Os Design Patterns fora desenvolvido na década de 90 pelos cientistas da computação Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm, com o objetivo de tornar os tornar componentes reutilizáveis que facilitam a padronização, que permita agilidade para as soluções de problemas recorrentes no desenvolvimento do sistema

**Existem dois padrões de projetos conhecidos** pela [engenharia de software](https://www.devmedia.com.br/engenharia-de-software/) que são: padrões **GoF**(Gang of Four) e os padrões **GRASP** (General Responsibility Assignment Software Patterns).

### **Padrões GRASP**

Ajudam a especificar, em que momento é atribuída a responsabilidade de decisão, quando se é um comportamento ou elemento ao objeto. A formação reúne nos seguintes padrões:

* Especialista na informação;
* Criador;
* Controlador;
* Acoplamento Fraco;
* Coesão Alta;
* Polimorfismo;
* Invenção Pura;
* Indireção;
* Variações Protegidas;

### **Padrões GoF**

Esses padrões tem como objetivo, solucionar problemas comuns de softwares que tenham algum envolvimento a [orientação a objetos](https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264). São formados por três grupos exibidos abaixo:

* **Padrões de criação**: Factory Method, Abstract Factory, Singleton, Builder, Prototype.
* **Padrões estruturais**: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy;
* **Padrões comportamentais**: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Method, Visitor.

### **Padrões GoF de criação**

Os padrões desse tipo, exigem um tratamento de como os objetos (classes) são criados, para atenderem as diversas necessidades. No [Java](https://www.devmedia.com.br/java/), os objetos são instanciados através de seus construtores, porém a utilização deles fica limitada quando:

* O código que referencia a criação de um objeto precisa conhecer os construtores dele, isso aumenta o acoplamento das classes.
* O objeto não pode ser criado parcialmente.
* O construtor não consegue controlar o número de instâncias presentes na aplicação.



**Figura 2**: Ilustração figurativa do Padrão GoF de criação.

**Factory Method** - Esse padrão define uma interface para a criação de um objeto, deixando que as subclasses fiquem responsáveis por decidir qual classe instanciar.

**Abstract Factory** - Permite elaborar uma interface para criação de famílias de objetos relacionados ou interdependentes, que não especifica suas classes concretas. A partir desse padrão consegue-se criar fábricas concretas, que são responsáveis pela criação de novos objetos para atender as necessidades do cliente. Portanto, essa prática ajuda a excluir a dependência entre o cliente e a classe dos objetos usados por ele.

**Singleton** - Usado quando desejado, que uma classe tenha apenas uma instância na aplicação. Abaixo, mostra alguns aspectos que devem serem cuidados ao criar esse padrão.

* O [construtor da classe](https://www.devmedia.com.br/construtores-em-java/28618) fica como privado (private), sendo que não pode ser instanciada para fora da própria classe.
* A [classe](https://www.devmedia.com.br/java-declaracao-e-utilizacao-de-classes/38374) é final, pois não permite a criação de subclasses da própria classe.
* O acesso é permitido [através do método](https://www.devmedia.com.br/metodos/7348) que retorna a instância única da classe, ou faz a criação de uma, caso não tenha sido criada.

**Builder** - Fornece uma interface genérica para a construção incremental de agregações. Esse padrão esconde os detalhes de como os componentes são criados, representados e compostos.

**Prototype** - Define os tipos de objetos a serem criados a partir de uma instância que funciona como um protótipo, fazendo com que novos objetos sejam criados com base nesse protótipo.

### **Padrões GoF Estruturais**

Esse padrões descrevem os seguintes aspectos: elaboração, associação e a organização entre objetos e classes/interfaces. Permitem combinar objetos em estruturas mais complexas, ou descrever como as classes são herdadas ou compostas a partir de outras.



**Figura 3**: Ilustração figurativa do Padrão GoF Estruturais.

**Adapter** - A ação desse padrão converte a interface de uma classe em outra, esperada pelo objeto cliente. Através dessa conversão, permite que classes com incompatibilidade de interfaces, consigam serem adaptadas para que outros objetos possam trabalhar juntos.

**Bridge** - Esse padrão separa uma abstração de sua implementação, permitindo que ambas possam variar independente, sendo estabelecida uma ponte (tradução de “bridge”) entre elas.

**Composite** - Compostos por objetos em árvores de agregação, fazendo que objetos agregados sejam tratados como um único objeto.

**Decorator** - Esse padrão busca em oferecer uma alternativa flexível para extensão de novas funcionalidades de objeto dinâmicos, sem o uso da herança. Um exemplo é o uso da Barra de Rolagem que pode ser incorporada de maneira dinâmica a uma janela, caixa de texto ou página web.

**Facade** - Oferece uma interface unificada para um conjunto de objetos que consistem um subsistema, definindo uma interface de alto nível que facilita no uso.

**Flyweight** - Utiliza o compartilhamento para dar suporte eficiente a um grande número de objetos com alto nível de granularidade. Esse padrão, cria modelos de referência para cada objeto, que concentra todas as características em comum em um objeto.

**Proxy** - Permite que o acesso seja controlado por meio de outro objeto, que atua como substituto. Geralmente, usado na programação orientada a aspectos, tendo como objetivo ajudar a separar, encapsular, modularizar métodos e organizar o código de acordo com a importância.

### **Padrões GoF Comportamentais**

Esses padrões, mostram o processo de como os objetos ou classes se comunicam. Em geral, buscam um baixo acoplamento entre os objetos, apesar da comunicação que existe entre eles.



**Figura 4**: Ilustração figurativa do Padrão GoF Comportamentais.

**Chain of Responsibility** - O padrão “Cadeia de Responsabilidades” encadeia vários objetos receptores que transporta uma solicitação, sendo aguardada até que um dos objetos responda. Com isso, evita-se o acoplamento entre o remetente de uma solicitação e seu destinatário, dando oportunidade para mais de um objeto tratar a solicitação

**Command** - Encapsula uma mensagem ou solicitação como um objeto, de modo que, se possa parametrizar clientes com diferente mensagens.

### **Interpreter**

São representações para gramáticas e abstrações para análise sintática, sendo usado mais para definição de linguagem.

### **Iterator**

Usado para prover um modo de acessar elementos de uma coleção de objetos de forma sequencial, sem exposição das suas estruturas internas.

### **Mediator**

Tem como objetivo, desacoplar e gerenciar as colaborações entre um grupo de objetos. Esse processo acontece no momento, quando é definido um objeto que encapsula a forma de como ocorrem as interações. Portanto, tem um baixo acoplamento entre os objetos, evitando que eles se refiram uns aos outros explicitamente.

**Memento** - Funciona como um “snapshot”, pois captura e externaliza o estado interno de um objeto, sem violar o encapsulamento, permitindo que o objeto consiga ser restaurado a esse estado futuramente. Um exemplo disso, são em operações que precisam ser anuladas, para que o estado anterior do objeto seja depois restaurado.

**Observer** - Usado para sincronizar, coordenar ou manter a consistência entre objetos relacionados. Sendo que tem a dependência um para muitos entre objetos, ou seja, quando o estado de um objeto muda, todos os que dependem são notificados e atualizados automaticamente.

**State** - Permite que um objeto mude seu comportamento quando seu estado interno é alterado, sendo mudado a classe do objeto.

**Strategy** - Permite definir novas conjunto de algoritmos sem alterar as classes dos elementos sobre os quais opera.

**Template Method** - Adequada o esqueleto de um algoritmo em uma operação, permitindo que subclasses componham o algoritmo e tenham a possibilidade de redefinir certos passos a serem tomados no processo, sem mudá-lo.

**Visitor** - Representa uma operação a ser realizada sobre elementos da estrutura de um objeto, sendo permitida a criação de uma nova operação sem que mude a classe dos elementos sobre as quais é operado.

Fontes:

FREEMAN, ERIC & FREEMAN, ELISABETH – Use a Cabeça! Padrões de Projetos (Design Patterns), 2ª Edição

Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm

Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos