Grupo 10:
Anderson Job Loeffler, Augusto Cesar Camargo, Bruno Haupenthal Martins, Ezequiel Santos Bitencourt, Leonardo Flores Bernardo, Thales da Silva Caruso Madera

Padrão GoF

Criado em 1995 por Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides, o padrão GoF pode ser considerado como um repertório de soluções e princípios que ajudam os desenvolvedores a criar software, e que são codificados em um formato estruturado, consistindo de nome, onde se encontra a descrição da essência do padrão, o problema, que descreve quando e em que condições aplicar o padrão, e a solução, que consiste na descrição de classes e objetos usados para solucionar o problema.

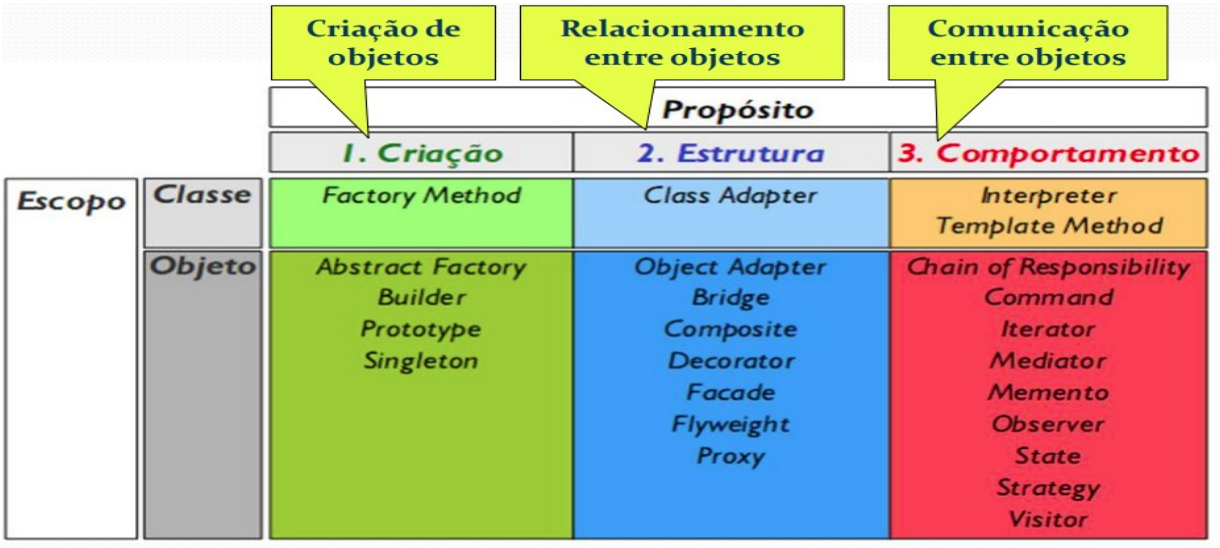
O padrão GoF está dividido em 3 seções:

- Padrão de criação, ligado ao processo de instanciação e relacionado a classes e objetos.

- Padrão Estrutural, relacionado a alteração da estrutura de um programa, e de associações de classes e objetos

- Padrão Comportamental,que consiste na observação da maneira que as classes e objetos interagem.

Classificação dos 23 padrões segundo GoF:



Adapter

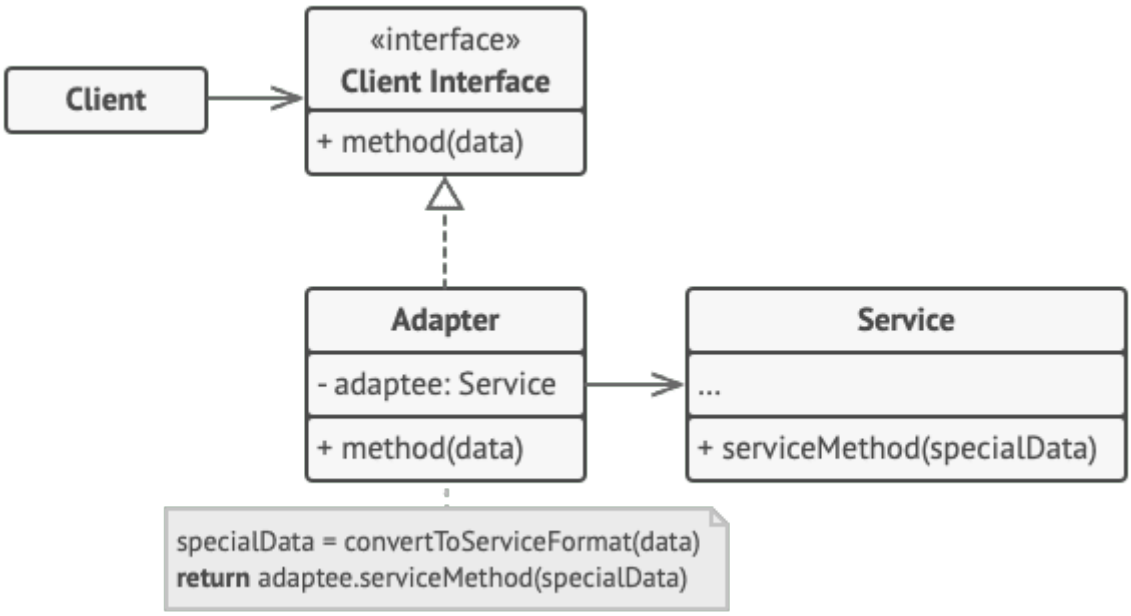
O padrão estrutural *adapter* faz com que uma interface adaptada (*adaptee*) seja compatível com outra, garantindo abstrações uniformes das diferentes interfaces de um projeto. A classe adaptadora (*adapter*), herda uma classe adaptada, e exprime sua interface.

O objetivo desse padrão é converter a interface de uma classe em outra interface, esperada pelos clientes, solucionando problemas de incompatibilidade.

O funcionamento ocorre da seguinte forma: o adaptador usa uma interface compatível com um dos objetos existentes; usando-a, o objeto existente pode tranquilamente chamar os métodos do adaptador; através de uma chamada, o adaptador passa a requisição para um segundo objeto, mas em um formato e ordenação compatíveis com o segundo objeto.

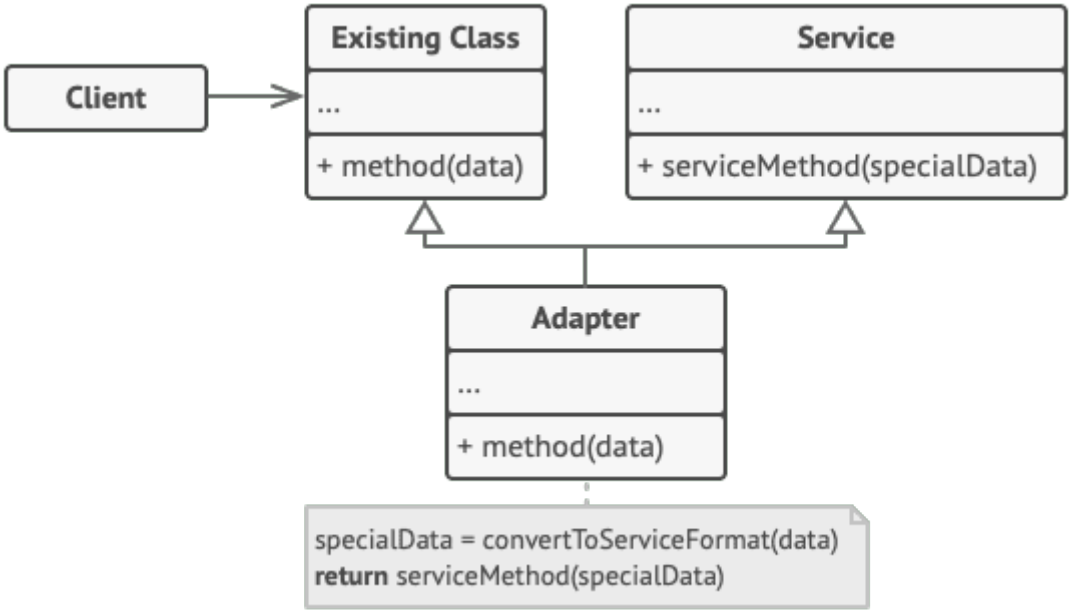
Para ilustrar de melhor forma, podemos imaginar um cenário onde uma aplicação recebe dados de uma API e usa-os em uma biblioteca de análise de dados. Vamos supor que a api entrega os dados em formato JSON e a biblioteca de análise de dados leia os dados em XML. Poderíamos então, dentro da aplicação, criar um adapter que converta dados JSON para XML. Lembrando que o adapter não é usado somente para manipular dados, mas também funcionalidades.

Exemplo de Object Adapter:



No exemplo acima, o Service pode ser visto como uma classe externa que é incompatível com o Client diretamente. Por isso a classe Adapter teve que ser desenvolvida para que possa “conversar” com as duas funcionalidades. O Adapter implementa a interface de Client enquanto envolve o objeto Service.

Exemplo de Class Adapter:



No exemplo de adaptador de classe acima, não se envolve nenhum objeto, pois ela herda comportamentos tanto do Client quanto do Service. A adaptação ocorre dentro do *override* dos métodos.

Bridge

A bridge é um padrão que permite a separação de uma grande classe, ou um conjunto de classes intimamente ligadas em duas hierarquias separadas: abstração e implementação, onde cada uma pode ser desenvolvida de forma independente uma da outra.

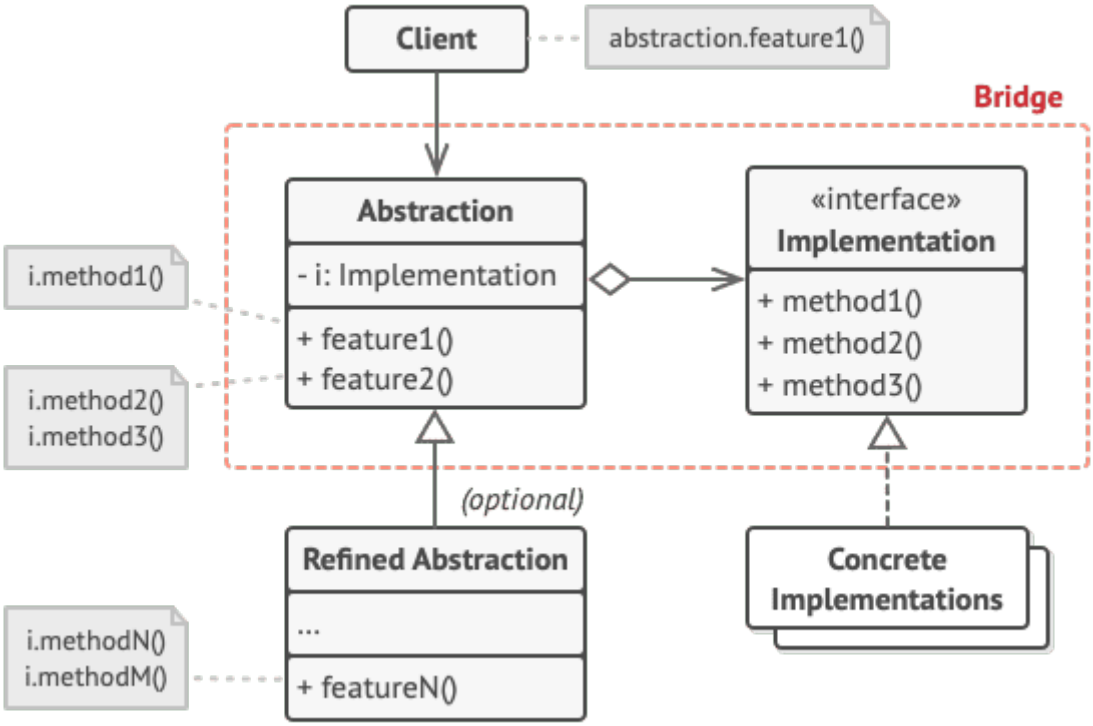
Às vezes somente a abordagem de interfaces e heranças não são suficientemente flexíveis. A herança liga uma implementação à abstração permanente, o que dificulta a modificação, aumento e reutilização de abstrações e implementações independentemente.

Imaginemos que exista uma classe chamada Forma que tenha um par de subclasses Círculo e Quadrado. Vamos supor que precisamos estender a hierarquia para incorporação de cores, onde criamos subclasses Azul e Vermelho. Entretanto, levando em consideração que já temos duas subclasses, precisaríamos criar quatro combinações, CirculoAzul, CirculoVermelho, QuadradoAzul e QuadradoVermelho. Ao adicionar outras formas e cores, as classes cresceriam de forma exponencial.

O problema ocorre porque estamos tentando estender classes em duas dimensões diferentes: forma e cor. O bridge soluciona esse problema substituindo herança por composição do objeto. Isso significa que deve-se separar uma das dimensões em uma classe separada, para que as classes originais referenciem o objeto na nova hierarquia, em vez de conterem todo o estado e comportamento em uma só classe.

Um exemplo da bridge poderia ser o funcionamento de uma aplicação multiplataforma, onde a camada de abstração seria a camada GUI e a implementação a API do sistema operacional. O objeto de abstração controla a aparência do app, delegando trabalho para o objeto de implementação ligado. Diferentes implementações funcionam de forma interligada desde que sigam uma interface em comum, permitindo que o mesmo GUI funcione sob os SO's Windows e Linux, por exemplo. Como resultado, pode-se mudar as classes da GUI sem tocar nas classes ligadas a API.

Exemplo de Bridge:



Na imagem acima, observamos que a *Abstraction* possui um controle de lógica de alto nível, onde depende da implementação do objeto para fazer o verdadeiro trabalho. A *Implementation* declara a interface comum para todas as implementações concretas. Uma abstração só pode se comunicar com um objeto de implementação através de métodos que são declarados aqui. *Concrete Implementations* contém código específico da plataforma. *Refined Abstractions* provém variantes do controle de lógica. O *Client* geralmente só está interessado em trabalhar com a camada de abstração.

Composite

A *composite* é um padrão de projeto que permite a composição de objetos em estruturas de árvores, onde cada uma dessas estruturas funciona como se fossem objetos individuais.

O modelo *composite* apenas faz sentido se o core do seu app pode ser representado como uma árvore.

Exemplificando, podemos imaginar dois tipos de objetos, Produtos e Caixas. Uma caixa pode conter vários produtos assim como pode conter caixas

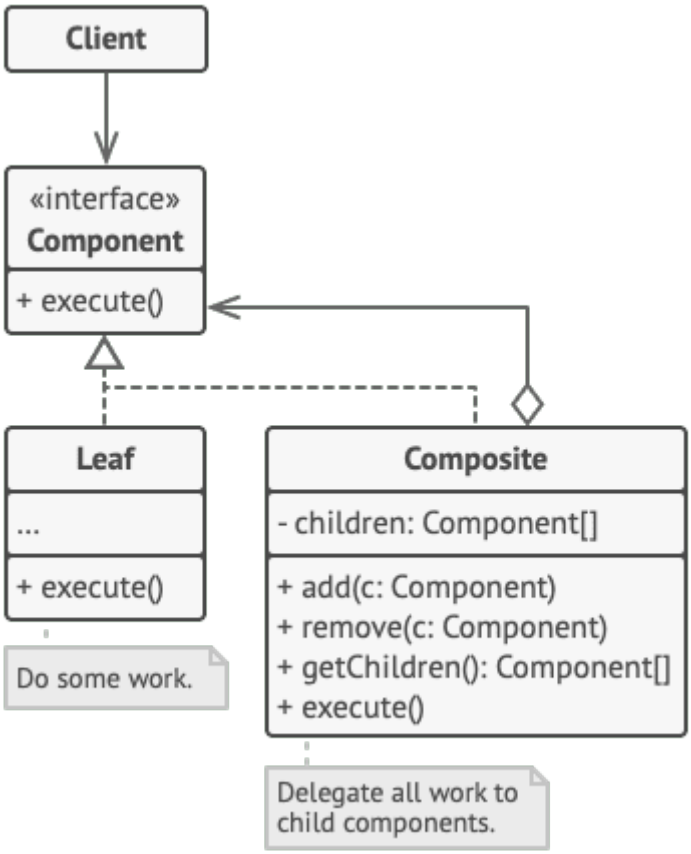
menores. Essas caixas menores podem conter outros produtos ou caixas ainda menores e assim por diante. Supondo que queiramos criar um sistema de entrega com o uso dessas classes. Pedidos podem conter produtos simples sem nenhum invólucro, assim como caixas contendo produtos, e outros produtos. Como determinaria-se o preço total de tal pedido?

Poderíamos pensar na solução mais óbvia, desempacotar as caixas até chegarmos no produto e, então, calcular o total. Isso seria possível no mundo real, mas no caso de um programa, não seria tão simples como rodar um loop.

Usando o padrão composite podemos solucionar o problema trabalhando as classes Produto e Caixa através de uma interface em comum que declara um método que calcula o preço total.

Para o Produto, o método poderia apenas retornar o preço. Já para a classe Caixa o método percorreria cada item que a caixa contém, perguntaria o preço a retornaria o total dessa caixa. Se um desses itens fosse uma caixa menor, ele também percorreria seu conteúdo e assim por diante, até que o preço de todos os componentes internos sejam calculados.

Exemplo de Composite:



Na imagem acima, podemos observar a interface *Component* que descreve operações comuns a elementos simples e complexos da árvore. A *Leaf* é o elemento que não possui subelementos. O *Composite* é o elemento que possui subelementos, ou seja, *Leaf* ou outros *Composite*; ele funciona com todos os subelementos apenas devido à interface *Component*.

Decorator

O *decorator* é um padrão de design que permite a atribuição de novos comportamentos a objetos através do acoplamento desses objetos a invólucros que contêm esses comportamentos.

Imagine uma biblioteca de notificações de importantes eventos para usuários de um sistema. A versão inicial da biblioteca foi baseada na classe Notifier que possuía poucos campos, um construtor e apenas um método send. O método poderia aceitar um argumento de mensagem de um cliente e enviar a mensagem para uma lista de emails que seriam passados para o Notifier através do construtor. Um aplicativo externo que atuaria como como um cliente deveria criar e configurar o objeto notificador uma vez, e então usá-lo cada vez que algo importante acontecesse.

Em dado momento, suponhamos que usuários da biblioteca esperassem mais do que só notificações de e-mail, e esperassem notificações de SMS, Facebook e Whatsapp por exemplo.

Para isso, poderíamos estender a classe Notifier e adicionar métodos de notificação adicionais.

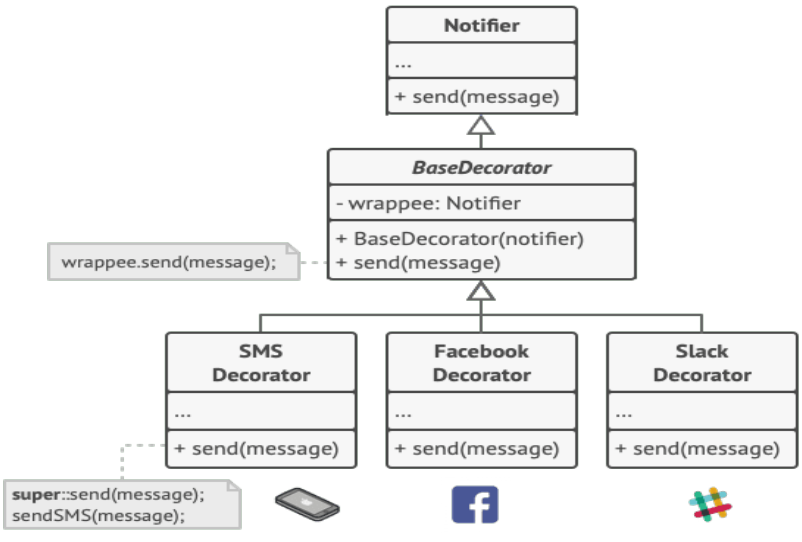
Entretanto, se precisássemos que a notificação aparecesse em diferentes locais ao mesmo tempo, ou apenas dois tipos de notificações ao mesmo tempo, teríamos que estender o Notifier para cada uma dessas ocasiões.

Uma maneira de solucionar esse problema é usando Agregação ou Composição, em vez de Herança. As duas alternativas funcionam da mesma maneira, um objeto tem uma referência a outro e o delega uma tarefa, enquanto que na herança, o próprio objeto é capaz de fazer a tarefa, herdando o comportamento da superclasse.

Com essa nova estratégia podemos substituir o objeto “helper” por outro. Um objeto pode usar o comportamento de várias classes, tendo referências a múltiplos objetos e delegando-os a todo tipo de tarefa. Agregação e Composição são os princípios chave por trás do Decorator.

Podemos então, usar um invólucro que implemente a mesma interface que o objeto a ser envolvido, fazendo com que o campo de referência aceite qualquer objeto que siga a interface. Isso permite que um objeto tenha múltiplos invólucros, adicionando seus comportamentos.

Exemplo de decorator:



Façade

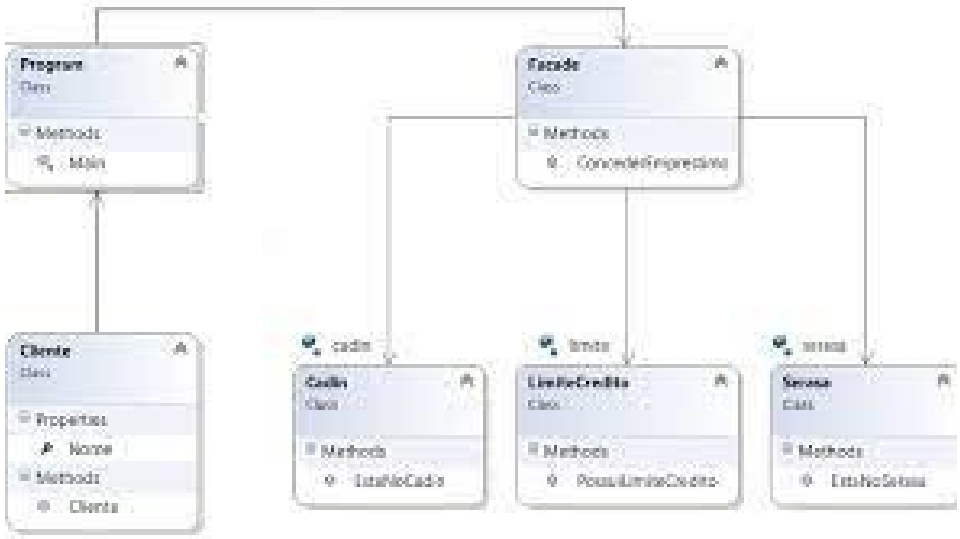
O Padrão Facade que é simples de ser aplicado e que traz grandes benefícios aos projetos é dito como sendo um padrão estrutural e está entre os 23 padrões de projeto do GoF (Gang of Four).

Entende-se por padrão estrutural todo padrão de projeto que trata da associação entre classes e objetos.

Como o nome sugere Facade, é realmente uma fachada, podemos fazer a seguinte analogia, quando caminhamos em frente a um prédio com uma bela fachada, vemos as belas janelas as paredes bem decoradas, ou seja, um ambiente bem amigável, e ignoramos toda a complexidade por trás da obra, a quantidade de salas, todas as empresas que estão neste prédio, deste modo o Facade também age nos projetos de software, dentre seus benefícios, alguns são:

- Reduz a complexidade de uma api, liberando acesso a métodos de alto nível encapsulando os demais.
- Produz uma interface comum e simplificada.
- Pode encapsular uma ou mais interfaces mal projetadas em uma mais concisa.
- Reduz drasticamente o acoplamento entre as camadas do projeto.
-

Exemplo de Façade:

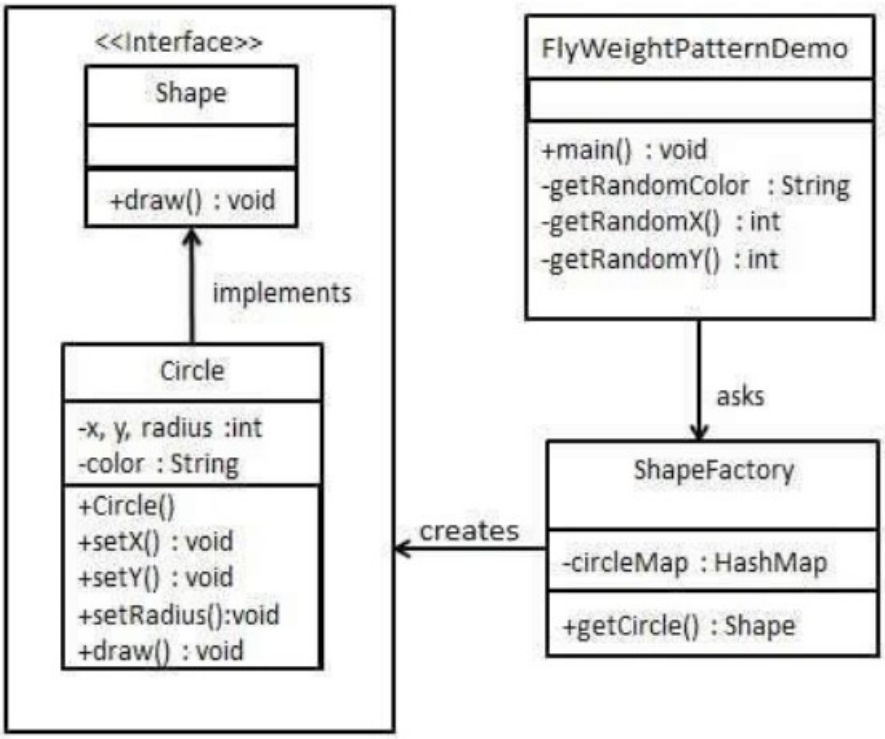


Flyweight

O principal objetivo do padrão de projeto Flyweight é melhorar o desempenho de um procedimento através de compartilhamento de objetos com características similares. Em outras palavras, o padrão provê um mecanismo para utilizar objetos já existentes, modificando suas propriedades conforme solicitado, ao invés da necessidade de sempre instanciá-los. É possível trabalhar com vários objetos de uma só vez.

Quando utilizarei este padrão de projeto?
Algumas literaturas sobre *Design Patterns* mencionam que o *Flyweight* é utilizado em casos bem específicos, o que implica na sua baixa utilização. No entanto, jamais será um padrão de projeto desdenhado. Um exemplo clássico, comum de se encontrar na internet sobre o *Flyweight*, é a reutilização de objetos de caracteres em processadores de texto. Ao invés de criar um objeto para cada caractere digitado – no qual carrega informações como fonte, tamanho, cor e formato – utiliza-se o *Flyweight* para compartilhá-los, reduzindo bastante o consumo de memória. Outro exemplo são exploradores de arquivos que podem compartilhar objetos do mesmo tipo de arquivo para reduzir o tempo de carga e exibição.

Exemplo de Flyweight:



Proxy

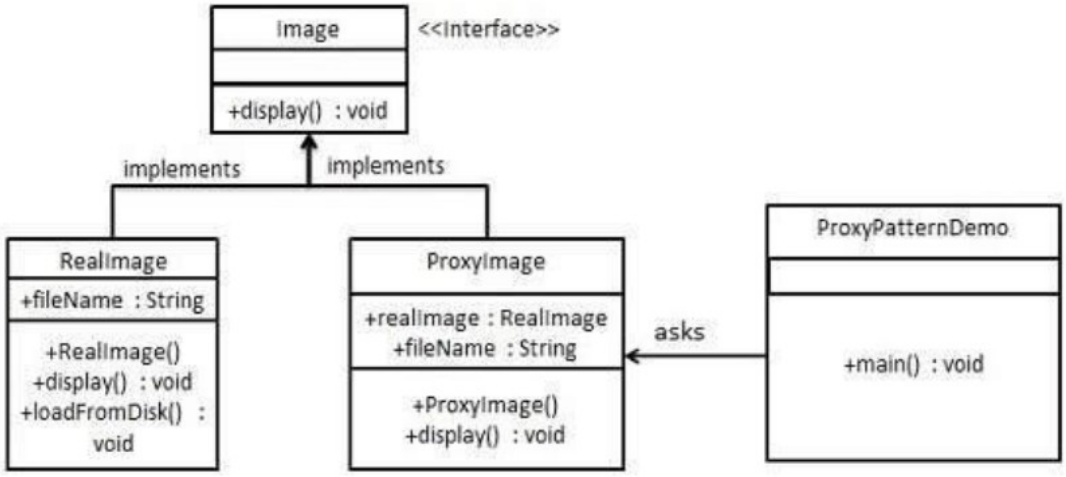
O Pattern Proxy é um padrão Estrutural definido pelo GoF (Gang of Four). O seu objetivo principal é encapsular um objeto através de um outro objeto que possui a mesma interface, de forma que o segundo objeto, conhecido como “Proxy”, controla o acesso ao primeiro, que é o objeto real.

As principais vantagens de se utilizar o pattern Proxy são:

- Permite deixar transparente o local (endereço) do objeto real. O cliente não precisa conhecer se o objeto é remoto ou não, este tipo de proxy é conhecido como Remote Proxy.
- O pattern Proxy é muito utilizado pela tecnologia J2EE, pelo objeto “javax.ejb.EJBObject”, que representa uma referência remota ao EJB. Para o cliente que está utilizando a interface remote de um EJB, é transparente a chamada remota ao servidor, permitindo que complexos sistemas distribuídos possam ser desenvolvidos como se fossem chamadas locais.
- O consagrado framework Hibernate também utiliza o pattern Proxy, por exemplo, ao fazer o “lazy-loading”, técnica utilizado para acessar o banco de dados apenas quando for necessário. Muitas vezes quando trabalhamos com o Hibernate, e uma busca é realizada, por exemplo usando o método “session.load(id)”, um Proxy para o objeto real é
- retornado. Neste caso o objeto ainda não está completamente preenchido, pois nenhum SQL foi realizado até este momento. Apenas quando uma propriedade deste objeto (métodos getX) ou um relacionamento, como por exemplo “empresa.getFuncionarios()” forem chamados, a consulta no banco será realizada. Tudo isto de forma transparente para o cliente.
- Útil para realizar otimizações, como cache de objetos. Também pode ser implementado rotinas de logs e controle de acesso (segurança). Este tipo de proxy é conhecido como Virtual Proxy.

Uma vez que neste padrão o objeto real é encapsulado, o objeto Proxy pode armazenar o resultado de um acesso em cache. Por exemplo, se um Remote Proxy é utilizado, o custo deste acesso remoto pode ser grande para a aplicação, neste caso o Proxy salva localmente os resultados em cache, diminuindo assim, a quantidade de acessos remotos.

Exemplo de Proxy:

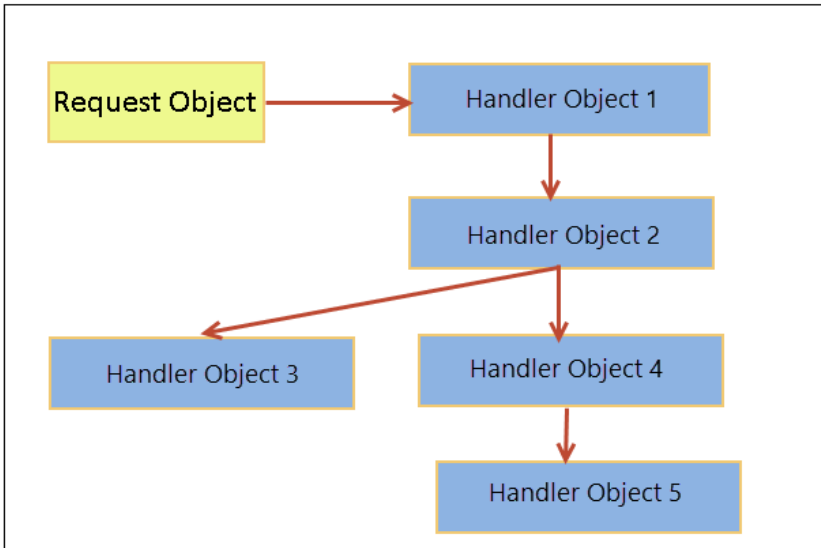


Padrões Comportamentais

Chain of Responsibility

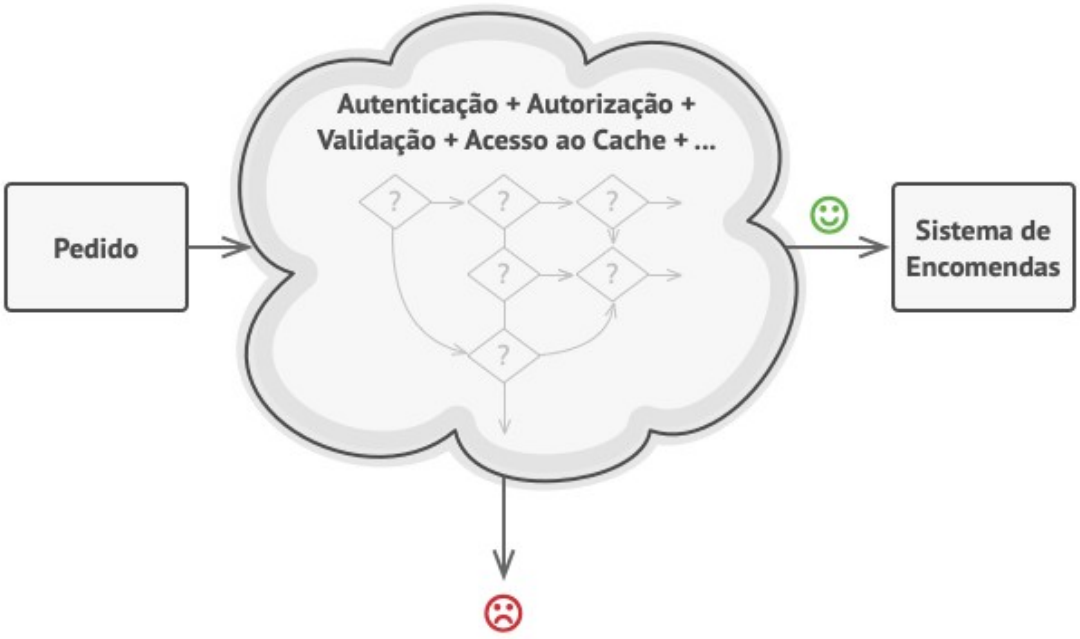
Este padrão é utilizado para obter um desacoplamento no design de software, onde um request (solicitação) do cliente é passado para um objeto (Handler object) que o processa . Se um objeto não conseguir lidar com uma solicitação, ele passa para o próximo objeto da cadeia. No final da cadeia, vamos ter um ou mais objetos implementando o comportamento para a solicitação.

Podemos relacionar este padrão como um Help Desk, onde o cliente entra em contato necessitando de uma ajuda técnica para algum produto e um atendente tenta resolvê-lo, se ele não consegue resolver ele irá mover para outro atendente e assim se repete até que o problema possa ser resolvido.

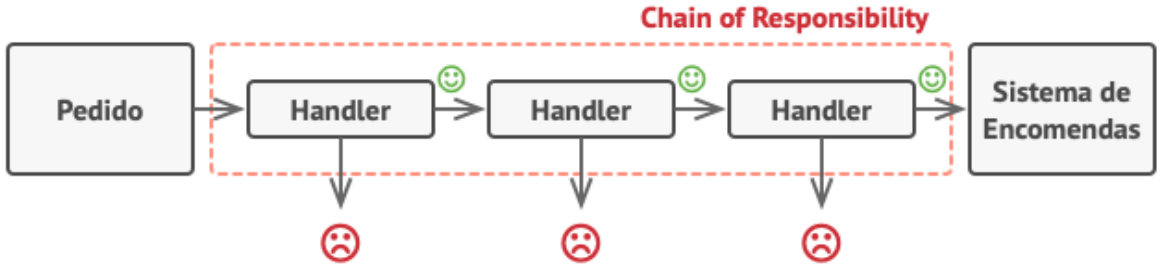


Exemplo:

Digamos que você trabalha em um sistema de encomendas e precisa restringir o acesso ao sistema para que somente usuários autenticados possam acessar.



Quanto maior o código para autenticação mais bagunçado fica. Então a solução pode ser utilizar o padrão Chain of Responsibility, onde cada checagem será movida para sua própria classe com um método. Assim um objeto (Handler) irá processar a solicitação e irá decidir se passa para o próximo até o usuário conseguir sua autenticação.



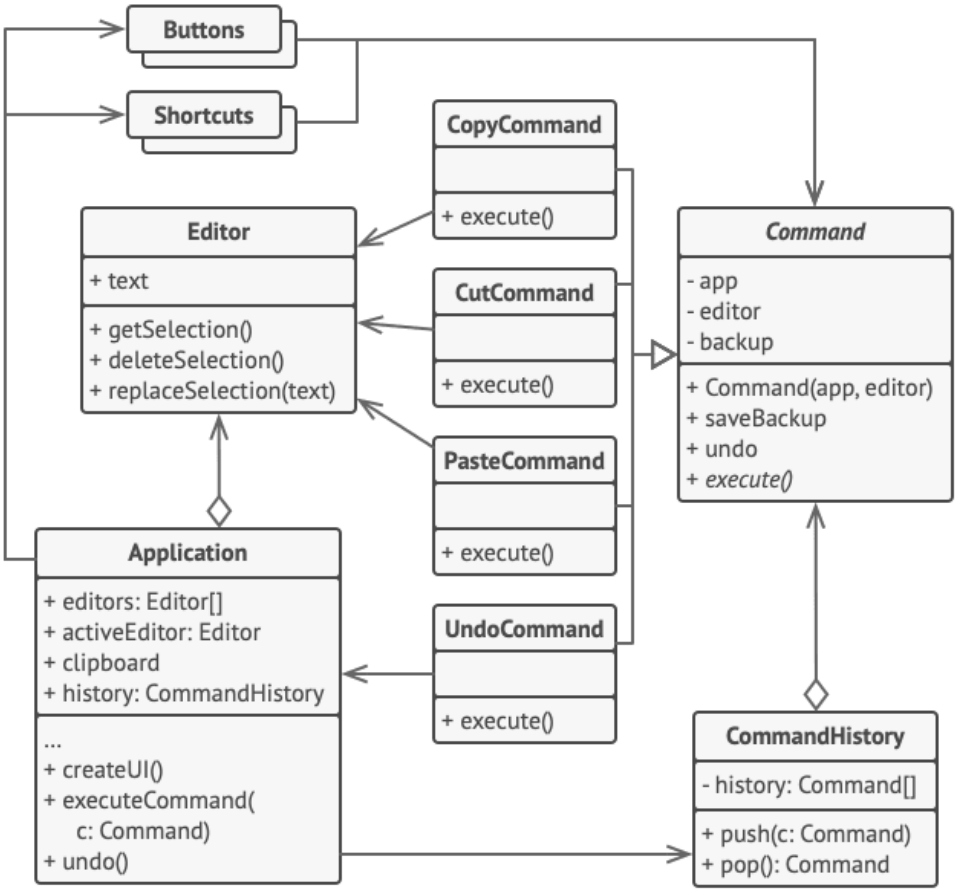
Handlers estão alinhados um a um, formando uma corrente.

Commad

É um padrão de design comportamental que transforma um request (Solicitação) em um objeto autônomo que contém todas as informações sobre a solicitação. Essa transformação permite que você passe solicitações como argumentos de método e atrasar ou enfileirar a execução de uma solicitação.

Exemplo

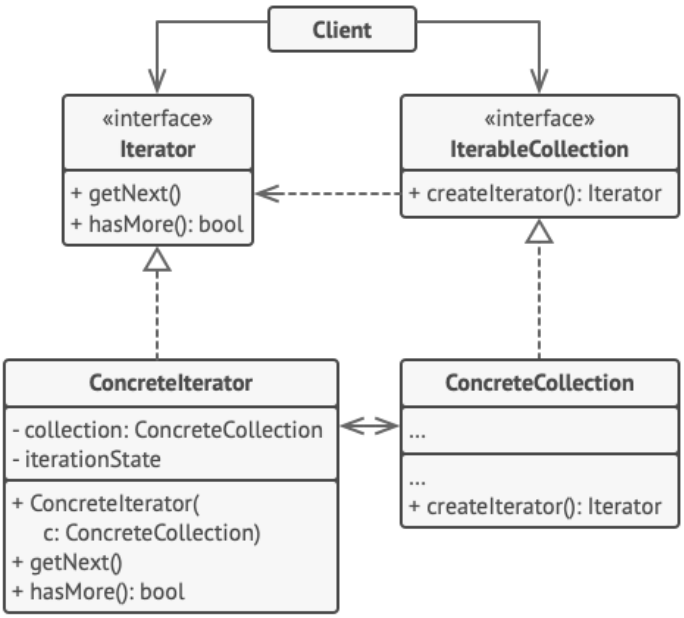
Neste exemplo, o padrão Command ajuda a rastrear o histórico de operações executadas e torna possível reverter uma operação, se necessário.



Os comandos que resultam na alteração do estado do editor (por exemplo, recortar e colar) fazem uma cópia de segurança do estado do editor antes de executar uma operação associada com o comando. Após a execução de um comando, este é colocado no histórico do comando (uma pilha de objetos de comando) juntamente com a cópia de segurança do estado do editor nesse momento. Mais tarde, se o usuário precisar reverter uma operação, a aplicação pode pegar no comando mais recente do histórico, ler o backup associado do estado do editor, e restaurá-lo.

Iterator

É um padrão de design comportamental que lhe permite atravessar elementos de uma coleção sem expor a sua representação subjacente (lista, pilha, árvore, etc.).



Iterator: Declara as operações necessárias para atravessar uma coleção: buscar o próximo elemento, recuperar a posição atual, reiniciar a iteração, etc.

ConcreteIterator: Implementa algoritmos específicos para percorrer uma coleção. O objeto iterator deve rastrear o progresso da travessia por si só. Isto permite que vários iteradores atravessem a mesma coleção independentemente um do outro.

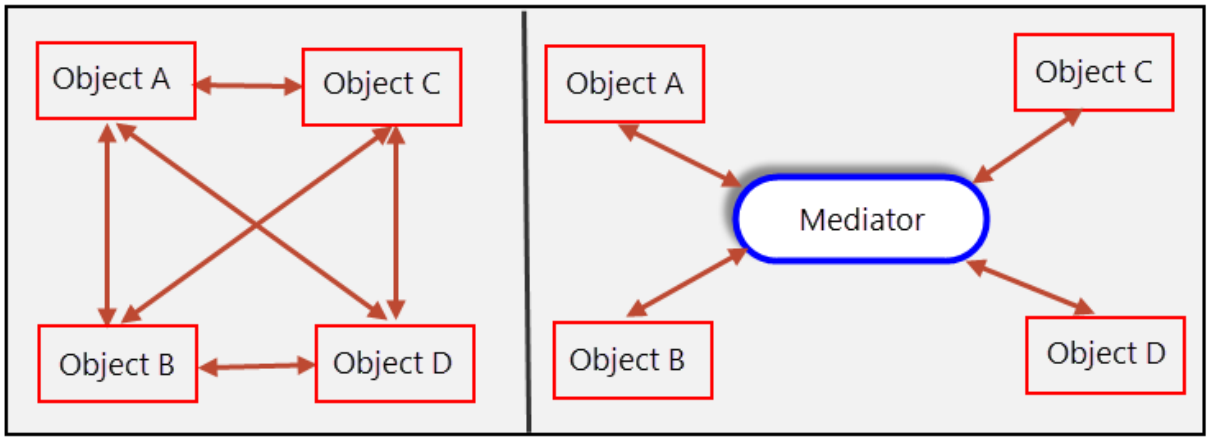
IterableCollection: A interface declara um ou vários métodos para tornar os iteradores compatíveis com a coleção. Note que o tipo de retorno dos métodos deve ser declarado como a interface do iterador para que as concrete collections possam retornar vários tipos de iteradores.

ConcreteCollections: Devolve novas instâncias de uma classe específica de ConcreteIterator cada vez que o cliente solicita por uma.

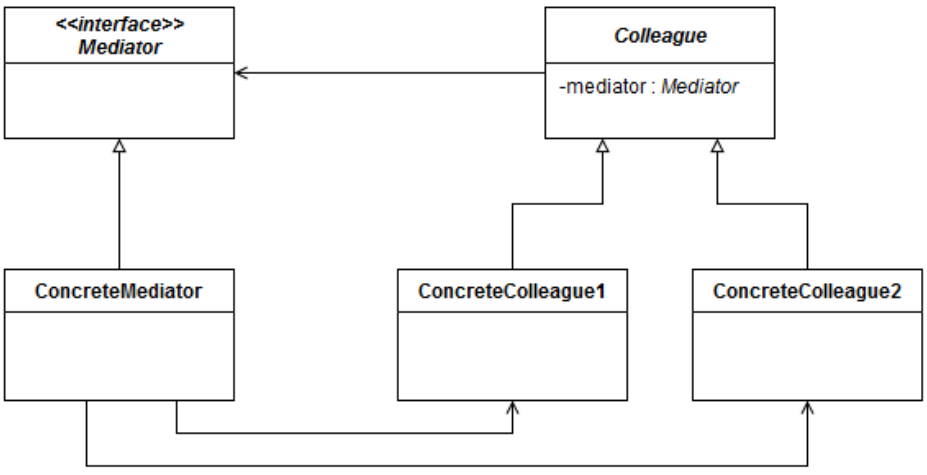
Client: Trabalha tanto com coleções como com iteradores através de suas interfaces. Desta forma, o cliente não é acoplado a classes concretas, permitindo utilizar várias coleções e iteradores com o mesmo código

Mediator

É um padrão de design comportamental que permite reduzir as dependências entre objetos. O padrão restringe a comunicação direta entre os objetos e os força a colaborar apenas através de um objeto mediador. O que o mediador realmente diz a esse conjunto de objetos é "fale comigo em vez de falar entre vocês mesmos".



Como mostrado na imagem acima, ele utiliza um objeto mediador para permitir que outros objetos da aplicação interajam sem conhecer a identidade um do outro. O mediador também encapsula um protocolo que os objetos podem seguir.



Neste diagrama podemos identificar os seguintes participantes:

A interface Mediator define a interface que os objetos Colleague utilizam para se comunicar. O Colleague define a classe abstrata com uma única referência ao Mediador. Já o ConcreteMediator encapsula a lógica de interação

entre os objetos Colleague. E por ultimo, ConcreteColleague1 e ConcreteColleague2 comunicam somente através do Mediator.

Memento

É um padrão de design comportamental que permite salvar e restaurar o estado anterior de um objeto sem revelar os detalhes de sua implementação.

Observer

O Observer é um padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

State

O State é um padrão de projeto comportamental que permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.

Strategy

O Strategy é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

Template Method

O Template Method é um padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse, mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura.

Visitor

O Visitor é um padrão de projeto comportamental que permite que você separe algoritmos dos objetos nos quais eles operam.