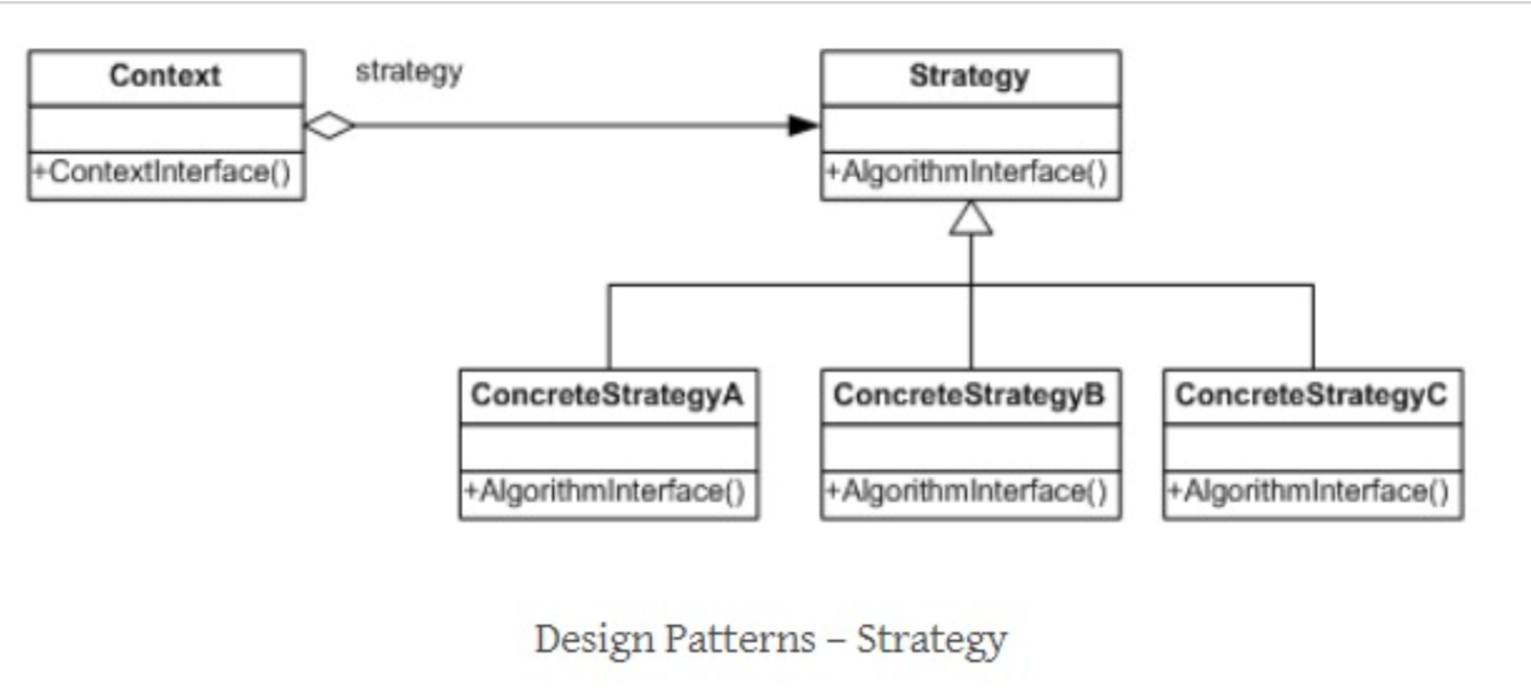
**Design Pattern**

**Padrão Strategy**

O padrão strategy tem com o objetivo de encapsular diferentes implementações de algoritmos, com isso o cliente chamaria a implementação específica de acordo com sua ESTRATEGIA.

basicamente você implementa em cima de uma interface ou de uma classe abstrata, pois vc não especifica que a implementação seria de uma classe específica... e com isso independente da classe especifica que for ser chamada pelo "cliente"... não vai precisar em alterar nada na implementação... somente novas classes especificas que herdam dessa interface para ser chamada no "cliente".

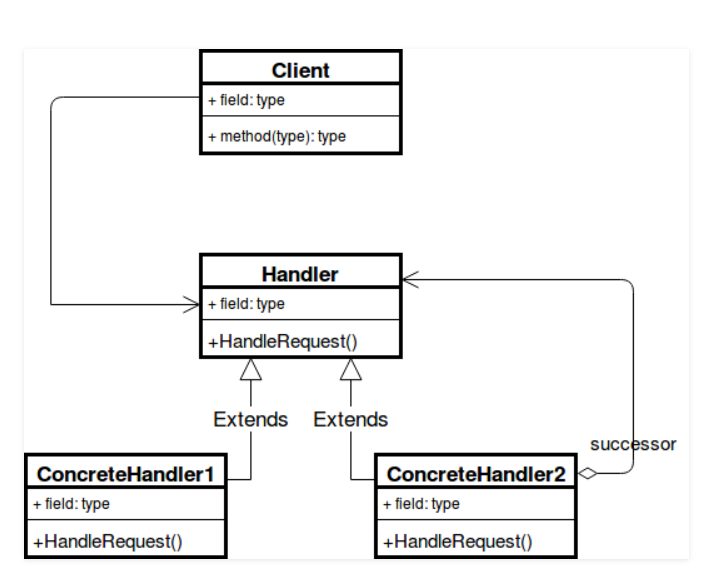


**Padrão Chain of Responsability**

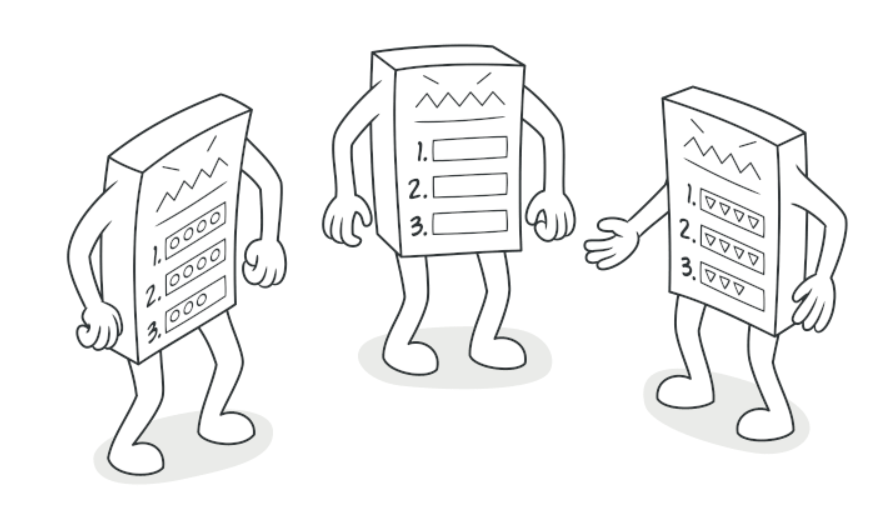
O padrão chain of responsability tem com o objetivo evitar algum acoplamento desnecessário, em que a solicitação do “cliente” será passada para uma cadeia de objetos, onde será executado até que a condição da execução seja satisfatória.



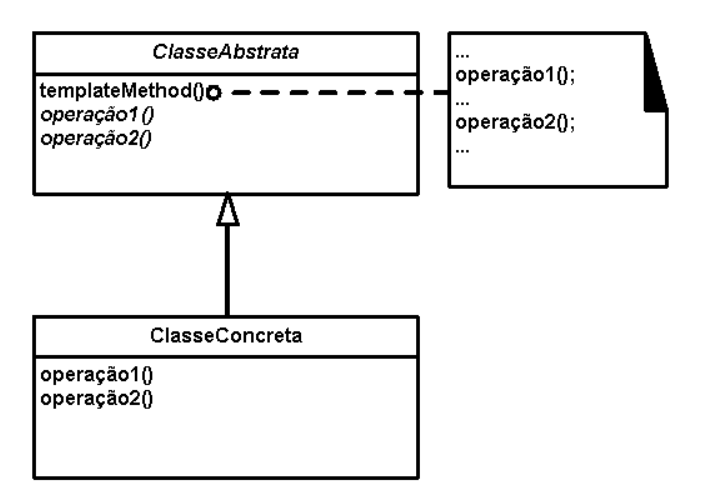
Por exemplo, imagine varias regras de negócios que somente uma delas será executada de acordo com algum range de validações, ou seja, criaríamos uma classe para cada regra de negócio em questão, assim a solicitação do “cliente” será passada no primeiro objeto nessa cadeia, e caso a validação não satisfazer será chamado o próximo objeto da cadeia, até que a validação seja executado corretamente.



**Padrão Template Method**  
O padrão template method tem com o objetivo definir o esqueleto de um algoritmo, deixando que as sub classes implemente algumas etapas.

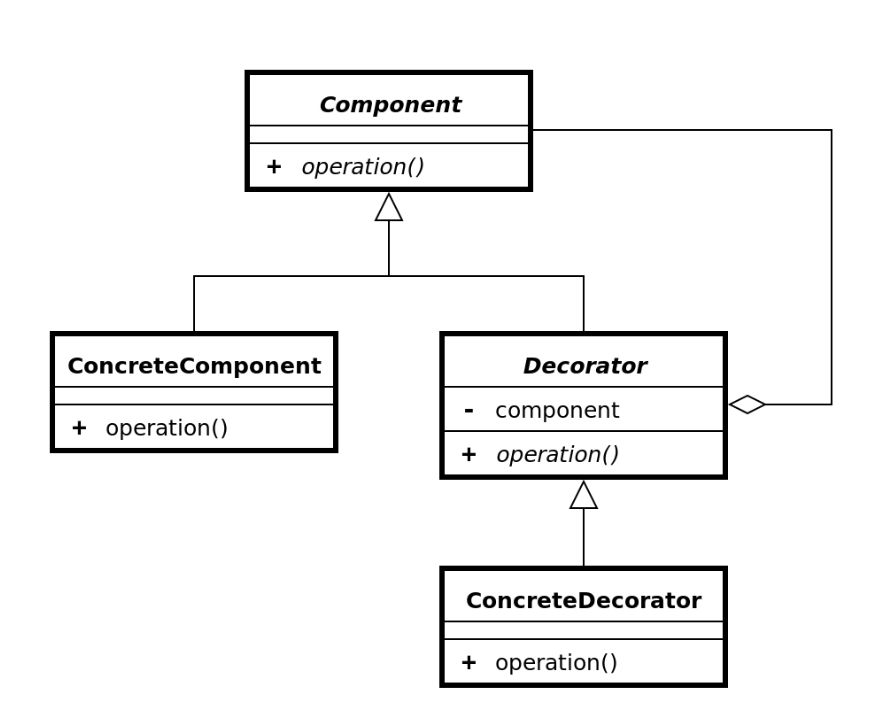


Por exemplo, imaginemos que duas ou mais classes tem uma implementação de algoritmo bem parecidas, que basicamente o que altera de uma classe para outra eh o modo de busca de uma informação, ou a verificação de uma condição que será executada, e que de restante a estrutura será a mesma. Nesse cenário criaremos uma classe abstrata, e nela criaremos um método com a implementação principal e de forma genérica que seria o que tem de comum entres as classes, e para cada ponto que seria especifico de cada sub classe, iremos expor outros métodos abstratos, para que ai sim, as sub classes implementem de forma especifica somente alguns pontos.



**Padrão Decorator**

O padrão decorator é muito utilizado quando queremos agregar funcionalidades para um objeto em tempo de execução. É visto como uma alternativa para a herança, pois é adicionado uma responsabilidade extra para um objeto e não para uma classe.



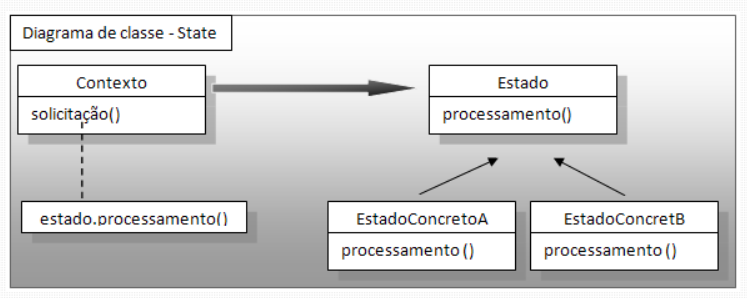
Se analisarmos o diagrama acima veremos que no geral teremos uma classe base abstrata “Decorator”, e é nele possui uma referência a ele mesmo. Dessa forma as classes concretas do decorator poderá ter essas novas “features” ou comportamentos, utilizando essa referência “duplicada” pelo construtor.

Dessa forma, não teremos mais o problema de ter várias combinações possíveis entre classes com vários códigos desnecessários.

**Padrão State**

O padrão state é muito utilizado quando queremos que um objeto altere seu comportamento quando o seu estado interno for alterado, que tanto o objeto parecera ter mudado de classe.

Isso acontece porque o padrão encapsula os estados em classes separadas e delega o comportamento do objeto para o estado atual.



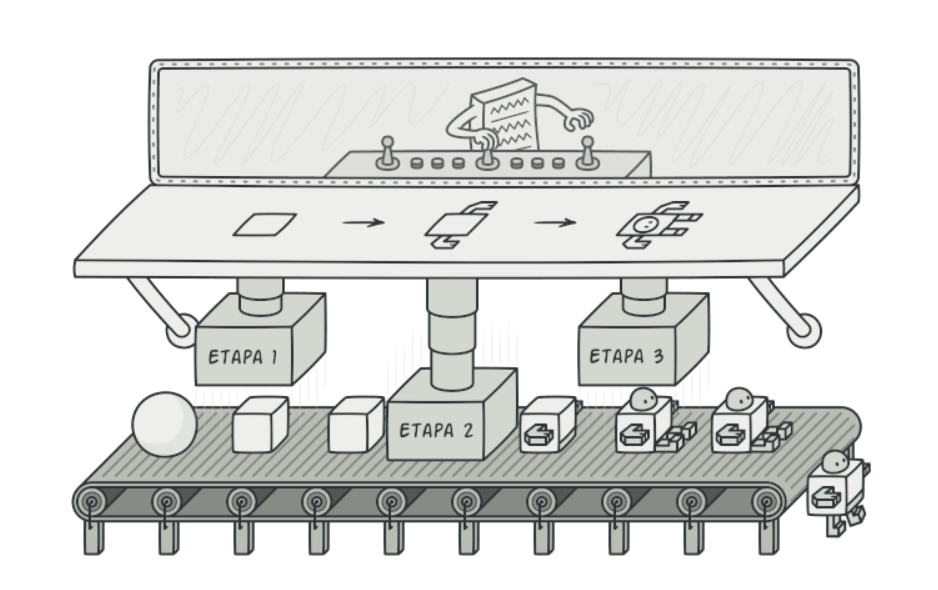
Como podemos ver no diagrama acima, a “interface” estado define a assinatura que todos os estados concretos devem ter, e será através dessa interface que o objeto terá a sua referência atribuído controlando o comportamento do estado inicial.

Vale destacar também, que podemos implementar no nosso padrão a troca de estados, onde permitimos ou não a troca gerando possíveis exception. Por exemplo em um objeto de orçamento que temos os estados em aprovação, aprovado, reprovado e finalizado, não podemos trocar o estado atual de em aprovação para finalizado direto.

**Padrão Builder**

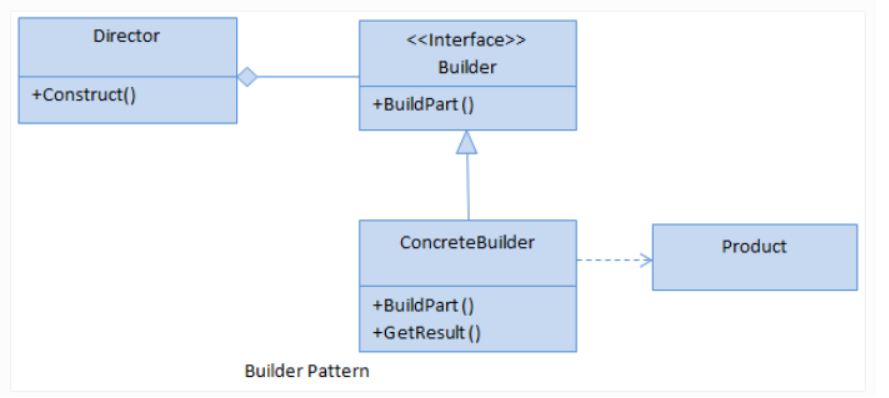
O padrão builder tem com o objetivo de nos ajudar a criar um objeto complexo. Pois ao invés de passarmos vários parâmetros em um construtor, podemos chamar uma classe que terá como responsabilidade apenas construir esse objeto complexo, passando uma serie de métodos inicializando cada propriedade do objeto, e no final chamamos um método chamado “build()” que será responsável por retornar esse objeto complexo construído.

Vale destacar, para passar uma serie de métodos inicializando cada propriedade do objeto, obviamente esses métodos devem retornar a própria referência do objeto, e esse mecanismo se chama “method chaining”.



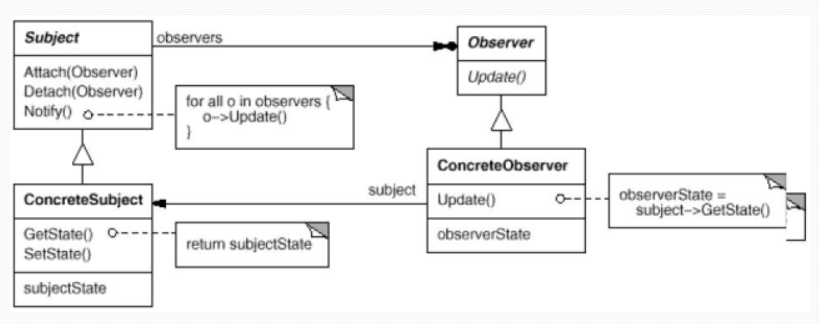
Vale destacar também que dependendo da complexidade do programa, se tivermos várias alternativas de como queremos construir nosso objeto complexo, podemos criar uma interface que se refere nas etapas que a nossa construtora irá criar o objeto, e nas classes concretas do construtor será implementado de forma específica, por exemplo ao construir uma casa, podemos construir cada casa com uma parede de madeira, outra casa com parede de vidro, etc.

Além disso, se for necessário, podemos criar a nossa classe “diretor” que é responsável por executar as etapas do construtor na ordem correta, funcionando de uma maneira parecida que um manual de instruções.



**Padrão Observe**

O padrão observe tem com o objetivo definir uma dependência entre objetos de um-para-muitos, do modo que quando muda o estado de um objeto, todos os seus dependentes serão notificados.



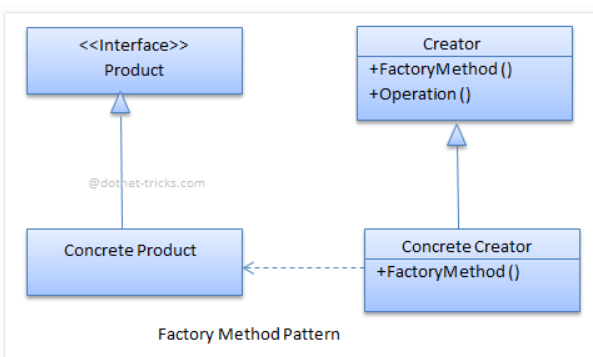
Como podemos ver no diagrama acima, teremos dois objetos núcleos principais, o “subject” e o “observer”.

* **Subject**: Objeto responsável por realizar as notificações para seus objetos dependentes.
* **Observer**: Objeto responsável por receber as notificações do “subject”, e realizar alguma ação específica.

Por exemplo, imagina que a cada nota fiscal de uma venda devera ser enviado automaticamente um e-mail, sms e gravar no banco suas informações. Logo esse padrão se encaixa muito bem, pois podemos criar uma classe “subject” onde as classes “observer(e-mail, sms, db)” podem se inscrever nela, logo toda vez que uma nota fiscal for gerada no subject, será executado cada ação de e-mail, sms, db, etc.

**Padrão Factory**

O padrão factory basicamente tem como objetivo de centralizar a criação de um objeto em um único lugar. Além disso podemos usar o mecanismo de interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo que será criado.



Veja no diagrama acima que a classe abstrata “Creator”, que apesar do nome, a sua principal função não é criar objetos, que tanto esse método “FactoryMethod()” será abstrato para que sim, suas classes concretas implementem a criação. Normalmente a classe abstrata criadora tem com o objetivo associar uma logica de negócio com a criação do objeto, que no diagrama acima está ilustrando com o método “Operation()”.

Basicamente esse padrão é bem semelhante ao padrão builder, com a diferença que o builder precisamos passar várias informações para a logica que monta o objeto, ou com construtores que podem ser enormes. Já o factory usamos quando queremos apenas isolar a construção de um objeto em um único lugar.

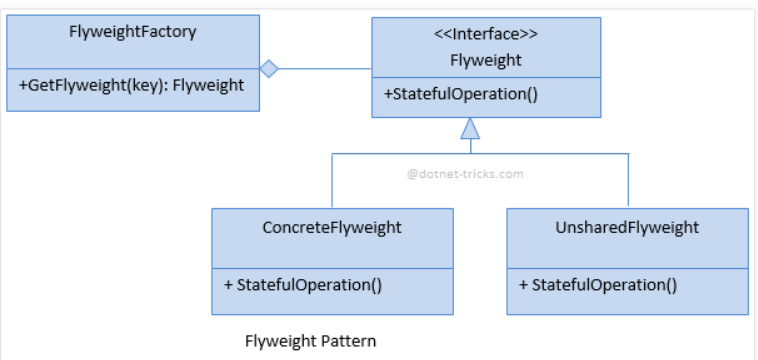
Vale destacar que podemos deixar a criação dos objetos em uma única fábrica, em várias fabricas, ou até mesmo criar um objeto para cada fábrica separada, pois vai depender muito como queremos implementar esse padrão. No entanto vale lembrar que o principal motivo do padrão factory é separar a lógica da criação de um objeto mais complexo em um único lugar.

**Padrão Flyweight**

O padrão flyweitht é um padrão estrutural, que tem como objetivo aproveitar uma única instancia do objeto em toda aplicação, assim temos um melhor uso da memória RAM do computador.

E para compartilhar esses objetos, cada objeto “flyweitht” é dividido em duas partes:

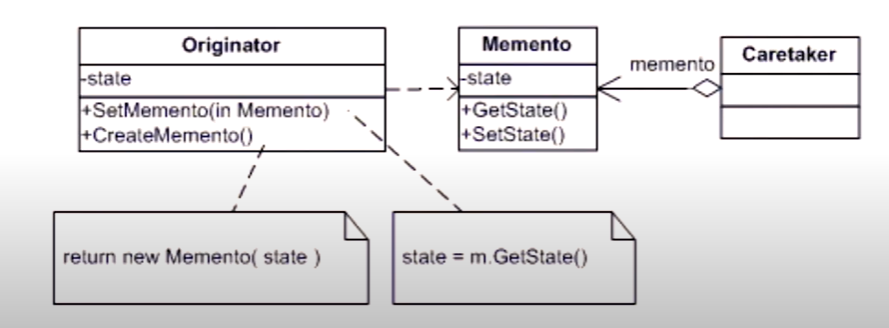
* **Depende do estado(extrínseca):** armazenado/calculado pelo objeto do “cliente” e passado para o “flyweitht” quando suas operações são invocadas.
* **Independente do estado(intrínseca):** armazenado e compartilhado no objeto “flyweitht”.



Esse padrão é parecido com o padrão factory, como que já vimos, com a diferença que o factory instancia uma classe importante/complexa de forma isolada e sempre ira instanciar aquele tipo de objeto definido pela fábrica, já o padrão flyweitht serve para quando temos várias instancias do mesmo tipo jogadas na aplicação e precisamos economizar e memoria, portanto será implementado uma fábrica que retornar apenas aquela instância do objeto que queremos, e caso não exista esse objeto já criado será feito a criação do objeto a adicionado na lista de objetos caso na próxima vez que possa ser retornado essa mesma instancia.

**Padrão Memento**

O padrão memento tem com o objetivo salvar e restaurar o estado anterior de um objeto, com isso o “cliente” pode realizar um rollback do estado atual para o estado anterior do seu objeto.



Analisando o diagrama acima, primeiro temos a classe “Originator” que é responsável por definir o estado atual e de gravar um novo estado, além de ter a opção de voltar para um estado anterior pelo “setMomento()”. A classe “Memento” basicamente encapsula nosso estado atual do objeto que estamos trabalhando ou definir um novo estado. Já a classe “Caretaker” geralmente é usada para listar os estados do seu objeto, ou seja, uma lista do tipo “Memento”, basicamente funciona como fosse um histórico dos estados do seu objeto.

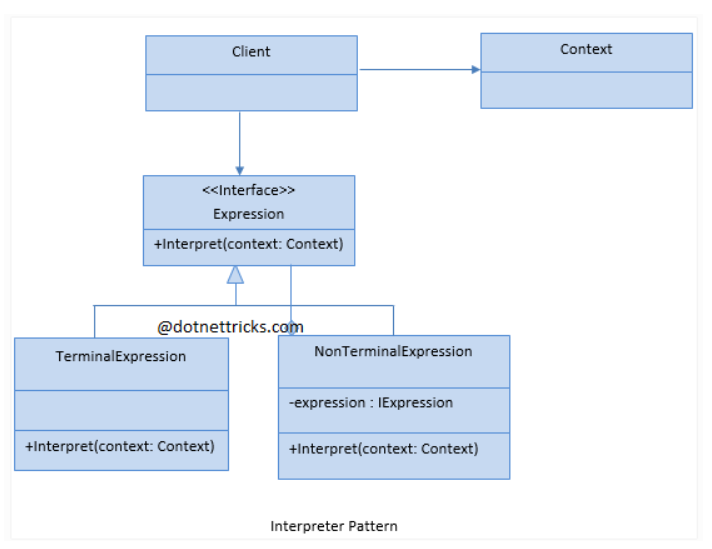
Vale destacar que esse padrão deve ser usado com cuidado, pois o lado ruim dele é conforme o uso teríamos várias instâncias sendo criadas por cada estado do objeto, embora podemos resolver isso com outros padrões, como o “flyweight” por exemplo.

E esse padrão é muito útil quando queremos desfazer algum estado do objeto ou mesmo comportamento quando trabalhamos com micro serviços, pois se der alguma exceção em alguma operação de um micro serviço, podemos gerenciar esse rollback do objeto e de repente atualizar o banco de dados na versão anterior que estava em outro micro serviço, entre outros exemplos.  
  
**Padrão Interpreter**

O padrão interpreter avalia as instruções escritas em uma gramática ou notações de idiomas. Esse padrão envolve uma implementação de uma interface de expressão que informa para interpretar um contexto específico. E esse padrão geralmente é usado nos compiladores.

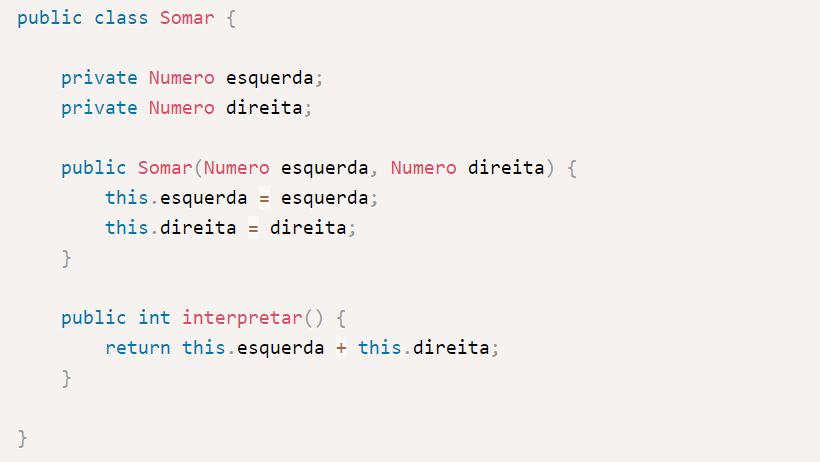
Exemplificando, o padrão interpreter discute definir uma linguagem de domínio, com uma gramática simples, representando regras de domínios como sentença de linguagem e interpretar essas sentenças para resolver o problema.

O padrão usa uma classe para representar cada regra gramatical, e como as gramáticas possuem uma hierarquia, portanto usando heranças de classes seria um mapeamento bem adequado para o padrão. Segue a estrutura do padrão:



Basicamente, esse padrão é responsável por criar interpretações de código, ou seja, passamos algo para ele, e o mesmo será responsável por processar e interpretar nosso parâmetro.

Por exemplo uma expressa matemática, poderíamos começar a criar uma classe que represente o “Numero”, e outra classe que represente a operação “Soma”, onde essa classe será responsável por interpretar expressões de adição.

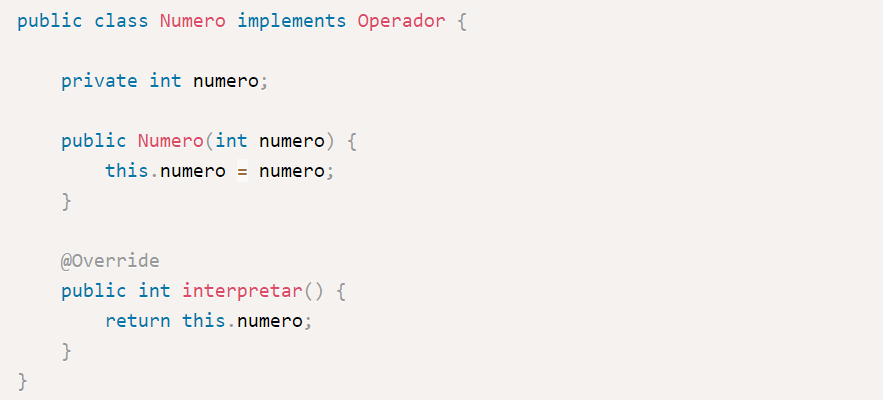


Porem como podemos ver esse código não vai funcionar, já que não podemos somar dois objetos “Numero” como definimos. Portanto devemos dizer que o objeto “Numero” deve se interpretar de alguma forma, devolvendo o seu valor para depois usarmos na soma.

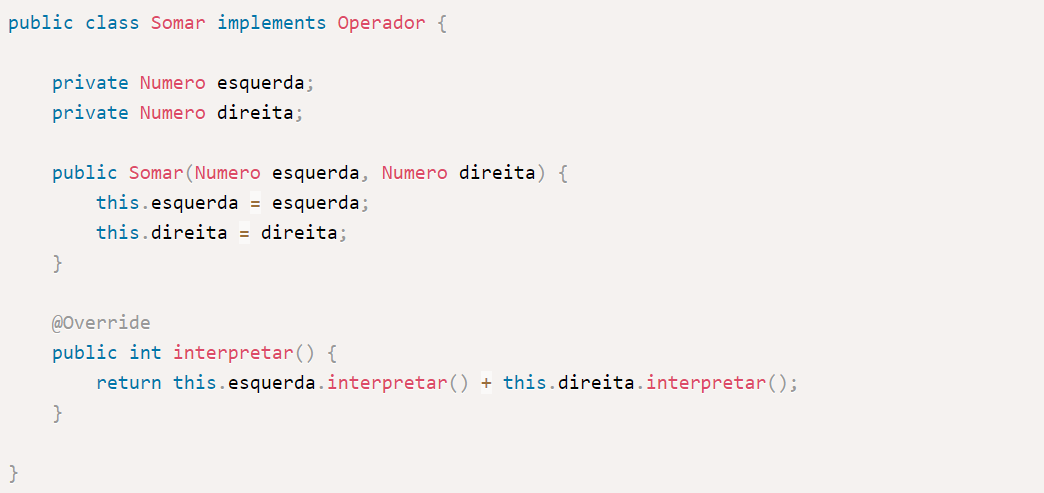
Logo veremos que temos um padrão em nossa linguagem de domínio e que cada classe deve se interpretar de alguma forma, como a classe “Numero” deve se interpretar e retornar um número, como a classe “Soma” que deve interpretar e realizar adição. Portanto iremos criar uma interface base para realizar essa interpretação.



Agora vamos assinar na classe “Numero” a nossa interface para poder se interpretar e retornar seu valor.

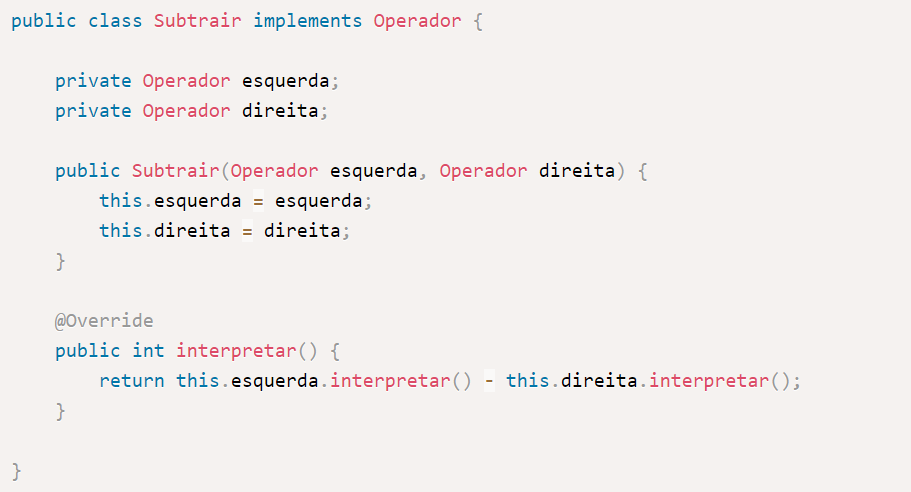


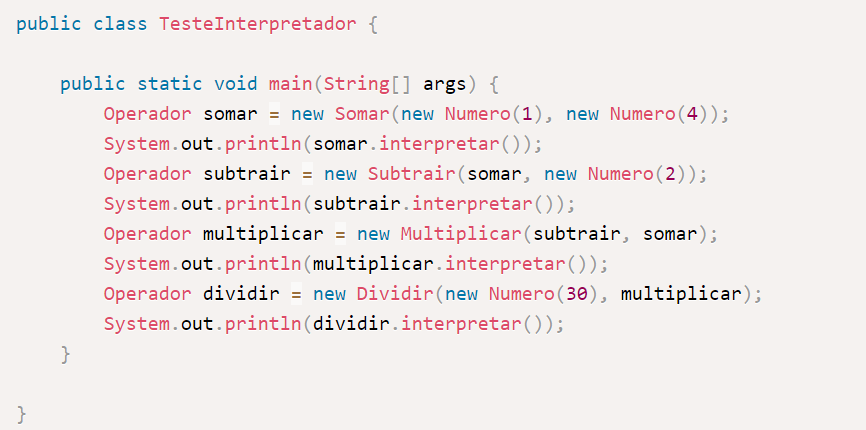
Enfim vamos ajustar a classe “Soma” e retornar à interpretação dos objetos “Numero”.



Agora está funcionando, mas se analisarmos uma expressão matemática, uma expressão pode ser composto por outras expressões, sendo assim precisamos o resultado dessa expressão como parâmetro.

Sabendo disso vamos alterar as chamadas do tipo “Numero” para a interface base “Operador”, assim podemos passar um operador de dividir, multiplicar e subtrair.





Enfim vemos que no final do código através do método “interpretar” será interpretado cada expressão de forma recursiva.

Com isso vemos que esse Padrão é usado normalmente quando temos um domínio em arvore onde todos os objetos precisam ser interpretados.

Finalizando, quando temos expressões que devem ser avaliadas, e transformamos em uma estrutura de dados, e depois fazemos com que a própria arvore se avalie, damos o nome do padrão de “interpreter”.

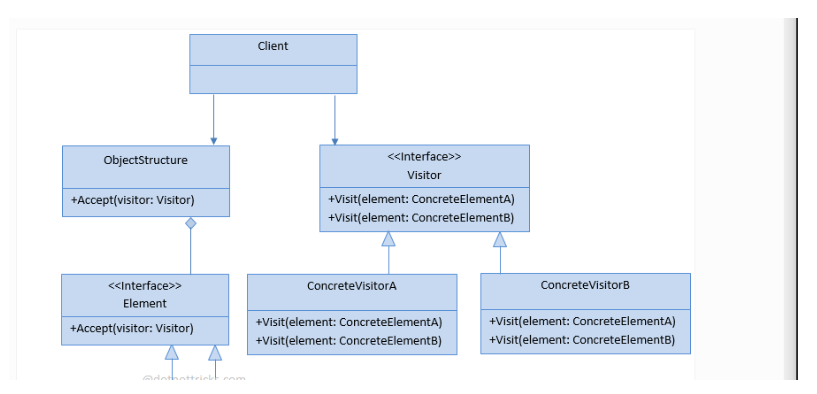
**Padrão Visitor**

O padrão visitor basicamente é usado quando precisamos navegar em uma estrutura de dados de arvore de maneira organizada e executar alguma lógica.

Por exemplo, como foi citado o código do padrão interpreter, podemos usar o padrão visitor para navegar em cada nó da expressão matemática, e consequentemente imprimir a expressão inteira processada.

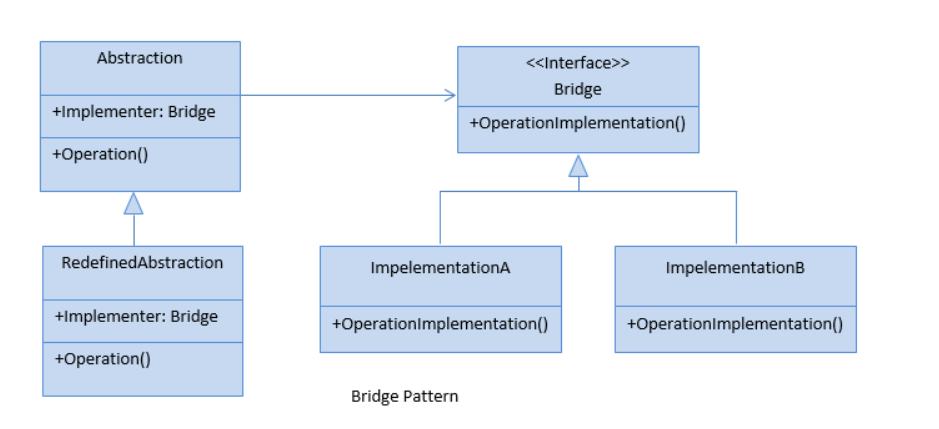
Vale destacar que de acordo com o cenário citado usando expressões, devemos permitir que cada classe referente ao nó da arvore, possibilita chamar o “visitor”, pois não sabemos qual expressão estamos imprimindo naquele momento pelo visitor, se é uma expressão que retorna apenas o número ou que retorna uma outra expressão(outra soma, subtração, etc.).

Segue a estrutura do padrão:



**Padrão Bridge**

O padrão bridge tem com o objetivo de separar a implementação com a abstração, semelhante ao padrão “strategy”, porem o padrão bridge trabalha com várias hierarquias diferentes, e por trabalhar com hierarquias deferentes será preciso realizar alguma “ponte”, entre essas hierarquias.

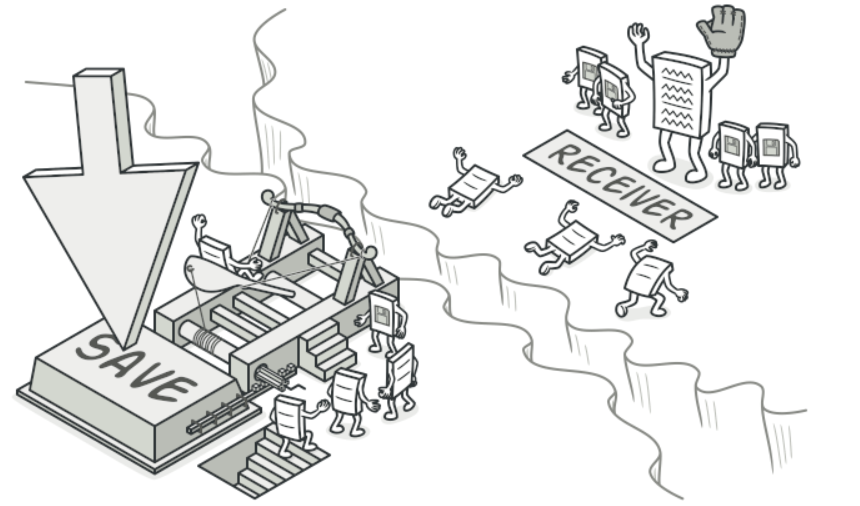


Por exemplo, imagine que temos um cenário onde precisamos enviar mensagens por sms ou por e-mail para um usuário, e que cada usuário pode ter um perfil de administrador ou de mensagem deve ser específica. Logo observamos que temos 2 hierarquias, hierarquia de mensagem e hierarquia do perfil do usuário.

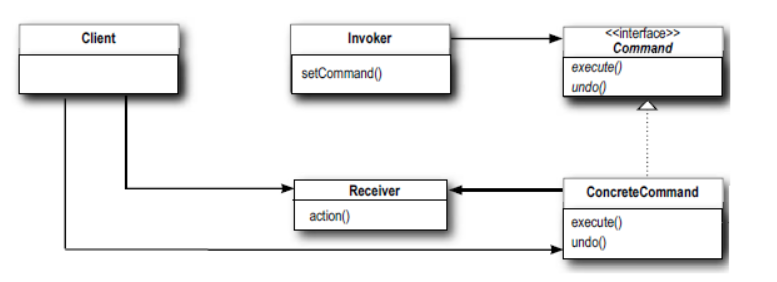
Sabendo disso teremos 2 interfaces(IMensagem e IEnviador), onde “IEnviador” será responsável por definir como será enviado a mensagem, e “IMensagem” será responsável por formatar a mensagem que será enviada. E para a mensagem de fator ser enviada, teremos uma propriedade nela referente ao “IEnviador”, representando uma “ponte”.

**Padrão Command**

O padrão command tem com o objetivo de encapsular em um objeto toda informação necessária para executar uma ação ou acionar um evento em um momento posterior.



Esse padrão é muito útil quando nossa aplicação receberá várias requisições para serem processadas, e caso não consigamos processar todas essas requisições naquele instante podemos colocar essas requisições em uma “fila” para ser executado todas em algum momento posteriormente de forma organizada, podemos se quiser ate cancelar algum comando na fila a ser executado podendo usar o padrão “memento(desfazer)” em conjunto.



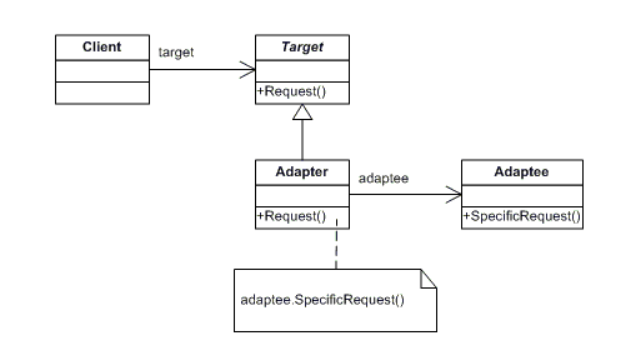
Basicamente transformamos um método que queremos executar em algum outro momento, em uma outra classe que implemente a interface do “Command”.

Vale destacar que a motivação desse padrão é que algumas vezes será necessário emitir solicitações para objetos sem saber sobre a operação que está sendo solicitada ou quem é o seu receptor.

**Padrão Adapter**

O padrão adapter tem com o objetivo em adaptar e converter uma interface de uma classe para uma outra interface que o “cliente” espera encontrar.

Muitas vezes uma classe que poderia ser reaproveitada e não é reutilizada justamente pelo fato de sua interface não corresponder a interface específica do domínio da sua aplicação. Ou seja, quando usamos uma biblioteca ou até mesmo um código legado e que não possuam a mesma “interface/estrutura” que a nossa aplicação necessita, usamos esse padrão para adaptar essas interfaces “diferentes” para usá-las dentro da nossa aplicação.

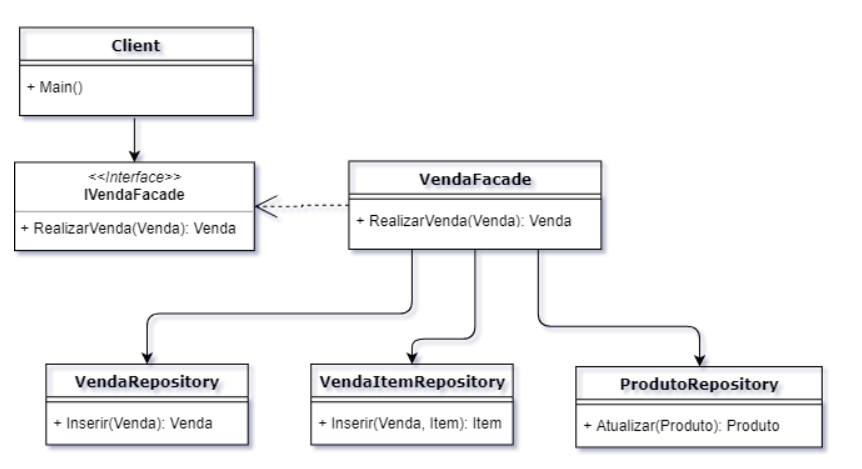


Por exemplo, podemos querer gravar informações de clientes no nosso banco de dados, e clientes estão em um arquivo xml, logo podemos usar esse padrão “Adapter” para adaptar a utilização desse xml, e converter em objetos de acordo com o domínio da nossa aplicação, que serão usados para gravar no nosso banco de dados.

**Padrão Facade**

O padrão facade tem com o objetivo de reduzir a complexidade de uma operação em um sistema, fornecendo um ponto de responsabilidade única para que o cliente consuma.

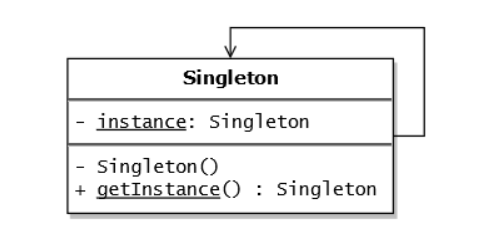
Esse padrão é muito utilizado quando nosso sistema utiliza outros subsistemas legados, onde teríamos que saber todas as suas classes e métodos para o cliente poder consumi-los ou não. E usando esse padrão, teríamos apenas um ponto(“interface”) utilizando todos esses métodos, assim teríamos que ter conhecimento apenas nessa classe de “fachada”.



Porem temos que ter muito cuidado ao utilizar esse padrão, pois com o tempo essa classe poderá ficar enorme e com dificuldades na manutenção.

**Padrão Singleton**

O padrão singleton tem com o objetivo de criar apenas uma instancia no objeto em toda a aplicação, bem semelhante ao padrão flyweight, porem o padrão flyweight trabalha com uma collection de objetos, já o padrão singleton trabalha com o objeto de forma individual.



Porem devemos ter muito cuidado com esse padrão, porem como ele terá uma única instancia para toda a aplicação, trabalhando como variáveis globais, ou seja, está centralizando parte de uma aplicação dificultando a modularização e aumentando o seu acoplamento. Além de dificultar implementações usando testes unitários, pois ele impede que sejam criados instancias para cada caso de teste, já que o estado do seu objeto é único e compartilhado, impossibilitando cenários de testes.