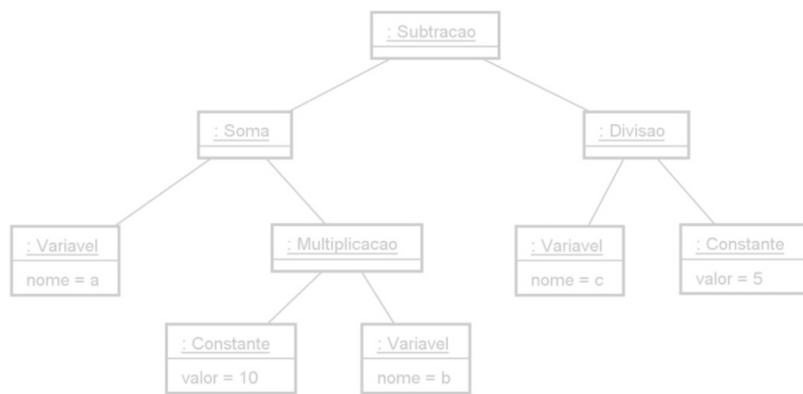


Diagrama de classes.

Os elementos da gramática da linguagem são definidos por objetos que formam uma árvore sintática abstrata. Os símbolos terminais correspondem ao participante TerminalExpression, enquanto os elementos compostos por outros correspondem ao participante NonTerminalExpression.

Veja, na imagem a seguir, como a expressão $a + (10 * b) - (c / 5)$ pode ser representada por uma estrutura hierárquica de objetos.



Estrutura hierárquica de objetos.

Atenção

Nessa gramática, constantes e variáveis são elementos terminais, enquanto os operadores de soma, subtração, multiplicação e divisão são elementos não terminais, estando ligados a um elemento à esquerda e outro à direita.

Consequências e padrões relacionados ao Interpreter

Esse padrão facilita a modificação e a ampliação de uma gramática. A ideia é implementar a estrutura com uma classe simples para cada elemento da gramática, em vez de tentar centralizar a solução em uma única classe. O padrão é:



Indicado

Para linguagens com uma gramática simples, como é o caso, por exemplo, das expressões aritméticas.



Não indicado

Para linguagens mais complexas, recomenda-se utilizar outras soluções, como ferramentas que gerem automaticamente interpretadores.

A operação interpret, definida nos participantes do padrão, pode ser vista como um processamento específico que se deseja realizar com esses elementos. Por exemplo, no caso das expressões aritméticas, o processamento poderia ser calcular o resultado. Entretanto, podem existir outros processamentos, como:



Verificação sintática



Verificação semântica



Obtenção do texto da expressão

Nesses casos, recomenda-se aplicar o padrão Visitor em conjunto com o Interpreter, em que o Interpreter define a estrutura dos elementos da linguagem, enquanto cada processamento é implementado em uma classe Visitor específica.

Atividade 1

Assinale a alternativa que expressa a intenção do padrão de projeto Interpreter:

A

Definir uma estratégia em que um objeto notifica outros objetos interessados em saber que ocorreu uma modificação no valor de um ou mais dos seus atributos, de modo que cada objeto notificado possa interpretar a notificação de forma independente.

B

Permitir a separação de diferentes algoritmos de interpretação de dados em classes distintas, desacoplando os algoritmos das estruturas de dados sobre as quais eles trabalham.

geração de cada relatório. Segundo, para cada novo relatório, é necessário abrir a classe `ServicoRelatorio` e acrescentar o código específico desse relatório, provavelmente copiando, colando e adaptando o código de um dos relatórios já implementados.

Solução do padrão Template Method

A estrutura da solução proposta pelo padrão `TemplateMethod` está representada no diagrama de classes a seguir:

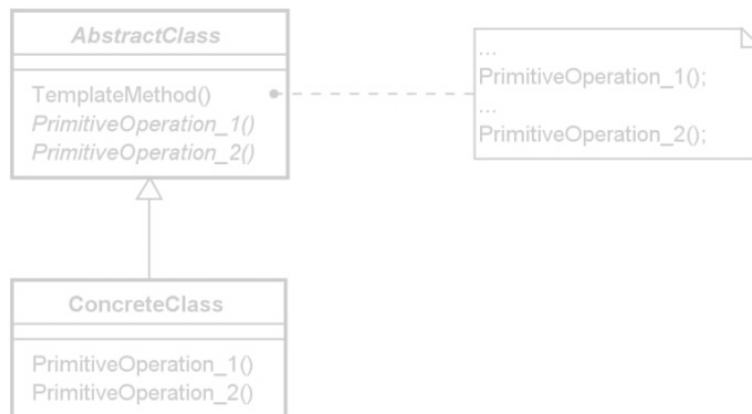


Diagrama de classes

O padrão sugere definir um método em uma classe abstrata que implemente a estrutura do algoritmo comum a todas as suas implementações específicas. A estrutura desse algoritmo é composta por um conjunto de passos que podem ser redefinidos pelas subclasses. Cada ponto de variação no algoritmo corresponde à definição de uma operação na superclasse que pode ser especializada nos seus descendentes. Esses pontos de variação podem ser:

Operações abstratas

Tais operações obrigam as subclasses a implementá-las.



Métodos hook (gancho)

Operações realizadas na superclasse e que podem ser substituídas por implementações específicas em uma ou mais subclasses.

O código a seguir ilustra a estrutura de implementação para os relatórios com a aplicação desse padrão. A classe abstrata `Relatorio` define três operações abstratas que serão implementadas por todas as subclasses. A operação `gerar` é o método padrão (template method) que define a estrutura comum do algoritmo, seguida por todos os relatórios.

O que você aprendeu neste conteúdo?

- Os padrões de projeto comportamentais Chain of Responsibility, Command e Iterator.
- Os padrões de projeto comportamentais Mediator, Memento e Strategy.
- Os padrões de projeto comportamentais Observer, Visitor e State.
- Os padrões de projeto comportamentais Interpreter e Template Method.



Podcast

Ouçá o podcast. Nele, abordaremos os padrões comportamentais mais utilizados.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



Explore +

Para saber mais sobre a programação orientada a objetos, acesse o site da DevMedia e leia o artigo **Utilização dos princípios SOLID na aplicação de padrões de projeto**.

O site **Padrões de Projeto / Design patterns – Refactoring.Guru** apresenta um conteúdo interativo e bastante completo de todos os

padrões GoF, com exemplos de código em diversas linguagens de programação.

Além dos padrões GoF tradicionais, outros padrões voltados para o desenvolvimento de aplicações corporativas em Java EE podem ser encontrados no livro **Java EE 8 Design Patterns and Best Practices**, de Rhuan Rocha e João Purificação, de 2018. A obra aborda padrões para interface com o usuário, lógica do negócio, integração de sistemas, orientação a aspectos, programação reativa e microserviços.

Referências

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1. ed. Boston: Addison-Wesley, 1994.

METSKER, S. J.; WAKE, W. C. **Design Patterns in Java**. 1. ed. Boston: Addison-Wesley, 2006.

Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

Download material

O que você achou do conteúdo?



 Relatar problema