Padrões de projeto criacionais

Os padrões de projeto criacionais (creational patterns) estão associados diretamente aos mecanismos que controlam a criação de objetos. Em alguns projetos, criar um objeto pode ser algo problemático, possuindo algumas particularidades que levam a um alto e desnecessário nível de complexidade.

## Factory method:

O Factory Method é um padrão de projeto (design pattern) que pertence à categoria dos padrões de criação. Ele fornece uma abordagem para a criação de objetos, permitindo que uma classe delegue a responsabilidade da instanciação para suas subclasses.

Em vez de uma classe criar diretamente um objeto, o Factory Method define um método que as subclasses podem implementar para criar e retornar uma instância adequada de um determinado tipo de objeto. Dessa forma, a classe cliente que utiliza o Factory Method não precisa conhecer a classe concreta do objeto que está sendo criado, mas apenas a interface comum que todas as subclasses do objeto devem implementar.

O Factory Method é um padrão de projeto (design pattern) que pertence à categoria dos padrões de criação. Ele fornece uma abordagem para a criação de objetos, permitindo que uma classe delegue a responsabilidade da instanciação para suas subclasses.

Em vez de uma classe criar diretamente um objeto, o Factory Method define um método que as subclasses podem implementar para criar e retornar uma instância adequada de um determinado tipo de objeto. Dessa forma, a classe cliente que utiliza o Factory Method não precisa conhecer a classe concreta do objeto que está sendo criado, mas apenas a interface comum que todas as subclasses do objeto devem implementar.

## Abstract Factory:

O Abstract Factory envolve a criação de uma interface abstrata, conhecida como "fábrica abstrata", que declara métodos para criar objetos relacionados. Cada família de objetos tem uma fábrica concreta correspondente que implementa a interface abstrata. Essas fábricas concretas são responsáveis por criar objetos específicos que compartilham uma semântica comum.

Ao utilizar o Abstract Factory, o cliente pode criar objetos através da interface da fábrica abstrata, sem conhecer as classes concretas dos objetos que estão sendo criados. Isso oferece maior flexibilidade, permitindo que diferentes famílias de objetos sejam utilizadas em tempo de execução.

Esse padrão é útil quando existe a necessidade de trabalhar com múltiplas famílias de objetos relacionados e a lógica de criação desses objetos precisa ser isolada e encapsulada. Além disso, o Abstract Factory também promove o princípio do "Open/Closed", permitindo que novas famílias de objetos sejam adicionadas sem modificar o código existente.

## Builder:

O objetivo principal do Builder é separar a construção de um objeto complexo de sua representação, de modo que o mesmo processo de construção possa ser utilizado para criar diferentes representações do objeto. Isso permite que o código do cliente seja desacoplado das etapas de construção específicas e simplifica a criação de objetos com configurações diferentes.

O Builder envolve a criação de uma classe Builder que define métodos para configurar os diferentes aspectos do objeto a ser construído. Esses métodos são chamados de forma encadeada, permitindo uma sequência clara e flexível de configurações. A classe Builder mantém o estado atual da construção do objeto.

Além do Builder, é comum ter uma classe Director, que é responsável por controlar a sequência de passos de construção usando o Builder. O Director recebe uma instância do Builder e chama seus métodos de configuração na ordem correta para construir o objeto desejado.

Ao utilizar o Builder, o cliente pode criar um objeto complexo passo a passo, configurando apenas as opções necessárias. Isso evita a necessidade de criar construtores com muitos parâmetros ou construtores com parâmetros opcionais, tornando o código mais legível e flexível.

## Prototype:

O Prototype envolve a criação de uma interface comum ou classe base, conhecida como "protótipo", que define um método de clonagem. Cada classe concreta que implementa o protótipo pode ter sua própria lógica de clonagem, fornecendo uma cópia profunda do objeto.

Ao utilizar o Prototype, o cliente pode criar novos objetos através da clonagem de um protótipo existente, em vez de criar um objeto do zero. Isso é feito chamando o método de clonagem do protótipo e, em seguida, realizando as modificações necessárias no objeto clonado, se necessário.

É importante ressaltar que o Prototype é diferente da simples cópia de um objeto. Ele envolve a clonagem de um objeto de maneira que a cópia seja independente do original e possa ser modificada sem afetar o objeto original.

## Singleton:

O Singleton é frequentemente usado quando é necessário controlar o acesso a um recurso compartilhado, como uma conexão de banco de dados, um gerenciador de arquivos ou uma configuração global. Ao restringir a criação de instâncias da classe a apenas uma, o Singleton garante que todos os clientes utilizem a mesma instância, evitando duplicações e inconsistências.

A implementação do Singleton envolve a definição de um construtor privado na classe, impedindo que outras classes criem instâncias diretas. Em vez disso, a classe Singleton fornece um método estático, geralmente chamado de "getInstance()", que retorna a única instância da classe. Esse método é responsável por criar a instância se ainda não existir e retorná-la.

O Singleton também pode armazenar estado compartilhado entre os clientes, mantendo variáveis de instância ou referências relevantes ao recurso compartilhado. Isso permite que os clientes acessem e modifiquem o estado por meio da instância Singleton.

É importante notar que a implementação correta do Singleton requer atenção especial para lidar com concorrência e multithreading, para evitar problemas de acesso simultâneo à instância. Também é importante ter cuidado para não abusar do Singleton, pois o acoplamento global pode dificultar a manutenção e o teste do código.

Padrões de projeto estruturais

Os padrões de projetos estruturais são uma categoria de padrões de projeto que se concentram na organização, composição e relacionamento entre classes e objetos. Eles visam facilitar o design de sistemas flexíveis, reutilizáveis e de fácil manutenção, fornecendo soluções para problemas comuns relacionados à estrutura do código.

## Adapter:

O objetivo principal do Adapter é permitir que classes com interfaces diferentes possam colaborar entre si sem precisar modificar seu código-fonte. Isso é útil quando se deseja reutilizar uma classe existente que não é compatível com a interface necessária pelo cliente.

A implementação do Adapter envolve a criação de uma classe intermediária, chamada de "adapter", que implementa a interface esperada pelo cliente. O adapter então se comunica com a classe incompatível, chamada de "adaptee", convertendo as chamadas e os dados da interface do cliente para a interface do adaptee.

Dessa forma, o cliente pode interagir com o adapter como se estivesse trabalhando com a interface desejada, sem precisar conhecer os detalhes internos do adaptee. O adapter encapsula a lógica de adaptação e garante a compatibilidade entre as classes.

O Adapter pode ser implementado de duas maneiras: adaptador de objeto e adaptador de classe. No adaptador de objeto, o adapter possui uma instância do adaptee e redireciona as chamadas para ela. No adaptador de classe, o adapter herda do adaptee e faz a adaptação por meio de herança.

O Adapter é frequentemente utilizado quando se integra um novo código com sistemas legados ou quando se deseja utilizar bibliotecas ou componentes de terceiros com uma interface diferente. Ele permite que classes incompatíveis trabalhem juntas de forma harmoniosa, facilitando a reutilização e a manutenção do código.

## Bridge:

O objetivo principal do Bridge é promover a flexibilidade, a extensibilidade e a desacoplagem entre a abstração e a implementação. Ele permite que as duas partes possam evoluir de maneira independente, sem que uma alteração em uma delas exija modificações na outra.

A implementação do Bridge envolve a criação de duas hierarquias de classes: uma para a abstração e outra para a implementação. A abstração possui uma referência para um objeto da implementação, delegando as chamadas de métodos para ele. Dessa forma, diferentes implementações podem ser associadas à mesma abstração, permitindo combinações flexíveis.

Através do Bridge, é possível adicionar novas abstrações ou implementações sem modificar as classes existentes. Isso oferece maior flexibilidade para o sistema, permitindo que novos recursos sejam adicionados sem afetar o código existente.

O Bridge é frequentemente utilizado em casos em que existem múltiplas dimensões de variação em um sistema, como diferentes plataformas, protocolos ou modos de exibição. Ele também é útil quando se deseja evitar uma explosão de subclasses, que ocorreria se todas as combinações de abstrações e implementações fossem implementadas diretamente.

## Composite:

O objetivo principal do Composite é fornecer uma estrutura flexível para criar objetos compostos, onde um objeto pode conter outros objetos como partes. Essa estrutura de árvore permite que os clientes acessem e manipulem os objetos da composição de maneira unificada, sem se preocupar se estão manipulando um objeto individual ou um grupo de objetos.

A implementação do Composite envolve a criação de uma interface ou classe abstrata que representa tanto os objetos individuais quanto os grupos de objetos. Essa interface define métodos comuns que permitem a adição, remoção e acesso aos elementos da composição. As classes concretas implementam essa interface, fornecendo a lógica específica para os objetos individuais e os grupos de objetos.

Ao utilizar o Composite, os clientes podem tratar um objeto individual da mesma maneira que tratam um grupo de objetos. Eles podem percorrer a estrutura hierárquica, acessar os elementos e realizar operações neles de forma transparente.

O Composite é útil em situações onde existe uma hierarquia de objetos que pode ser tratada de forma uniforme. Ele é amplamente utilizado em interfaces gráficas de usuário, onde elementos individuais, como botões e caixas de texto, podem ser agrupados em componentes maiores, como painéis e janelas.

## Decorator:

O objetivo principal do Decorator é fornecer uma alternativa flexível à herança tradicional para estender a funcionalidade de um objeto. Em vez de criar subclasses para cada combinação possível de funcionalidades, o Decorator permite a combinação e a aplicação dinâmica dessas funcionalidades usando composição.

A implementação do Decorator envolve a criação de uma classe base abstrata que define a interface comum para o objeto original e os decoradores. Tanto o objeto original quanto os decoradores implementam essa interface. Os decoradores também possuem uma referência para o objeto original, permitindo que chamadas de métodos sejam encadeadas e modificadas.

Ao utilizar o Decorator, os objetos podem ser envoltos por um ou mais decoradores, adicionando responsabilidades extras em tempo de execução. Esses decoradores podem adicionar novos comportamentos antes, depois ou até mesmo substituir os comportamentos do objeto original.

O Decorator é útil quando se deseja adicionar funcionalidades de forma flexível e modular, evitando uma explosão de subclasses. Ele é comumente usado em cenários em que se requer a adição de recursos extras a um objeto, como a adição de formatação, criptografia, logging ou controle de acesso.

## Facade:

O objetivo principal do Facade é fornecer uma interface de nível superior que seja mais fácil de usar e entender do que as interfaces individuais das classes ou subsistemas subjacentes. Ele simplifica a interação com o sistema, fornecendo métodos e operações de alto nível que encapsulam as operações complexas e reduzem a dependência do cliente em relação à estrutura interna do sistema.

A implementação do Facade envolve a criação de uma classe que atua como uma fachada para o sistema. Essa classe conhece as classes e subsistemas internos e direciona as chamadas do cliente para as partes apropriadas do sistema. O Facade pode fornecer métodos simples que escondem a complexidade subjacente e realizam uma série de operações em conjunto.

Ao utilizar o Facade, os clientes não precisam conhecer os detalhes de implementação do sistema. Eles interagem apenas com a interface fornecida pela fachada, que cuida de toda a complexidade interna. Isso simplifica o uso do sistema, reduz o acoplamento e torna o código mais legível e fácil de manter.

O Facade é útil em situações em que um sistema é composto por várias classes ou subsistemas complexos e é desejável fornecer uma interface mais simples e intuitiva para os clientes. Ele também é útil quando se deseja desacoplar o cliente da implementação do sistema, fornecendo uma camada de abstração entre eles.

## Flyweight:

O objetivo principal do Flyweight é reduzir o consumo de memória armazenando informações compartilhadas em objetos compartilhados, em vez de replicá-las em cada objeto individualmente. Ele separa as características intrínsecas e extrínsecas dos objetos, armazenando as características intrínsecas de forma compartilhada e mantendo as características extrínsecas separadas.

A implementação do Flyweight envolve a criação de uma classe flyweight que contém as características intrínsecas compartilhadas. Em seguida, um gerenciador de flyweights é usado para rastrear e fornecer os flyweights quando necessário. Os objetos cliente possuem referências para os flyweights, que podem ser compartilhados entre vários clientes.

Ao utilizar o Flyweight, é possível reduzir significativamente a quantidade de memória necessária, pois os objetos compartilham as mesmas características intrínsecas. As características extrínsecas podem ser fornecidas ou calculadas pelos objetos cliente quando necessário, mantendo a individualidade de cada objeto.

O Flyweight é útil quando há um grande número de objetos semelhantes e a quantidade de memória disponível é limitada. Ele é comumente usado em situações em que é necessário criar e manipular muitos objetos, como em jogos, sistemas gráficos ou aplicações que lidam com grandes volumes de dados.

## Proxy:

O objetivo principal do Proxy é controlar o acesso e adicionar funcionalidades adicionais a um objeto sem que o cliente precise estar ciente disso. Ele permite que o proxy assuma a responsabilidade de gerenciar a complexidade associada ao objeto real, fornecendo uma interface simplificada para o cliente.

A implementação do Proxy envolve a criação de uma classe proxy que implementa a mesma interface do objeto real. O proxy possui uma referência para o objeto real e controla as chamadas de método, podendo executar ações adicionais antes ou depois de repassar a chamada ao objeto real.

O Proxy pode ser usado para diversos propósitos, como controle de acesso, criação tardia de objetos, cache, registro de chamadas e monitoramento. Ele permite que o cliente trabalhe com o proxy sem conhecer ou interagir diretamente com o objeto real, fornecendo uma camada de abstração e controle.

O Proxy é útil em situações em que é necessário adicionar funcionalidades extras a um objeto, controlar o acesso a ele ou reduzir a complexidade do sistema. Ele também é útil quando é necessário atrasar a criação de objetos até que sejam realmente necessários.

Padrões de projeto comportamentais

Padrões de projeto comportamentais são um conjunto de soluções para problemas recorrentes relacionados ao comportamento e interações entre objetos em um sistema de software. Esses padrões se concentram em como classes e objetos se comunicam e interagem uns com os outros, buscando aumentar a flexibilidade, a reutilização e a legibilidade do código.

## Chain of Responsibility:

O objetivo principal do Chain of Responsibility é desacoplar o remetente de uma solicitação do seu destinatário, dando a vários objetos a chance de processar a solicitação de forma independente. Ele promove a flexibilidade e a extensibilidade, permitindo que diferentes objetos lidem com diferentes tipos de solicitações sem a necessidade de modificar o remetente ou a classe dos destinatários.

A implementação do Chain of Responsibility envolve a criação de uma cadeia de objetos, onde cada objeto possui uma referência para o próximo objeto na cadeia. O objeto que recebe a solicitação decide se deve processá-la ou passá-la para o próximo objeto da cadeia. A cadeia continua até que um objeto seja capaz de lidar com a solicitação ou até que a solicitação seja descartada.

Ao utilizar o Chain of Responsibility, é possível evitar o acoplamento rígido entre remetentes e destinatários, permitindo que a solicitação seja tratada por diferentes objetos de maneira flexível. Cada objeto na cadeia tem a oportunidade de tratar a solicitação de acordo com sua lógica específica, aumentando a modularidade e a reutilização do código.

O Chain of Responsibility é útil em situações em que várias opções de tratamento de uma solicitação estão disponíveis e a decisão sobre qual objeto deve processar a solicitação é determinada em tempo de execução. É comumente usado em sistemas de eventos, manipuladores de erros, filtros e validações.

## Command:

O objetivo principal do Command é separar o emissor de uma solicitação do objeto que a executa. Isso permite que as solicitações sejam tratadas como objetos independentes, que podem ser gerenciados, manipulados e transmitidos de maneira flexível dentro do sistema.

A implementação do Command envolve a criação de uma interface comum que define um método execute(), que representa a ação a ser executada. Em seguida, uma classe concreta é criada para cada solicitação, implementando essa interface e encapsulando os detalhes da solicitação e do objeto receptor.

Ao utilizar o Command, o cliente cria um objeto de comando e o associa a um objeto receptor. O cliente pode executar o comando chamando o método execute(), que será redirecionado para o objeto receptor apropriado.

O Command é útil em situações em que é necessário desacoplar o emissor de uma solicitação de seu receptor, permitindo que as solicitações sejam manipuladas, gerenciadas e transmitidas de maneira flexível. Ele também é útil quando se deseja enfileirar ou registrar solicitações, suportar operações de desfazer ou refazer e fornecer uma maneira uniforme de tratar diferentes tipos de solicitações.

## Iterator:

O objetivo principal do Iterator é fornecer uma interface comum para percorrer os elementos de uma coleção, independentemente do tipo específico de coleção. Isso promove a flexibilidade e a reutilização do código, permitindo que diferentes algoritmos percorram uma coleção de maneira consistente.

A implementação do Iterator envolve a criação de uma interface Iterator, que define métodos como hasNext() para verificar se há elementos restantes, next() para retornar o próximo elemento e remove() para remover o elemento atual da coleção. Cada classe de coleção concreta implementa essa interface, fornecendo sua própria implementação específica para acessar seus elementos.

Ao utilizar o Iterator, o cliente obtém um objeto Iterator da coleção e, em seguida, usa esse objeto para percorrer os elementos da coleção. O cliente não precisa conhecer os detalhes de como os elementos estão armazenados ou organizados na coleção, pois o Iterator cuida disso.

O Iterator é útil em situações em que é necessário percorrer os elementos de uma coleção sem expor sua estrutura interna. Ele simplifica o código do cliente, promove a flexibilidade e a reutilização, além de permitir que diferentes algoritmos percorram a mesma coleção de maneira consistente.

## Mediator:

O objetivo principal do Mediator é reduzir a dependência direta entre os objetos, permitindo que eles se comuniquem de maneira indireta por meio do mediador. Isso evita que os objetos tenham conhecimento uns dos outros e, assim, torna o sistema mais flexível, extensível e fácil de manter.

A implementação do Mediator envolve a criação de uma classe Mediator que define a interface para comunicação entre objetos. Os objetos participantes têm uma referência ao mediador e usam essa referência para se comunicar entre si. O mediador coordena a interação entre os objetos e direciona as mensagens corretamente.

Ao utilizar o Mediator, os objetos participantes não precisam ter conhecimento direto uns dos outros. Em vez disso, eles se comunicam indiretamente por meio do mediador, que controla a lógica de comunicação e coordenação. Isso reduz o acoplamento e facilita a adição, remoção ou modificação de objetos participantes sem afetar o restante do sistema.

O Mediator é útil em situações em que existe um conjunto complexo de interações entre objetos e é desejável reduzir o acoplamento entre eles. É comumente usado em sistemas de GUI, onde vários componentes precisam se comunicar e responder a eventos de forma coordenada.

## Memento:

O objetivo principal do Memento é fornecer uma maneira de salvar e restaurar o estado de um objeto, permitindo que ele retorne a um estado anterior sem revelar detalhes internos ou violar o encapsulamento. Isso é útil quando se deseja implementar recursos como desfazer/refazer ou checkpoints em um sistema.

A implementação do Memento envolve três componentes principais: o Originator (originador), que é o objeto cujo estado será salvo; o Memento, que representa o estado salvo do Originator; e o Caretaker (zelador), que é responsável por solicitar e armazenar os Mementos.

Ao utilizar o Memento, o Originator cria um Memento que contém uma cópia do seu estado interno. Esse Memento pode ser fornecido ao Caretaker, que pode armazená-lo em uma estrutura adequada, como uma pilha, para suportar operações de desfazer/refazer. Em algum momento, o Originator pode solicitar ao Caretaker a restauração do seu estado a partir de um Memento específico.

O Memento é útil em situações em que é necessário salvar e restaurar o estado de um objeto, permitindo que ele retorne a um estado anterior sem expor sua implementação interna. Ele é comumente usado em sistemas que exigem recursos de desfazer/refazer ou checkpoints, onde a captura e restauração de estados são fundamentais.

## Observer:

O objetivo principal do Observer é permitir que objetos se comuniquem de forma eficiente e desacoplada, garantindo que as mudanças de estado em um objeto sejam refletidas em outros objetos interessados ​​sem que haja acoplamento direto entre eles.

A implementação do Observer envolve dois componentes principais: o Subject (sujeto) e o Observer (observador). O Subject mantém uma lista de observadores registrados e fornece métodos para adicionar, remover e notificar os observadores. O Observer registra-se no Subject para receber notificações e implementa um método de atualização que é chamado pelo Subject quando ocorrem mudanças.

Ao utilizar o Observer, os objetos interessados ​​registram-se como observadores do objeto sujeito. Quando o objeto sujeito sofre uma alteração de estado, ele notifica automaticamente todos os observadores registrados, que podem atualizar seu estado de acordo.

O Observer é útil em situações em que há uma relação de dependência um-para-muitos entre objetos e é necessário garantir que as mudanças de estado em um objeto sejam propagadas e atualizadas em outros objetos dependentes. É comumente usado em sistemas de eventos, interfaces gráficas de usuário, distribuição de notificações e em situações em que a sincronização entre objetos é necessária.

## State:

O objetivo principal do State é permitir que um objeto tenha diferentes comportamentos com base no seu estado interno, sem a necessidade de grandes estruturas condicionais ou lógica complexa. Isso promove a flexibilidade, a extensibilidade e o princípio de aberto/fechado, pois novos estados podem ser adicionados sem modificar o código existente.

A implementação do State envolve a criação de uma hierarquia de classes de estado, onde cada classe representa um estado específico do objeto. Essas classes implementam uma interface comum, que define os métodos relacionados ao comportamento do objeto. O objeto principal possui uma referência para o estado atual e delega a chamada dos métodos para o estado correspondente.

Ao utilizar o State, o objeto principal pode alterar seu estado interno durante a execução, alterando o objeto de estado atual. Isso permite que o objeto tenha comportamentos diferentes, dependendo do estado em que se encontra. As transições de estado podem ser controladas pelo objeto principal ou pelo próprio estado, dependendo da implementação.

O State é útil em situações em que um objeto pode ter diferentes comportamentos com base no seu estado interno e a lógica condicional se torna complexa ou difícil de manter. É comumente usado em sistemas de gerenciamento de fluxo de trabalho, máquinas de estado, interfaces gráficas e em qualquer situação em que o comportamento do objeto precise ser adaptável e extensível.

## Strategy:

O objetivo principal do Strategy é encapsular diferentes algoritmos em classes separadas, chamadas de estratégias, e permitir que o objeto cliente selecione e utilize uma estratégia específica de acordo com a necessidade. Isso evita a duplicação de código e facilita a substituição de algoritmos sem afetar o objeto cliente.

A implementação do Strategy envolve a criação de uma interface comum para as estratégias, que define os métodos que serão utilizados pelo objeto cliente. Cada estratégia concreta implementa essa interface, fornecendo sua própria implementação específica do algoritmo. O objeto cliente possui uma referência para a estratégia atual e utiliza seus métodos para executar o algoritmo correspondente.

Ao utilizar o Strategy, o objeto cliente pode selecionar e trocar a estratégia em tempo de execução, alterando o comportamento do objeto sem necessidade de modificação no código existente. Isso permite a adaptação do comportamento de acordo com a situação ou os requisitos específicos.

O Strategy é útil em situações em que diferentes algoritmos precisam ser aplicados dinamicamente em um objeto sem a necessidade de alterar o código existente. É comumente utilizado em sistemas que requerem várias opções de algoritmos, personalização de comportamento, redução de código duplicado e extensibilidade.

## Template method:

O objetivo principal do Template Method é fornecer uma estrutura para a implementação de algoritmos em várias etapas, onde a sequência geral de passos é definida na classe base, mas os detalhes de cada etapa podem ser implementados de forma flexível pelas subclasses.

A implementação do Template Method envolve a criação de uma classe base abstrata que contém o método de template, que define a sequência de passos do algoritmo. Alguns desses passos podem ser implementados diretamente na classe base, enquanto outros são declarados como métodos abstratos e deixados para as subclasses concretas.

Ao utilizar o Template Method, as subclasses herdam o algoritmo definido na classe base e podem substituir os métodos abstratos para fornecer a implementação específica de cada etapa do algoritmo. Dessa forma, o algoritmo geral é mantido consistente e reutilizável, enquanto os detalhes específicos podem ser personalizados em cada subclasse.

O Template Method é útil em situações em que existe um algoritmo comum que precisa ser aplicado de maneira consistente, mas com variações específicas em certas etapas. Ele promove a reutilização de código, evita duplicação de código e facilita a extensibilidade do algoritmo sem modificar sua estrutura geral.

## Visitor:

O objetivo principal do Visitor é fornecer uma forma de adicionar novas operações a uma estrutura de objetos existente sem modificar as classes desses objetos. Ele permite que você adicione novos comportamentos sem alterar a estrutura do objeto, promovendo o princípio de aberto/fechado e evitando a necessidade de modificar o código existente.

A implementação do Visitor envolve a criação de duas hierarquias de classes: uma hierarquia de elementos, que representa os objetos da estrutura, e uma hierarquia de visitantes, que define as operações que podem ser executadas nesses elementos. Cada elemento possui um método de aceitação que permite que um visitante execute a operação correspondente.

Ao utilizar o Visitor, você pode percorrer a estrutura de objetos chamando o método de aceitação em cada elemento. O elemento então redireciona a chamada para o visitante apropriado, que executa a operação específica para aquele elemento. Isso permite que você adicione novos visitantes para executar operações adicionais sem modificar as classes de elementos existentes.

O Visitor é útil em situações em que você tem uma estrutura de objetos complexa e precisa executar operações variadas nessa estrutura. Ele é especialmente útil quando você deseja adicionar novas operações à estrutura sem modificar as classes dos elementos existentes. O Visitor é comumente usado em compiladores, processadores de árvores de sintaxe, análise de dados e qualquer situação em que seja necessário percorrer e executar operações em uma estrutura de objetos complexa.