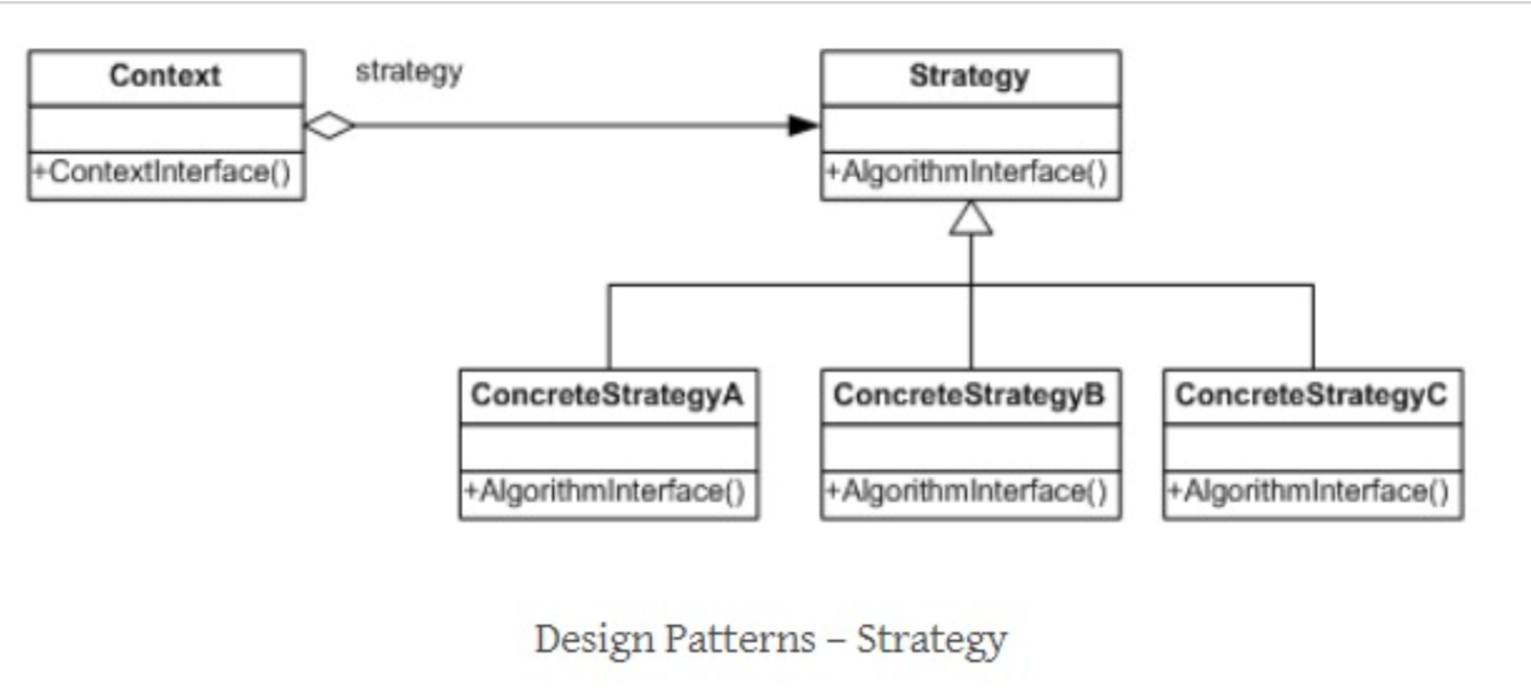
**Design Pattern**

**Padrao 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".

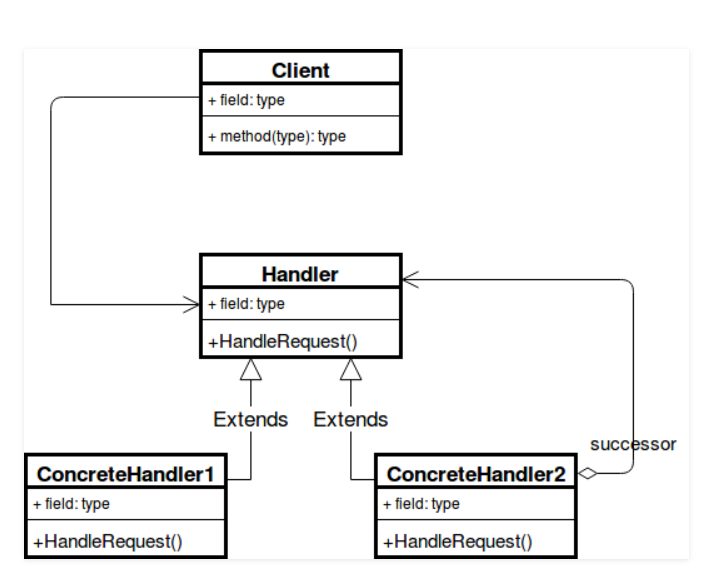


**Padrão Chain of Responsability**

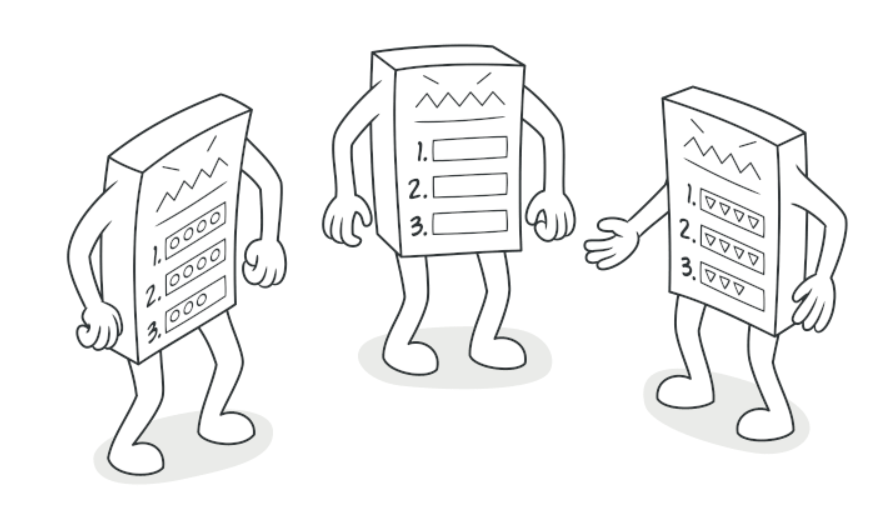
O padrão chain of responsability tem com o objetivo evitar algum acoplamento desnecessário, em que a solicitação do “cliente” será passada para uma cadeia de objetos, onde será executado até que a condição da execução seja satisfatória.



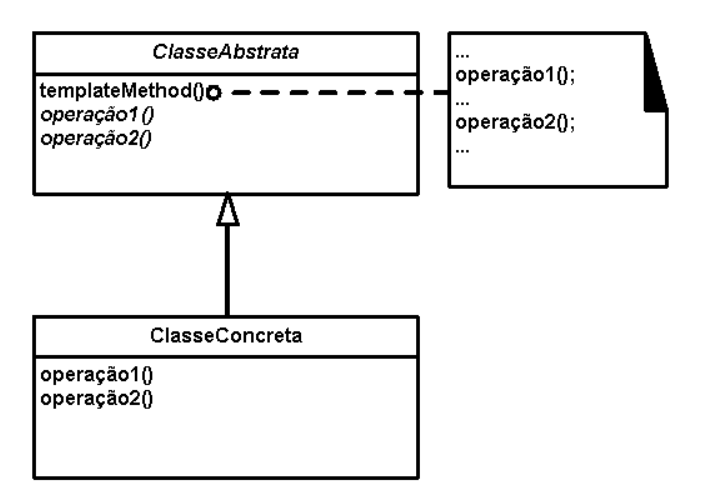
Por exemplo, imagine varias regras de negócios que somente uma delas será executada de acordo com algum range de validações, ou seja, criaríamos uma classe para cada regra de negócio em questão, assim a solicitação do “cliente” será passada no primeiro objeto nessa cadeia, e caso a validação não satisfazer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

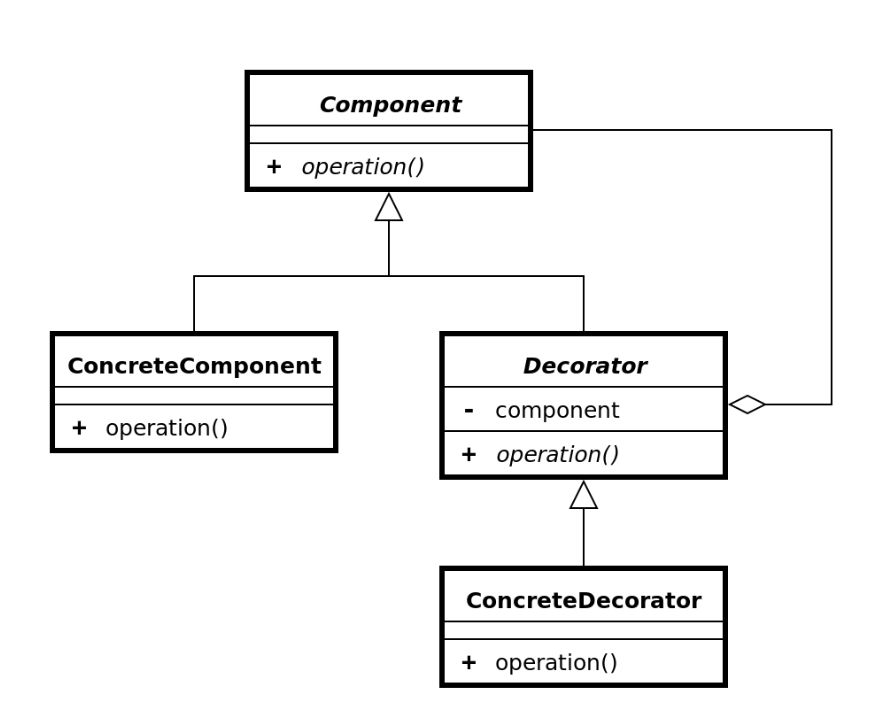


Por exemplo, imaginemos que duas ou mais classes tem uma implementação de algoritmo bem parecidas, que basicamente o que altera de uma classe para outra eh o modo de busca de uma informação, ou a verificação de uma condição que será executada, e que de restante a estrutura será a mesma. Nesse cenário 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.



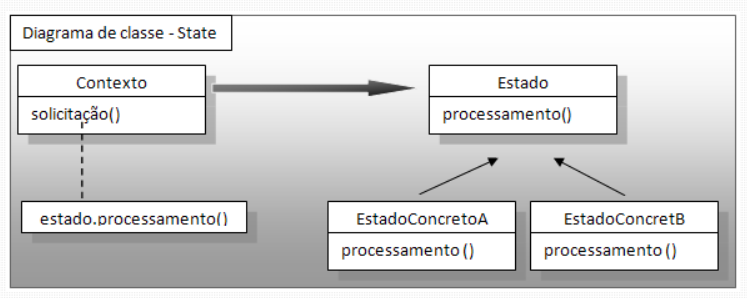
Se analisarmos o diagrama acima veremos que no geral teremos uma classe base abstrata “Decorator”, e é nele possui uma referência a ele mesmo. Dessa forma as classes concretas do decorator poderá ter essas novas “features” ou comportamentos, utilizando essa referência “duplicada” pelo construtor.

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.



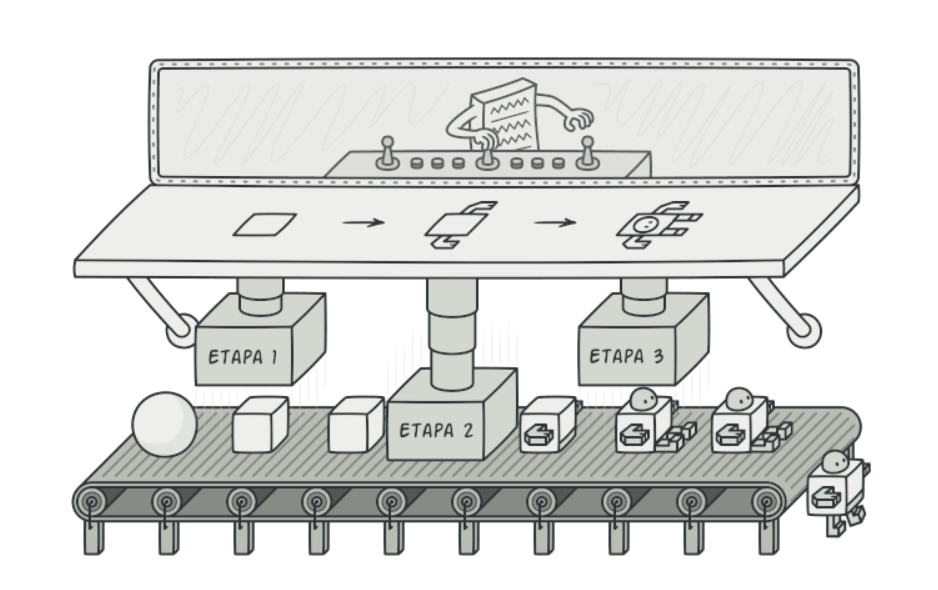
Como podemos ver no diagrama acima, a “interface” estado define a assinatura que todos os estados concretos devem ter, e será através dessa interface que o objeto terá a sua referência atribuído controlando o comportamento do estado inicial.

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**

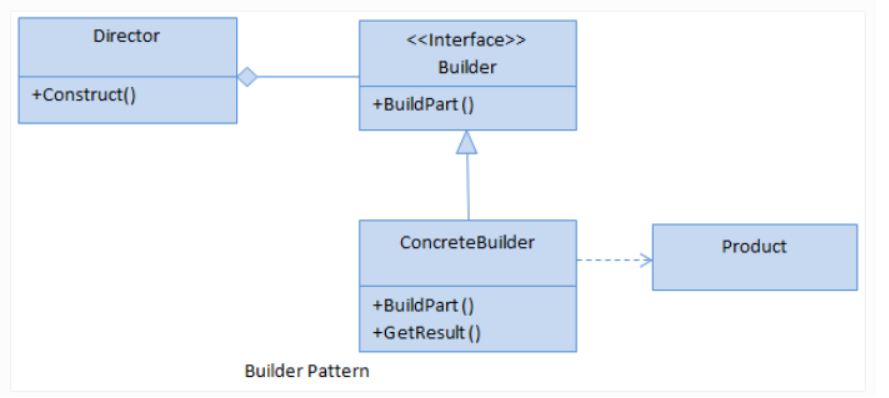
O padrão builder tem com o objetivo de nos ajudar a criar um objeto complexo. Pois ao invés de passarmos vários parâmetros em um construtor, podemos chamar uma classe que terá como responsabilidade apenas construir esse objeto complexo, passando uma serie de métodos inicializando cada propriedade do 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”.



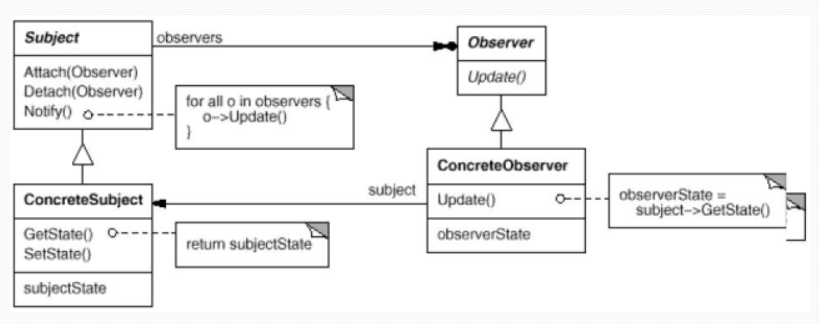
Vale destacar também que dependendo da complexidade do programa, se tivermos várias alternativas de como queremos construir nosso objeto complexo, podemos criar uma interface que se refere nas etapas que a nossa construtora irá criar o objeto, e nas classes concretas do construtor será implementado de 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.