Padrões GoF Comportamentais

Profª. Alexandre Luis Correa

Descrição

Padrões GoF de Projeto Comportamentais: Chain of Responsibility; Command; Interpreter; Iterator; Mediator; Memento; Observer; State; Strategy; Template Method; Visitor.

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.

Módulo 4

Padrões de projeto comportamentais Interpreter e Template Method

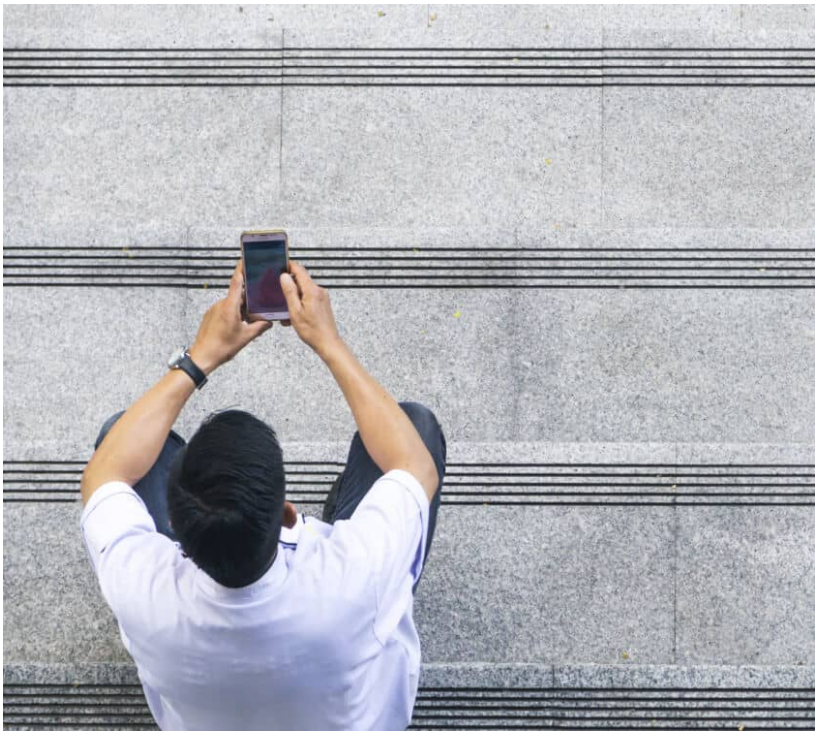
Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Interpreter e Template Method.

Introdução

Os padrões GoF, do inglês Gang of Four, são padrões de projeto orientado a objetos, divididos em três categorias: de criação, estruturais e comportamentais. São assim denominados por terem sido introduzidos pelos quatro autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software* (GAMMA *et al*., 1994).

Os padrões de projeto GoF comportamentais descrevem estratégias para organizar algoritmos e atribuir responsabilidades aos objetos. Esses padrões visam organizar fluxos de controle mais complexos, quebrando-os em responsabilidades menores e mais simples, que são distribuídas pelos objetos participantes da solução. Neste conteúdo, você aprenderá os onze padrões desse grupo: Chain of Responsibility, Command, Iterator, Mediator, Memento, Strategy, Observer, Visitor, State, Interpreter e Template Method.

É comum encontramos implementações em Java com classes contendo diversos métodos com dezenas ou centenas de linhas, concentrando grande parte da complexidade de um sistema. Essa concentração de responsabilidades em um pequeno número de classes com grande quantidade de código gera projetos inflexíveis e de difícil manutenção. Os padrões comportamentais propõem soluções baseadas na distribuição da complexidade por um conjunto maior de objetos, com o objetivo de gerar sistemas mais flexíveis, menos acoplados e mais fáceis de manter.



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

Intenção do padrão Chain of Responsibility

Chain of Responsibility é um padrão que permite o envio de uma requisição para o primeiro elemento de uma cadeia de objetos que podem realizar algum processamento relacionado a essa requisição, fazendo com que o objeto requisitante não precise ficar acoplado a todos os objetos da cadeia, mas apenas ao objeto para o qual ele envia a requisição.

Problema resolvido pelo padrão Chain of Responsibility

Imagine que você esteja implementando um módulo tarifador de ligações telefônicas, responsável por calcular o custo de cada ligação telefônica efetuada em uma central telefônica do tipo PABX de uma empresa.

Suponha que exista uma regra de precificação específica para cada tipo de ligação: ligação interna entre ramais, ligação local, ligação DDD e ligação DDI. Uma solução possível para esse problema seria concentrar toda a lógica de tarifação em uma única classe, conforme podemos ver no código a seguir.

Javascript

Essa solução concentra toda a lógica de tarifação em um único módulo, tornando-o extenso e complexo. Imagine como ficaria esse módulo, caso outros tipos de ligação fossem adicionados, como, por exemplo, ligações para celular ou ligações considerando as operadoras de origem e destino.

Solução 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.

> **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.

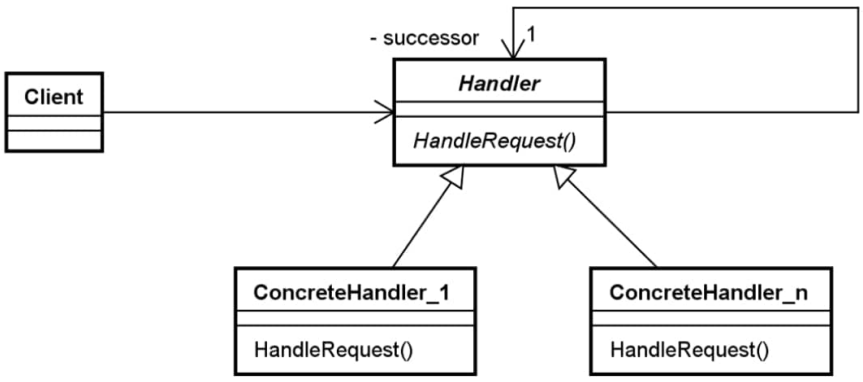


Diagrama de classes.

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.

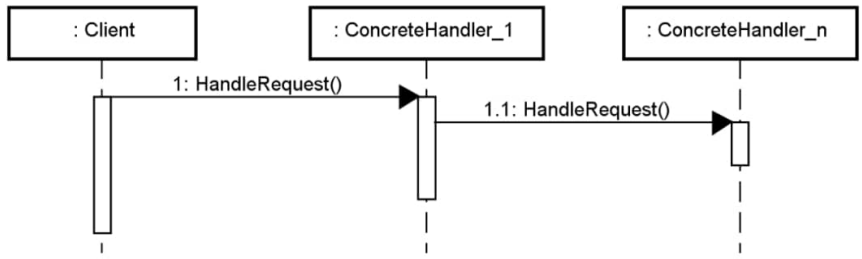


Diagrama de sequência

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 ConcreteHandler definido no padrão. A implementação do **método tarifar** em cada tarifador específico retorna o custo da ligação, se ela for do tipo apropriado ou, caso contrário, chama o método tarifar definido na superclasse que, por sua vez, repassa a responsabilidade do cálculo para o próximo tarifador da cadeia.

Java

Finalmente, observe como na nova implementação a classe **ServicoTarifacaoTelefonica** pode operar com qualquer tarifador que seja plugado no seu construtor. A operação tarifar dessa classe simplesmente faz a requisição para o primeiro tarifador da cadeia.

A classe **ConfiguradorServicoTarifacaoTelefonica** é um exemplo simples de como uma cadeia de responsabilidades pode ser criada e injetada em uma classe cliente. Ela cria uma cadeia de tarifadores, um para cada tipo de ligação, usando a operação setSucessor para ligar um tarifador ao outro. Uma vez criada, a cadeia é passada como argumento para o construtor da classe **ServicoTarifacaoTefefonica**.

Java

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.

Padrão Command

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
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.

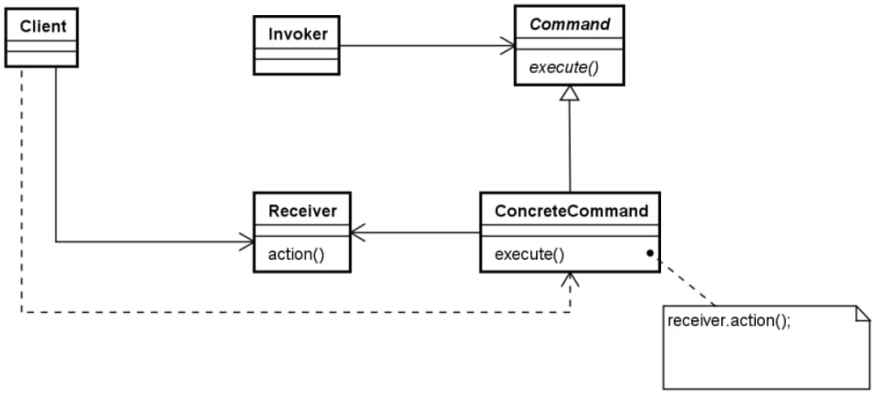


Diagrama de classes

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

> **Note que o construtor de cada comando concreto recebe o objeto da classe Jogo para o qual a requisição será passada. Portanto, Jogo corresponde ao participante Receiver do padrão.**

Em seguida, vamos criar a configuração do jogo, associando os eventos gerados pela interface com o usuário, isto é, teclas e botões do mouse, com os comandos. Note que podemos associar o mesmo comando a diferentes eventos de interface, como ocorre com o comando pular. A classe **ConfiguracaoJogo** corresponde ao participante **Client** do padrão.

Java

Por fim, vamos ver a nova implementação da classe **InterfaceJogo** correspondente ao participante Invoker do padrão.

Essa classe recebe uma configuração dos códigos dos eventos de interface e seus respectivos comandos. Note como, em comparação com a versão original, o método **tratarEvento** ficou desacoplado de todos os possíveis comandos. Isso permite adicionarmos comandos ao jogo sem que esta classe precise ser modificada.

Java

Consequências e padrões relacionados ao padrão Command

O padrão Command promove o desacoplamento entre o objeto que faz a requisição e o objeto que realiza a operação requisitada. Cada comando passa a ser uma classe e, com isso, é possível reutilizar um código comum entre os comandos e adicionar comandos sem precisar alterar classes já existentes.

Esse desacoplamento também resulta em maior facilidade para a implementação de testes unitários automatizados, uma vez que podemos executar uma sequência de comandos de forma isolada do seu mecanismo de disparo, como os eventos da interface gráfica com o usuário, por exemplo. A interface Comando pode ser estendida adicionando-se as seguintes operações:

undo

Responsável por desfazer as ações realizadas pelo comando.

redo

Responsável por realizar novamente as ações desfeitas pela operação undo.

Como o comando é encapsulado em um objeto, ele pode armazenar as informações necessárias para reverter ou repetir suas ações, tornando possível a implementação dessas operações.

Reflexão

Como um macro comando, isto é, uma sequência de comandos, pode ser implementado, aplicando-se simultaneamente os padrões Command e Composite?

Padrão Iterator

Intenção do padrão Iterator

O objetivo do padrão Iterator é permitir o acesso sequencial aos objetos de uma coleção, sem expor a sua representação interna.

Problema resolvido pelo padrão Iterator

Suponha que você tenha que percorrer uma coleção de produtos armazenados em um ArrayList. Você poderia escrever um código como o apresentado a seguir:

Java

O problema dessa solução é que a classe Exemplo conhece a representação interna da coleção de produtos, isto é, ela sabe que os produtos estão organizados em um ArrayList.

Atenção

No exemplo apresentado, caso tivéssemos outra coleção de produtos organizada em uma estrutura de mapa indexado pelo código do produto, não poderíamos utilizar a classe Exemplo, pois ela só funciona com produtos organizados em um ArrayList.

Solução do padrão Iterator

A solução proposta pelo padrão Iterator consiste em colocar a responsabilidade pelo percurso e pelo acesso aos elementos de uma coleção em um objeto específico, denominado Iterator. A estrutura do padrão está ilustrada no diagrama de classes a seguir.

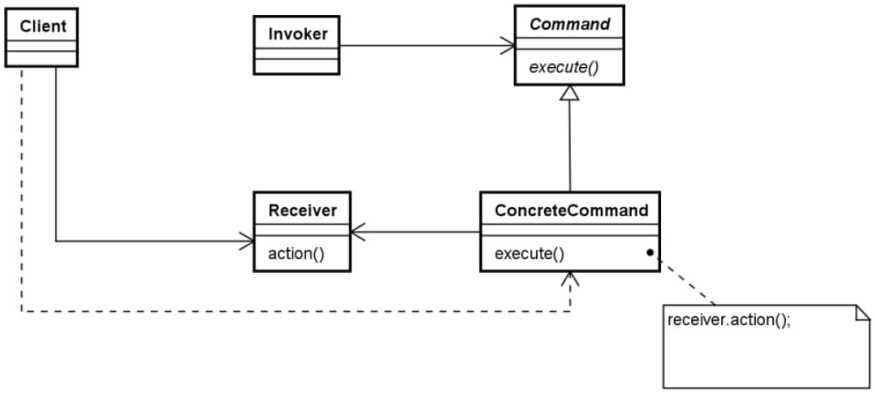


Diagrama de classes

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 (ArrayIterator), LinkedList (ListIterator), HashSet (KeyIterator) e TreeSet (ValueIterator).

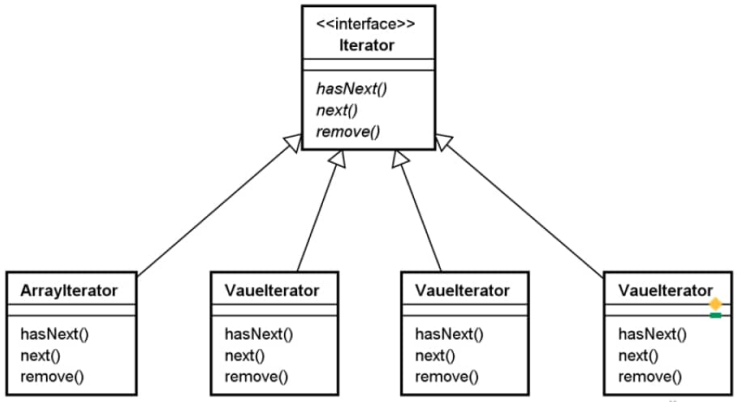


Diagrama de classes

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

Atenção

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 ativas para uma mesma coleção, uma vez que cada Iterator guarda o seu próprio estado, isto é, a posição corrente na coleção.

Além disso, é possível implementar iteradores com diferentes formas de percurso. Por exemplo, podemos percorrer uma árvore binária em pré-ordem, ordem simétrica e pós-ordem. Cada forma de percurso pode ser definida em uma implementação específica da interface Iterator.

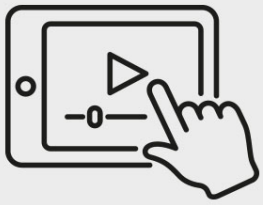
Atenção

O padrão Iterator é frequentemente utilizado com o padrão Factory Method, uma vez que cada método Iterator do participante ConcreteAggregate é um Factory Method responsável por instanciar o respectivo ConcreteIterator.

Padrão de projeto Command

No vídeo, apresentamos um exemplo de aplicação do padrão Command no desenvolvimento de software e seus impactos na automação de testes unitários.

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 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.

B Permitir a utilização de objetos remotos, como se tanto o objeto remoto chamado quanto o objeto chamador estivessem rodando no mesmo processo.

C Fornecer uma interface de alto nível para um subsistema, tirando do módulo cliente a responsabilidade de interagir com os diversos elementos que compõem esse subsistema, o que reduz o nível de acoplamento do cliente.

D Fornecer um mecanismo para salvar e restaurar o estado de um objeto sem quebrar o seu encapsulamento.

E Encapsular requisições em objetos, possibilitando registrar o histórico das requisições e gerenciar uma fila de requisições, por exemplo.

Parabéns! A alternativa A está correta.

O padrão Chain of Responsibility cria uma cadeia de objetos que podem realizar algum processamento relacionado a uma requisição, fazendo com que o objeto requisitante esteja acoplado apenas ao primeiro elemento da cadeia. As alternativas B, C, D e E correspondem, respectivamente, às intenções dos padrões Proxy, Facade, Memento e Command.

Questão 2

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

A Permitir que um objeto possa disparar uma operação em outros objetos conectados em uma lista encadeada sequencial, sem que o objeto disparador precise interagir diretamente com todos eles.

B Facilitar a implementação de interações do usuário com o sistema.

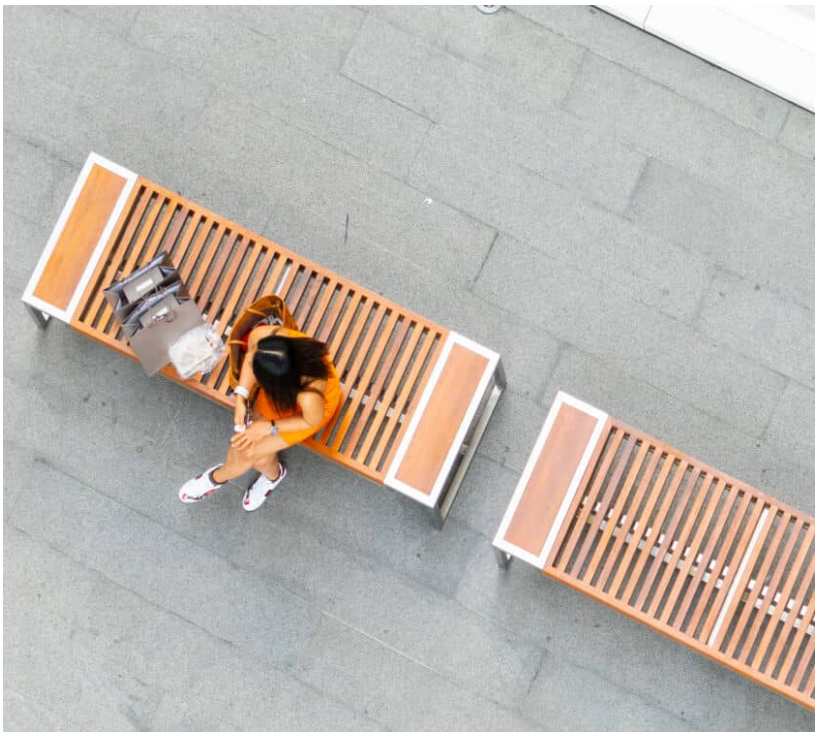
C Permitir que um módulo possa fazer o percurso por uma coleção de objetos sem precisar conhecer a representação interna dessa coleção.

D Facilitar o desenvolvimento iterativo e incremental de um software.

E Adicionar funcionalidades a uma classe sem utilizar subclasses, mas sim por meio de uma estrutura de composição dinâmica e flexível.

Parabéns! A alternativa C está correta.

A alternativa A descreve a intenção do padrão Chain of Responsibility. A alternativa B está incorreta, pois o padrão Iterator não está relacionado com a implementação de interface com o usuário. A alternativa D está incorreta, pois o padrão Iterator não está relacionado com o ciclo de vida do desenvolvimento de software. A alternativa E descreve a intenção do padrão Decorator.



2 - Padrões de projeto comportamentais Mediator, Memento e Strategy

Ao final deste módulo, você será capaz de reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Mediator, Memento e Strategy.

Padrão Mediator

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.

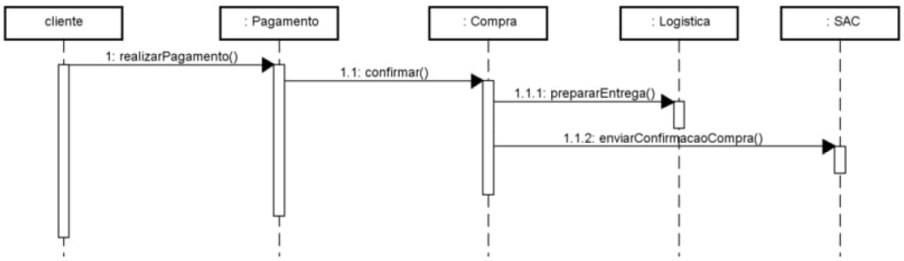


Diagrama de sequência

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:

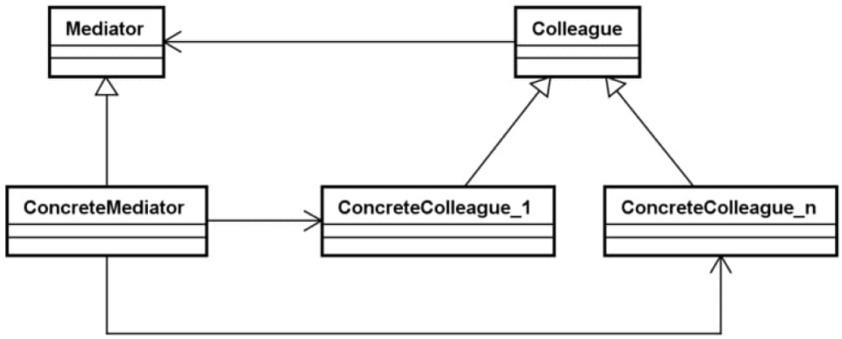


Diagrama de classes

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, Logistica 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

Consequências e padrões relacionados ao padrão Mediator

O padrão Mediator gera uma solução de menor acoplamento para interações complexas entre objetos, pois ele substitui uma estrutura de interação do tipo muitos para muitos para uma estrutura um para muitos, que são mais fáceis de entender e manter. Por outro lado, o controle fica centralizado no mediador. Um cuidado especial deve ser tomado na implementação do mediador para evitar que ele se torne um componente monolítico com grande complexidade de manutenção.

> **O mediador deve ser apenas um concentrador de eventos e um coordenador de execução, ficando a lógica do processamento distribuída pelos elementos a ele**

> conectados.

Esse padrão é muito utilizado na implementação de componentes de interface com o usuário, em que um mediador centraliza a recepção e o tratamento dos eventos gerados pelos componentes da interface, como botões, campos de texto e listas de seleção, por exemplo. Em uma tela em que o usuário entra um endereço, a modificação do campo CEP pode acarretar a atualização dos campos logradouro, cidade e estado. Um mediador que receba a notificação de que o CEP foi alterado e promova a atualização dos demais campos envolvidos é uma solução normalmente empregada nesse contexto.

Atenção

Como a comunicação entre os participantes e o mediador se dá por meio de notificações de eventos, é comum o Mediator ser aplicado em conjunto com o padrão Observer. Nesse caso, o mediador corresponde ao participante Observer, enquanto os objetos notificadores correspondem ao participante Subject do padrão Observer.

Padrão Memento

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.

Problema resolvido pelo padrão Memento

Suponha que você esteja desenvolvendo uma aplicação em que seja necessário desfazer o efeito produzido por uma ou mais operações em determinado objeto ou em um conjunto de objetos, procedimento conhecido como **undo**. Para isso, é necessário guardar o estado do objeto anterior à execução das operações que precisarão ser desfeitas.

Veja a classe Pedido cujo código é listado a seguir. Ela define o número do pedido, o número do item a ser utilizado na criação de um novo item e a lista de itens do pedido. Apenas a classe Pedido tem acesso ao array de itens. Portanto, a adição de novos itens ao pedido deve ser realizada por meio da operação adicionarItem.

Java

Agora pense como você implementaria uma funcionalidade que permitisse o usuário desfazer as últimas operações de adição de itens ao pedido. E se oferecêssemos um recurso que possibilitasse o usuário interromper sua compra e voltar mais tarde, sem perder o conteúdo já adicionado ao pedido?

> **O padrão Memento serve para nos ajudar em situações como essas.**

Solução do padrão Memento

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

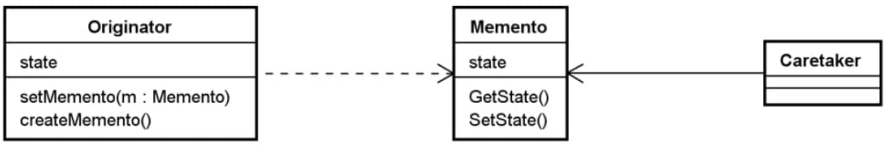


Diagrama de classes

No código a seguir, implementamos um memento para objetos da classe Pedido. A classe **PedidoMemento** representa o participante Memento do padrão, armazenando todas as informações necessárias para a

restauração de um objeto Pedido.

Java

A classe Pedido representa o participante **Originator** do padrão, pois é o seu estado que deve ser armazenado e restaurado por meio do uso de mementos. Veja o próximo código:

Java

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

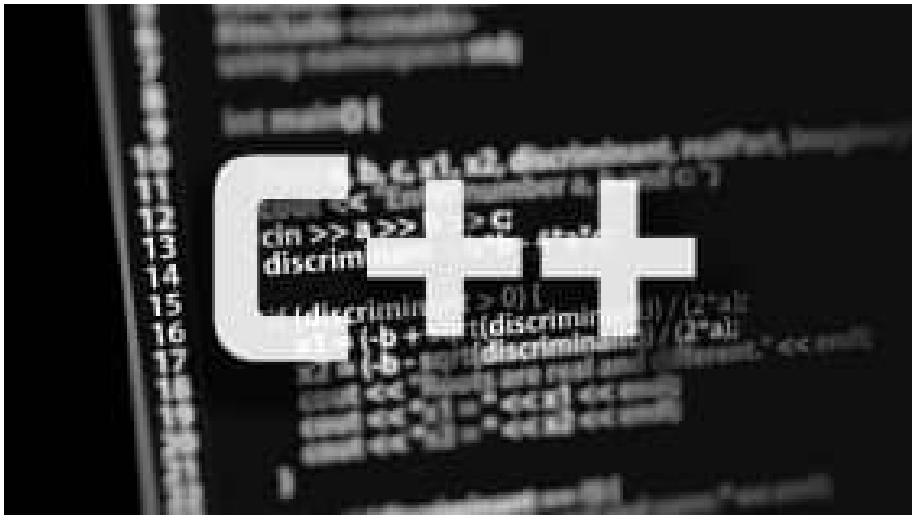
O padrão Memento facilita a implementação de problemas nos quais precisamos desfazer certas modificações de estado em objetos decorrentes da execução de operações ou implementar algum

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

Dificulta a definição da interface do memento, de forma que somente o originador tenha acesso.



C++

Permite uma definição mais rigorosa por meio da utilização da palavra reservada *friend*.

Atenção

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.

Padrão Strategy

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

Boostrap

Interpolação polinominal 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?

Saiba mais

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.

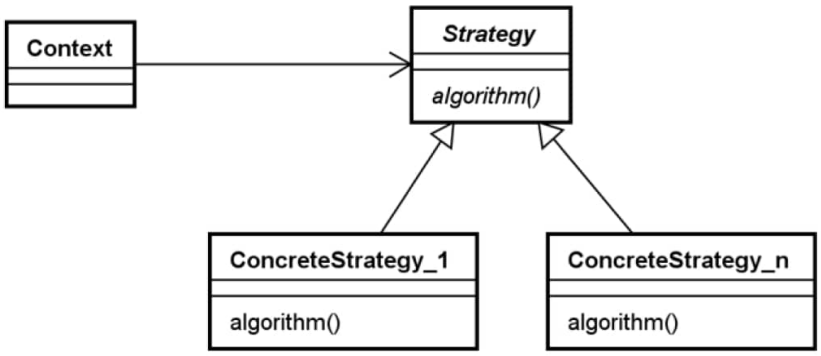


Diagrama de classes

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.

Java

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 dependências, para generalizar o processo de instanciação do algoritmo específico a ser utilizado.

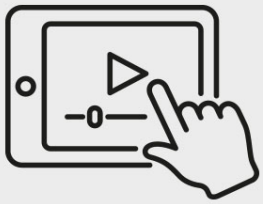
Atenção

O padrão Strategy, por gerar objetos sem estado, pois os dados de que o algoritmo precisa estão definidos na classe contexto, pode ser implementado em combinação com o padrão Flyweight, de modo que uma única instância de cada algoritmo seja compartilhada pelos vários contextos. Em nosso exemplo, se fossem criadas 50 instâncias de CurvaJuros, bastaria instanciar um único objeto de cada algoritmo, caso adicionássemos o padrão Flyweight à solução.

Padrão de projeto Strategy

No vídeo, apresentamos exemplos de aplicação do padrão Strategy no desenvolvimento de software.

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

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

A Fornecer aos clientes um objeto intermediário com uma réplica da interface do objeto fornecedor que contém os métodos que realmente serão executados. Esse objeto intermediário delega as requisições dos clientes para o objeto fornecedor.

B Encapsular algoritmos em objetos, permitindo que eles possam ser utilizados de forma intercambiável pelos módulos clientes.

C Definir uma estratégia em que um objeto notifica outros objetos interessados em saber que ocorreu uma modificação no seu estado.

D Permitir a composição de objetos, a partir de diversos pequenos objetos imutáveis, por meio de uma estratégia baseada em compartilhamento.

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.

Questão 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.

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

Intenção do padrão Observer

O padrão Observer define uma relação de dependência entre objetos, de modo a garantir que, quando alguma modificação no estado de determinado objeto ocorrer, todos os objetos dependentes sejam notificados e atualizados automaticamente.

Problema resolvido pelo padrão Observer

Imagine que você esteja desenvolvendo um *software* para acompanhamento de vendas de uma empresa e que os dados de venda por produto e região sejam apresentados em três painéis simultaneamente.

O primeiro painel é apresentado em tabela:

Produto	Região 1	Região 2	Região 3	Total
A	40	30	100	170
B	120	80	70	270
C	80	50	90	220
Total	240	160	260	660

Tabela: Painel 1
Elaborada por: Alexandre Luis Correa

Os painéis 2 e 3 são apresentados em gráficos:

Gráfico: Painel 2
Elaborado por: Alexandre Luis Correa

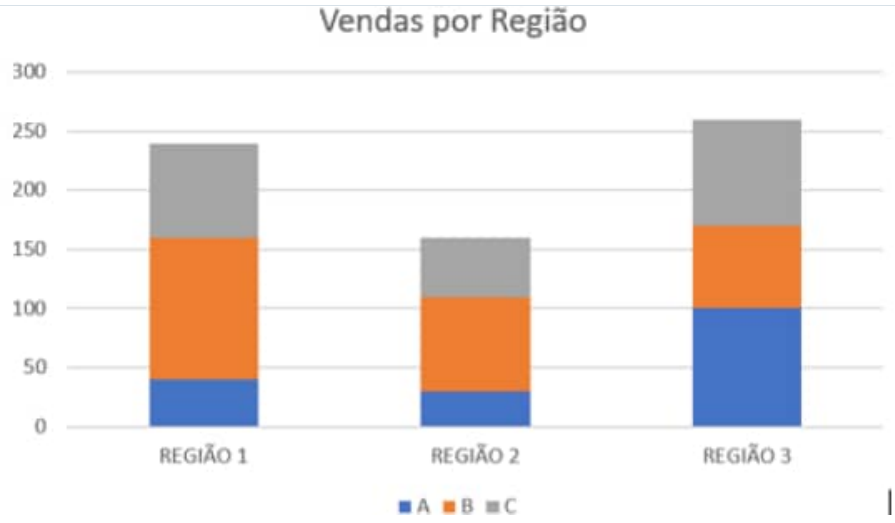


Gráfico: Painel 3
Elaborado por: Alexandre Luis Correa

Os três painéis apresentam dados oriundos de um sumário de vendas. Quando qualquer dado desse sumário mudar, os três painéis devem ser atualizados.

> **O problema resolvido pelo padrão Observer consiste em manter a consistência entre objetos relacionados de modo que, se o estado de um objeto mudar, todos os objetos afetados por essa mudança sejam notificados e atualizados.**

Solução do padrão Observer

O padrão Observer possui a estrutura definida no diagrama de classes a seguir:

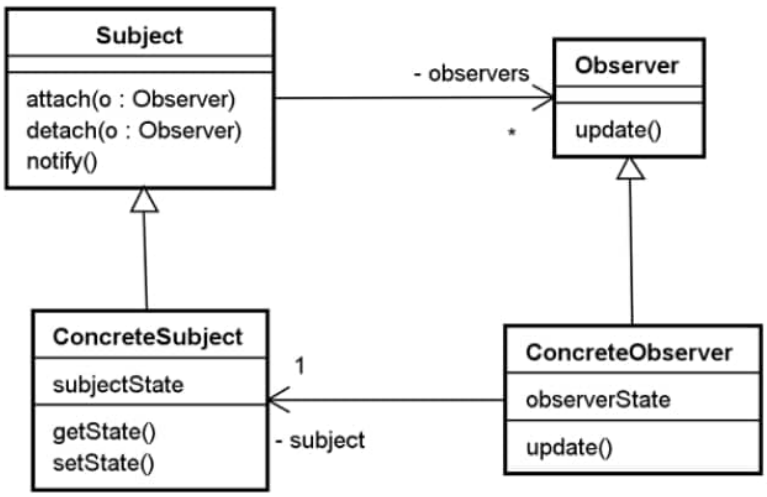


Diagrama de classes

Vamos analisar cada participante do diagrama:

Subject

O participante Subject define uma interface para registro (attach) e desligamento (detach) de observadores que devem ser notificados quando houver uma mudança no estado de um objeto concreto. Os observadores são armazenados em uma coleção mantida pelo Subject.

Observer

O participante Observer define uma interface para o recebimento das notificações enviadas pelo Subject.

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.

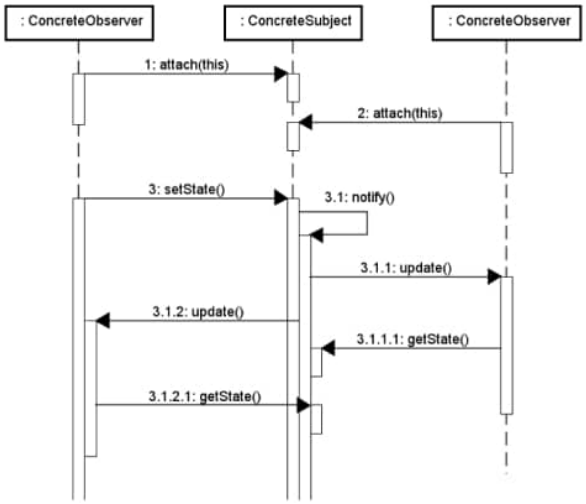


Diagrama de sequência

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

Em seguida, definimos duas classes que desempenham o papel de observadores de modificações nas instâncias da classe Ponto. As duas classes implementam a operação update, definida na classe Observable, que é chamada pela operação notifyObservers. Assim, cada observador pode recuperar os dados atualizados do ponto e fazer qualquer tipo de atualização, como exibi-los na tela, por exemplo.

Java

Vamos ver agora como o objeto subject e os observadores devem ser conectados para que o mecanismo funcione. O código a seguir apresenta um exemplo em que uma instância da classe Ponto é criada e dois observadores são adicionados a ela por meio da operação addObserver. A partir daí, toda vez que uma operação setX ou setY for executada, a operação update de cada observador será automaticamente chamada pelo mecanismo de notificação.

Java

Consequências e padrões relacionados ao Observer

O padrão Observer permite que os objetos detentores de informação relevante possam notificar que ocorreram modificações nessa informação aos possíveis interessados. Isso é feito por meio de uma interface genérica que facilita a adição de novos interessados. Esse padrão é especialmente utilizado na implementação de elementos de interface gráfica com o usuário que, além de gerarem eventos que precisam ser notificados para outros elementos, precisam ficar sincronizados com a sua fonte de dados.

Uma consequência negativa desse padrão é que a solução proposta pode dar origem a muitas chamadas da operação update. Imagine uma atualização de diversas informações de um Subject com dezenas de observadores registrados. Nessa situação, seriam geradas centenas de chamadas da operação update. Portanto, certos problemas podem demandar a implementação de um protocolo específico de atualização, indicando quais modificações devem ser notificadas para cada observador e em que momento.

Atenção

O padrão Observer pode ser utilizado em conjunto com o padrão Mediator, especialmente quando certas modificações no Subject gerarem uma lógica complexa de atualizações em outros objetos. Em vez de estabelecermos diversas conexões diretas entre os observadores e os subjects, podemos inserir um mediador que centralize as notificações enviadas pelos subjects e coordenar a lógica de atualização dos diversos observadores.

Padrão Visitor

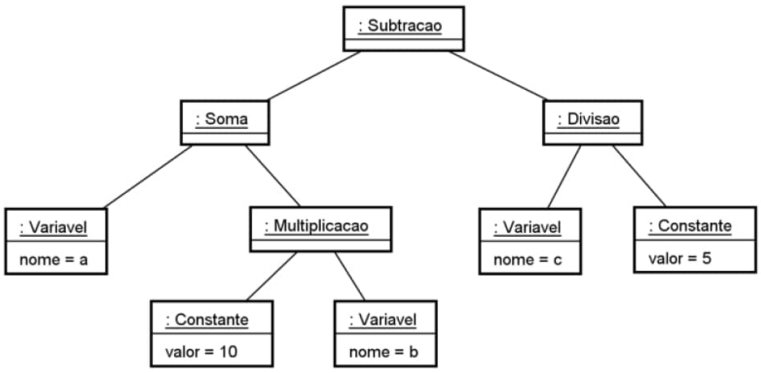
Intenção do padrão Visitor

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

Imagine que você esteja desenvolvendo um sistema para a área financeira que permite a parametrização de alguns cálculos por meio de expressões.

Por exemplo, a expressão a + (10 * b) – (c / 5) pode ser representada pela seguinte estrutura de objetos:



Objetos.

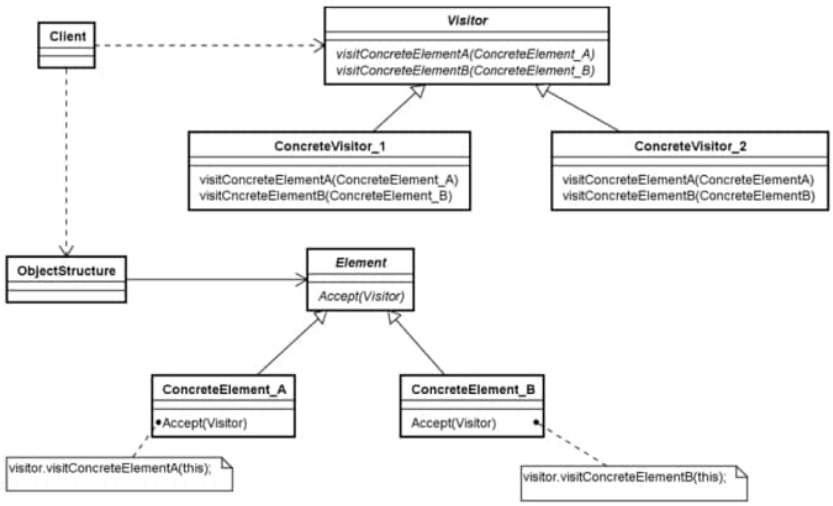
Essa estrutura de objetos pode ser percorrida para diferentes finalidades, tais como: verificar se a expressão é válida, calcular o valor da expressão, imprimir a expressão em formato texto, ou imprimir a expressão em notação polonesa.

Para cumprir todas essas finalidades, podemos acrescentar aos objetos dessa estrutura operações que possibilitem a validação de uma expressão, o cálculo de valor e assim por diante. Entretanto, sempre que quisermos acrescentar uma nova finalidade, teremos que modificar todas as classes dessa estrutura.

> **O padrão Visitor nos permite acrescentar novas funcionalidades sem precisar modificar as classes que representam a estrutura fundamental dos objetos que compõem a expressão.**

Solução do padrão Visitor

A estrutura da solução proposta pelo padrão Visitor é apresentada no diagrama UML a seguir:



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

Recomendação

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 dessas classes. Portanto, o padrão Visitor não é adequado para estruturas que mudem com frequência.

Atenção

O padrão Visitor pode ser utilizado em conjunto com os padrões Composite e Interpreter.

Padrão State

Intenção do padrão State

O padrão State permite que um objeto modifique o seu comportamento quando o seu estado mudar, como se o objeto tivesse mudado de classe. Em vez de uma única classe tratar os estados dos seus objetos 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.

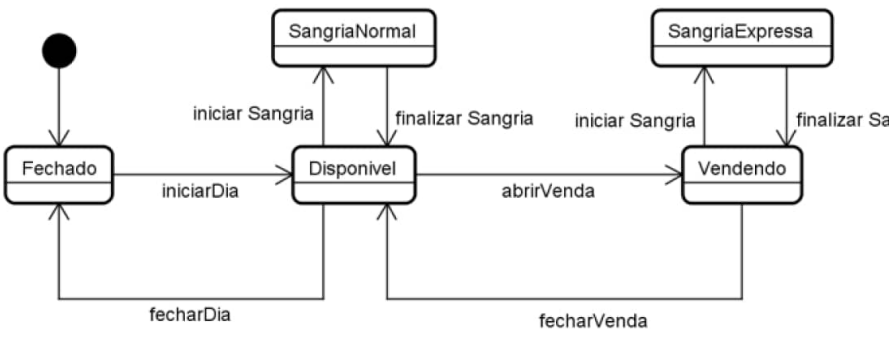


Diagrama de estado.

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.

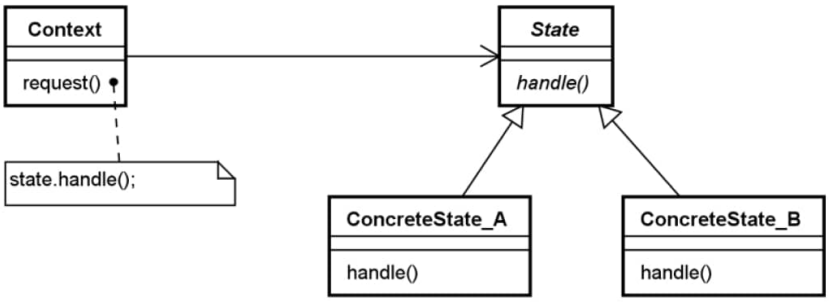


Diagrama de estado

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.

Padrão de projeto Observer

No vídeo, abordaremos a implementação do padrão Observer em diferentes versões do Java.

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

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

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.

Questão 2

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

A Fornecer uma interface para a visita de um objeto remoto, de modo que os módulos chamadores possam se comunicar com o objeto remoto como se ele estivesse rodando localmente.

B Reduzir o acoplamento de um módulo cliente com os elementos que compõem um subsistema, fornecendo para o módulo cliente uma interface de alto nível aos serviços desse subsistema.

C

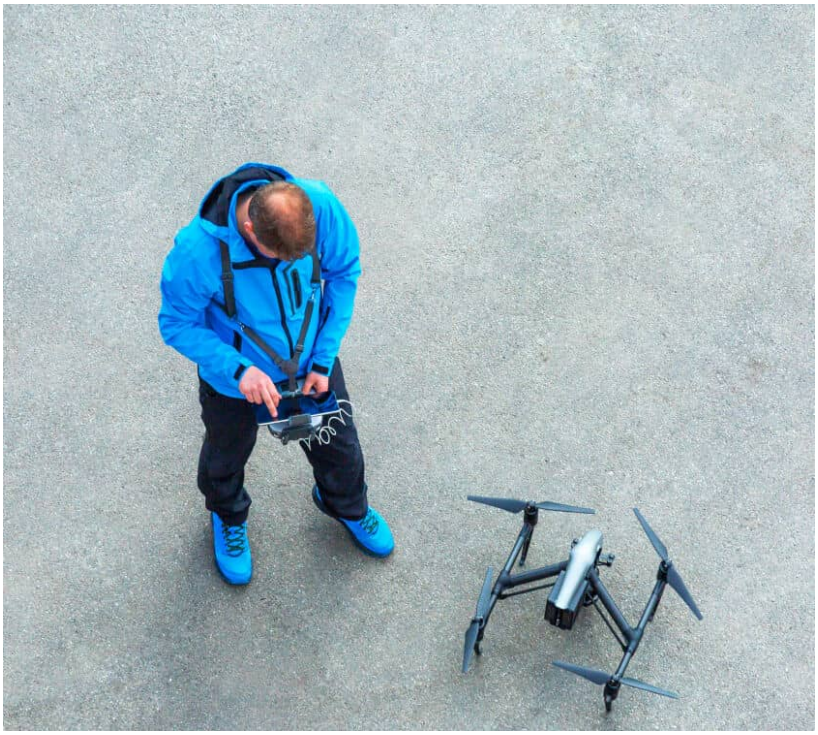
Permitir a adição de novas funcionalidades, por meio da utilização de classes ortogonais a uma estrutura tipicamente hierárquica de objetos.

D Representar hierarquias de composição de objetos, de modo que tanto os objetos individuais como os agregados possam ser visitados pelo mesmo conjunto de operações.

E 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. Os objetos notificados devem visitar o objeto notificador para a obtenção os novos valores.

Parabéns! A alternativa C está correta.

A alternativa A se aproxima do propósito do padrão Proxy, mas a palavra visitar não se aplica a esse padrão. As alternativas B e D correspondem, respectivamente, aos padrões Facade e Composite. A alternativa E se aproxima da definição do propósito do padrão Observer.



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.

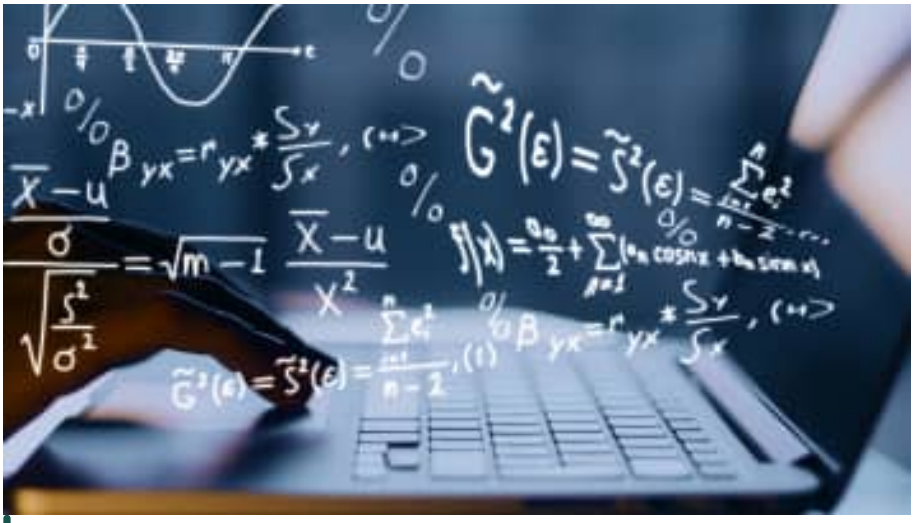
Padrão Interpreter

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.



> **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.

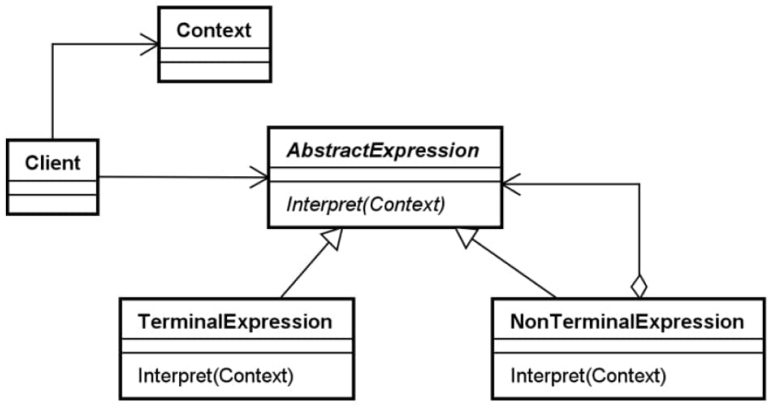
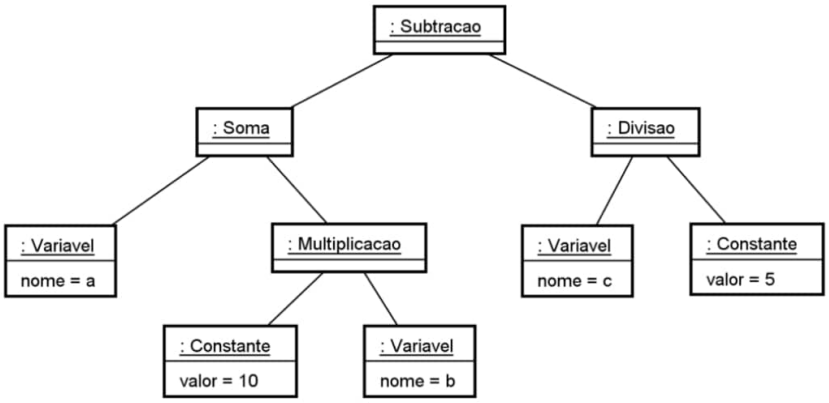


Diagrama de classes

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.

Padrão Template Method

Intenção do padrão Template Method

O propósito do padrão Template Method é definir o esqueleto de um algoritmo em uma superclasse, em que os passos comuns podem ser implementados na própria superclasse e os passos específicos são implementados nas subclasses.

Problema do padrão Template Method

Suponha que você esteja implementando um sistema que deve gerar alguns relatórios textuais. A preparação dos dados para qualquer relatório segue três passos fundamentais: gerar o cabeçalho, gerar o corpo e gerar o sumário.

Imagine, por exemplo, que você tenha que implementar um relatório de vendas e um relatório de devoluções em um período. O código a seguir ilustra uma solução inicial para esses relatórios.

Java

Essa solução não é adequada, primeiro por ser baseada em repetição de código com estrutura muito similar, apesar de diferirem nos detalhes de 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:

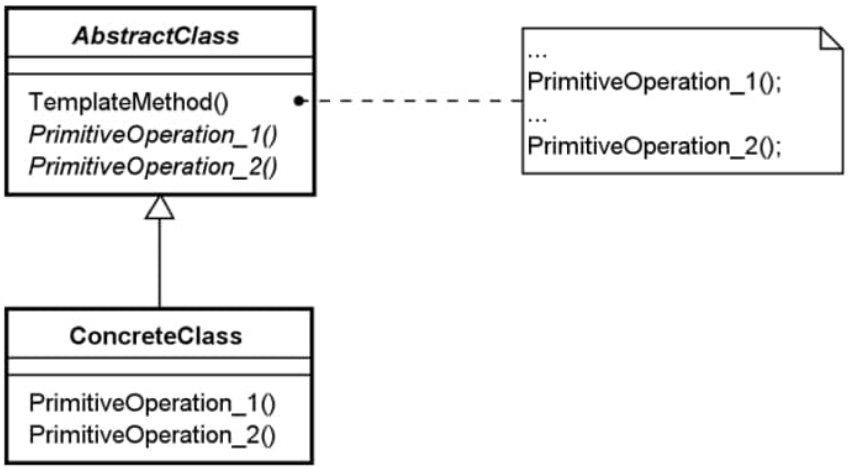


Diagrama de classes

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 obrigarão 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. Desse modo, cada relatório define as particularidades da geração de cabeçalho, corpo e sumário na implementação dessas operações genéricas.

Java

Atenção

Outro benefício dessa implementação é que o código em comum entre os diferentes relatórios pode ser efetuado uma única vez na superclasse Relatório, reduzindo a quantidade de código repetido.

Consequências e padrões relacionados ao Template Method

O padrão Template Method é muito utilizado na implementação de bibliotecas genéricas de classes e de *frameworks*. O resultado obtido pela sua aplicação é conhecido como o princípio de Hollywood, isto é, “Não nos chame, nós o chamaremos”, uma referência a como a superclasse chama as operações específicas das subclasses e não o contrário.

O padrão Factory Method costuma ser utilizado no contexto de um Template Method, nos casos em que um dos passos do algoritmo seja instanciar algum objeto específico. Os padrões Template Method e Strategy são formas de representação de algoritmos:

Template Method

Permite a variação de partes de um algoritmo, por meio de uma estrutura de herança.

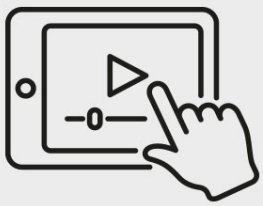
Strategy

É aplicado quando desejamos criar algoritmos diferentes, mas que não possuam uma estrutura comum.

Padrão de projeto Template Method

No vídeo, apresentaremos exemplos de aplicação do padrão Template Method no desenvolvimento de *software*.

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 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.

C Permitir a utilização mais eficiente de memória, evitando a criação de um número muito grande de objetos imutáveis simples, por meio de uma solução baseada em compartilhamento.

D Definir uma representação para a gramática de uma linguagem é uma forma de tratar sentenças expressas com essa linguagem.

E Oferecer uma interface simplificada de alto nível de abstração para um cliente acessar os serviços oferecidos por um subsistema.

Parabéns! A alternativa D está correta.

A alternativa A se aproxima da definição do padrão Observer. A alternativa B se parece com a definição do padrão Strategy, exceto pelo fato que ele não se aplica somente a algoritmos de interpretação de dados. As alternativas C e E correspondem, respectivamente, aos padrões Flyweight e Facade.

Questão 2

Assinale a alternativa que expressa a intenção do padrão de projeto Template Method:

A Permitir a instanciação de objetos a partir da geração de uma cópia de um objeto, fazendo com que o módulo cliente não precise conhecer a classe específica que está sendo instanciada.

B Implementar uma família de algoritmos que possuam uma estrutura comum de passos, de modo que essa estrutura seja definida apenas na superclasse, com os passos

específicos sendo implementados em métodos das subclasses.

C Definir um método padrão para salvar e restaurar o estado de um objeto, sem quebrar o seu encapsulamento.

D Fornecer uma forma padronizada de utilização de classes fornecidas por terceiros e que não podem ser alteradas.

E Permitir a adição de novas funcionalidades, por meio da utilização de classes ortogonais para uma estrutura tipicamente hierárquica de objetos.

Parabéns! A alternativa B está correta.

A alternativa A corresponde ao propósito do padrão Clone. A alternativa C se aproxima do propósito do padrão Memento, porém, o seu objetivo não é fornecer um método padrão, e sim uma forma de salvar e restaurar o estado de um objeto. A alternativa D não está correta, pois o template method não está relacionado à utilização de classes de terceiros. A alternativa E corresponde ao propósito do padrão Visitor.

Considerações finais

Neste conteúdo, vimos como os padrões de projeto GoF comportamentais podem nos auxiliar a distribuir as responsabilidades do sistema em classes mais coesas e menos acopladas. Muitos projetos implementados em Java são difíceis de manter. Uma causa frequente para esse problema é a concentração da complexidade do sistema em poucas classes com milhares de linhas de código, o que torna a estrutura frágil e de difícil entendimento, prejudicando a automação de testes e a evolução do sistema.

Os padrões Command, Strategy, State e Visitor encapsulam operações em objetos. Esse é o ponto principal da abordagem de descentralização do processamento em objetos, transformando operações de classes

gigantescas em classes com um propósito bem definido. Lembre-se de que a essência do projeto orientado a objetos consiste em dividir a complexidade em objetos simples e com um propósito bem definido. A complexidade de um projeto orientado a objetos reside na forma com que os objetos estão interconectados, o que nos dá muito mais flexibilidade para evoluir o sistema, pois a ênfase da evolução passa a ser modificar as conexões e acrescentar novos objetos ao invés de modificar classes monolíticas e complexas.

Os padrões Chain of Responsibility, Iterator, Mediator, Memento e Observer tratam de diferentes problemas, porém, todos possuem o espírito de reduzir o acoplamento entre os módulos de um sistema por meio do emprego de abstrações e generalizações. Produzir soluções genéricas é um dos maiores desafios de um projeto de *software*, e, normalmente, a recompensa é um enorme ganho de produtividade na evolução do produto.

Podcast

Ouça 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 microsserviços.

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