

# PADRÕES DE PROJETO – GANG OF FOUR (GOF)

## Conceitos Básicos

INCIDÊNCIA EM PROVA: BAIXA

Pessoal, vamos iniciar pelo básico! *O que é um Padrão?* É um modelo testado ou documentado para se alcançar um determinado objetivo. O autor Christopher Alexander afirma que cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente e o núcleo de sua solução, de tal forma que ela pode ser usada diversas vezes sem que seja realizada da mesma maneira. *E o que seriam Padrões de Projeto (do inglês, Design Patterns)?*

Respondo: são descrições de objetos que se comunicam e classes que são customizadas para resolver um problema genérico em um contexto específico. Esses padrões nomeiam, abstraem e identificam aspectos comuns em uma estrutura e os torna úteis para que sejam realizados. *Como é isso, professor?* Suponham que haja uma fábrica de software que desenvolve diversos sistemas para as mais variadas áreas de negócio.

Sabe-se que, mesmo em áreas aparentemente distintas ou contrastantes, grande parte dos problemas já possuem uma solução conhecida, formalmente documentada e exaustivamente testada, uma vez que se tratam de problemas recorrentes. Portanto, não há necessidade de se resolver o problema do início se já existe uma solução adequada. Assim sendo, os padrões de projeto são modelos para solucionar problemas gerais de projeto em um contexto particular.

*Poxa, professor... o engenheiro de software que inventou isso era genial, né?!* Qual é, pessoal! Mais uma vez, nós “roubamos” isso de outras engenharias. Em meados de 1977, um sujeito chamado Christopher Alexander (arquiteto, matemático e urbanista austríaco) estudava como melhorar o projeto de edifícios. **Ao se cansar de observar repetidos problemas, ele lançou um manifesto com o registro de diversos padrões de projeto recorrentes na engenharia civil.**

Esse manifesto continha algumas regras e figuras que descreviam métodos para a construção de projetos práticos, seguros e atrativos em quaisquer escalas, desde regiões inteiras a cidades, vizinhanças, jardins, edifícios, salas, entre outros. **Cada padrão foi testado no mundo real, sendo revisado por diversos arquitetos e engenheiros.** Alexander fez uma importante declaração a respeito de padrões de projeto de arquitetura:

*“Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente e, então, descreve o núcleo da solução para aquele problema, de tal maneira que pode-se usar essa solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes”.*

Bem, isso chamou a atenção de programadores que estavam de saco cheio de ter que refazer várias partes do código para cada projeto. **Foi então que, em 1994, quatro engenheiros de software, conhecidos como The Gang of Four (GoF), resolveram compilar um conjunto de bibliotecas de**



**soluções para problemas comuns de codificação e lançaram um livro clássico com 23 Padrões de Projeto de Software.**

Esse livro se tornou um clássico *best-seller* da literatura orientada a objetos e é recomendado na nossa bibliografia, não obstante ser de leitura extremamente técnica. **Concernente aos padrões de projeto, os autores desse livro dizem:** “*São descrições de objetos que se comunicam e classes que são customizadas para resolver um problema genérico de design em um contexto específico*”. Professor, que contexto específico é esse? Bem, o contexto é inerente a área de negócio!

**Cada um adapta o padrão genérico ao seu ambiente específico para resolver um mesmo problema.** Os padrões de projeto abstraem e identificam os aspectos-chave de uma estrutura de projeto comum para torná-la reutilizável. Entre as vantagens dos padrões de projeto, podemos mencionar que é possível aprender com a experiência de outros, identificando problemas comuns de engenharia de software e utilizando soluções testadas e bem documentadas.

É possível utilizar soluções com nomes que facilitam a comunicação, compreensão e documentação. Eles facilitam o reúso de soluções arquiteturais bem-sucedidas e permitem desenvolver softwares de melhor qualidade. Padrões capturam a estrutura estática e a colaboração dinâmica entre objetos participantes no projeto de sistemas. Eles utilizam conceitos de orientação a objetos para construir código reutilizável, eficiente, de **alta coesão e baixo acoplamento**.

*Então há apenas benefícios em se utilizar padrões de projetos?* Claro que não! Muitas vezes, as tentativas de se escrever um código que esteja em conformidade com um determinado padrão de projeto aumentam desnecessariamente a complexidade do código. **Em outras palavras, a tentativa de simplificar pode acabar complicando a implementação.** Ademais, estudos mostram que grande parte dos padrões podem ser simplificados ou eliminados. *Como, professor?* utilizando-

Nome
Problema
Solução
Consequências

Se recursos diretos das próprias linguagens de programação, como em LISP e DYLAN. Portanto, **os Design Patterns podem ser simplesmente um indicativo de que algumas linguagens de programação falham em oferecer determinadas características. Cada padrão de projeto é organizado segundo quatro elementos: Nome, Problema, Solução e Consequências.** Nome simplesmente identifica o padrão; Problema descreve a condição em que ele deve ser aplicado; Solução descreve como resolvê-lo; e Consequência descreve os impactos.

**Por fim, algumas observações importantes!** *Esses 23 Design Patterns somente podem ser utilizados em projetos de tecnologia da informação?* Claro, eles são padrões de projeto de software. *Esses 23 Design Patterns somente podem ser utilizados em projetos orientados a objetos?* Sim, também. Pessoal, Padrões GOF somente com Orientação a Objetos! Padrões de Projeto, em geral, podem usar qualquer paradigma ou linguagem!



		Propósito		
Escopo	Objeto	Criação	Estrutura	Comportamento
		Builder Abstract Factory Prototype Singleton Factory Method	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor Interpreter Template Method

Os 23 Padrões de Projeto podem ser classificados quanto aos seus propósitos em três tipos: Padrões Criacionais, Padrões Estruturais e Padrões Comportamentais. **Infelizmente, vocês têm que decorar como se classificam todos os padrões de projeto.** A tabela acima apresenta todos os padrões de acordo com seus respectivos propósitos. Os **Padrões Criacionais** abstraem o processo de criação de objetos a partir da instanciação de classes.

Já os **Padrões Estruturais** tratam da forma como classes e objetos estão organizados para formar estruturas maiores. Por fim, **Padrões Comportamentais** se preocupam com os algoritmos e responsabilidades dos objetos, que ocorrem em tempo de execução. Pessoal... quando eu estudei esse assunto na minha vida de concurseiro, utilizei de um mnemônico maneiríssimo criado pelo Prof. Rogério Araújo para memorizar:



Explicação: A fábrica (*Factory Method*) abstrata (*Abstract Factory*) constrói (*Builder*) um protótipo (*Prototype*) único (*Singleton*). A ponte (*Bridge*) adaptada (*Adapter*) é composta (*Composite*) de decorações (*Decorator*) na fachada (*Facade*) para o peso-mosca (*Flyweight*) se aproximar (*Proxy*). E não tem frase para o último? **Não, porque não é necessária! Se não é um padrão criacional ou estrutural, é um padrão comportamental.**



## Padrões Criacionais

INCIDÊNCIA EM PROVA: ALTÍSSIMA

### Abstract Factory

*Esse padrão fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.*

Esse padrão de projeto deve ser utilizado quando o sistema for configurado como uma família de produtos, que – uma vez relacionados – são projetados para serem utilizados em conjunto. **O Abstract Factory busca assegurar essa restrição, revelando apenas suas interfaces e, não, suas implementações.** Considerem a hipótese de se visualizar a *globo.com* de um *desktop* ou de um *smartphone*. Evidentemente, são duas interfaces gráficas diferentes (Clássica e Mobile).

Apesar de serem distintas, existem muitas similaridades. Logo, é útil aproveitar todo código em comum e implementar com as classes concretas apenas as diferenças. **Assim, o usuário interage apenas com a interface provida pelo padrão Abstract Factory.** Em tempo de execução, ela descobre, a partir de algum parâmetro fornecido, se o acesso parte de um *smartphone* ou *desktop* e instancia apenas a classe concreta específica que renderizará a interface gráfica pretendida.

**Galera, todos os padrões do tipo Factory encapsulam a criação de objetos.** O padrão Factory Method, por exemplo, deixa as subclasses decidirem quais objetos criar.

### Builder

*Esse padrão separa a construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes tipos de representações.*

Esse padrão de projeto deve ser utilizado quando o algoritmo para criação de um objeto complexo for independente das partes que compõem o objeto e independente de como ele é montado. **Ademais, o processo de construção deve permitir diferentes representações para o objeto que será construído.** Esse padrão é bastante parecido com o Abstract Factory. A diferença é que não se constrói uma família de objetos de uma única vez, mas partes do objeto passo-a-passo!

No exemplo anterior, havia duas famílias de objetos compostas de várias partes (botões, barra de rolagem, caixas de seleção, ícones, etc), que compunham a interface de um website em um *smartphone* ou *desktop*. Ao se descobrir de que dispositivo se tratava, instanciava-se a interface completa do *smartphone* ou *desktop*. **Agora suponham que um tablet seja capaz de reunir simultaneamente elementos de ambas as interfaces gráficas.**

O Builder é capaz de coordenar a construção da interface gráfica do tablet ao instanciar partes dessas interfaces. Logo, ele pode criar diferentes representações de suas interfaces gráficas ao



instanciar, por exemplo, um botão da interface do *smartphone*, a barra de rolagem da interface do *desktop*, caixas de seleção da interface do *smartphone* e ícones da interface do *desktop*. **Ele possui uma classe que coordena a construção desses objetos até a criação da representação final.**

## Factory Method

*Esse padrão define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar.*

**Esse padrão de projeto deve ser utilizado quando uma classe não puder antecipar qual objeto ela deve criar.** Deve ser usado, também, quando uma classe quer que suas subclasses especifiquem os objetos que ela criar. Por fim, deve ser usado para delegar responsabilidades a diversas outras subclasses. Considerem a hipótese de se visualizar as características de um carro de uma concessionária.

No entanto, sabe-se que em uma concessionária há diversos modelos de carro. Então, pode-se criar uma classe abstrata base para representar todos os carros e especializá-la em subclasses que representem cada tipo de carro (Corsa, Celta, Cruze, Camaro, etc). **No entanto, o problema surge quando se quer criar um objeto, uma vez que – de alguma forma – precisa-se identificar qual objeto se deseja criar.**

**O Factory Method cria objetos concretos que implementam a classe base e especializam cada objeto, sendo definidos, pelo usuário, em tempo de execução, qual carro deverá ser instanciado.** É bom destacar que ele – conforme especificado – contém diversos elementos, tais como: Creator, ConcreteCreator, Product e ConcreteProduct.

## Prototype

*Esse padrão especifica os tipos de objetos para criar usando uma instância como protótipo e cria novos objetos copiando este protótipo.*

**Esse padrão de projeto deve ser utilizado quando um sistema possuir componentes cujo estado inicial tenha poucas variações.** É oportuno disponibilizar um conjunto pré-estabelecido de protótipos que dão origem aos objetos que compõem o sistema. Dessa forma, basta modificar os atributos que forem diferentes e adaptar o objeto para o uso pretendido. Considerem a hipótese de se preencher os dados de todas as pessoas de uma família.

Considerem também que cada objeto Pessoa contém centenas de atributos, tais como: Nome, Idade, Endereço, Telefone Residencial, Nacionalidade, Classe Social, etc. Vamos supor que sejam preenchidos todos os atributos do objeto Pessoa referente ao pai, mas ainda falta preencher os dados da mãe e filhos. **Logo, sabe-se que atributos como Endereço, Telefone Residencial, Nacionalidade, etc provavelmente serão idênticos para todos os membros da família.**



Logo, ao invés de se criar objeto para cada membro e preencherê-los um a um integralmente, pode-se clonar o objeto pai já preenchido e modificar apenas os atributos diferentes, como Nome, Idade, etc. Entre outros benefícios, temos: acrescenta e remove produtos em tempo de execução; reduz o número de subclasses; configura dinamicamente uma aplicação com classes; especifica novos objetos pela variação de valores; e especifica novos objetos pela variação da estrutura. *Bacana?*

## Singleton

*Esse padrão garante que uma classe tenha apenas uma instância e provê um ponto de acesso global a ela.*

Esse padrão de projeto deve ser utilizado quando houver a necessidade de existir exatamente uma instância de uma classe e ela deverá ser acessível aos clientes a partir de um ponto de acesso conhecido. Vamos considerar a hipótese de um sistema que trabalhe com diversas conexões com o banco de dados em uma mesma execução. **Imaginem a perda de desempenho ao se instanciar a classe de conexão com o banco de dados várias vezes.**

O Padrão Singleton garante que só haverá uma instância de conexão com o banco de dados e, assim, assegura que – durante a execução – a classe será instanciada apenas uma única vez.



## Padrões Estruturais

INCIDÊNCIA EM PROVA: ALTÍSSIMA

### Adapter

*Esse padrão converte a interface de uma classe em outra interface que normalmente não poderiam trabalhar juntas por serem incompatíveis.*

**Esse padrão de projeto deve ser utilizado quando se quer usar uma classe e sua interface não é compatível com aquela de que você necessita.** Deve ser usado, também, quando se quer criar classes reusáveis que cooperem com classes que não têm necessariamente interfaces compatíveis. Essa classe funciona como um adaptador de tomada. Frequentemente, quer se utilizar uma tomada de três pinos, porém a tomada disponível só permite dois pinos.

Usa-se, então, um adaptador que converte a interface de uma das tomadas permitindo que se faça a conexão normalmente. **Elá permite que um objeto cliente utilize serviços de outros objetos com interfaces diferentes.** Considerem a hipótese de um objeto que contenha dados de um formulário de Vistos de Imigração do sistema da polícia federal, como Data de Nascimento (no formato DD:MM:AAAA).

Porém, esse objeto deve se comunicar com outro objeto do sistema da embaixada americana, que adota o formato MM:DD:AAAA. **Nesse caso, um Adapter seria extremamente oportuno!**

### Bridge

*Esse padrão desacopla uma interface de sua implementação, de forma que ambas possam variar independentemente.*

**Esse padrão de projeto deve ser utilizado quando se deseja evitar um vínculo permanente entre uma abstração e sua implementação.** Ele é recomendado também quando se quer evitar que mudanças na implementação de uma abstração causem impacto nos clientes, isto é, seu código não deve ter que ser recompilado. O Bridge fornece um nível de abstração maior que o Adapter, na medida em que permite variações independentes da interface e da implementação.

Ele provê alto nível de desacoplamento de componentes, fazendo uma ponte entre as hierarquias de classes relacionadas. **Esse padrão de projeto é um pouco complicado de se entender, portanto prestem atenção!** Considerem a hipótese de um conjunto de janelas gráficas que funcionem em diversas plataformas (Windows, Linux, Mac, etc). **Professor, por que não usar o Adapter? De fato, ele poderia adaptar interfaces para cada uma das plataformas.**

No entanto, há dezenas de tipos de janelas gráficas (Diálogo, Aviso, Erro, etc), tornando a utilização inviável para grandes quantidades. **Uma ideia melhor seria criar uma interface para plataformas**



**e outra para janelas.** A primeira conteria os elementos comuns das plataformas e teria classes concretas específicas para implementar janelas em Linux, Mac, etc; e a segunda conteria elementos comuns das janelas e teria classes concretas específicas para implementar janelas Aviso, Erro, etc.

**Logo, ao chamar um método da classe concreta da interface de janelas, ela realizará chamadas aos métodos concretos da interface de plataformas.** Assim, caso se queira adicionar mais tipos de janelas ou mais plataformas, não haverá impacto no cliente, não haverá recompilação de código nem vínculo entre interfaces de um e implementações de outros. O desacoplamento permite que se modifique o código da interface sem alterar a implementação das janelas e vice-versa.

Notem que, assim, pode-se combinar as interfaces e implementações de quaisquer maneiras possíveis – **essa é vantagem do padrão Bridge.** Esse padrão nos lembra do padrão *bridge*.

## Composite

*Esse padrão compõe objetos em estruturas de árvore para representar hierarquias parte-todo, permitindo aos clientes tratarem objetos individuais e composições de objetos uniformemente.*

Esse padrão de projeto deve ser utilizado quando se deseja representar hierarquias parte-todo de objetos e quando se deseja que os clientes ignorem a diferença entre composições de objetos e objetos individuais. **Assim, eles tratarão todos os objetos em uma estrutura composta uniformemente.** Considerem a hipótese de uma interface gráfica composta de vários objetos. Em muitos casos, esses objetos são compostos de outros objetos. Exemplo de implementação:

Por meio de uma árvore de forma transparente para o cliente – ele não precisa saber de nada disso. Assim, ele não deve diferenciar objetos individuais (folhas) de objetos compostos (nós).

## Decorator

*Esse padrão anexa responsabilidades adicionais a um objeto dinamicamente. Fornece uma alternativa flexível em relação à herança para estender funcionalidades.*

Esse padrão de projeto deve ser utilizado quando se quer adicionar responsabilidades a objetos individuais dinâmica e transparentemente, **isto é, sem afetar outros objetos.** Também é utilizado quando extensões por subclasses forem impraticáveis, tendo em vista o possível número de extensões independentes. Considerem a hipótese de um Subway! Usando-se herança, cria-se uma classe abstrata sanduíche e várias classes concretas para implementá-la.

No entanto, há sanduíches com almôndega, sem queijo, com tomate, sem cebola, etc. É customizável, portanto é completamente inviável construir classes concretas para cada tipo de sanduíche, devido a gigantesca quantidade de combinações. O Padrão Decorator sugere que o objeto sanduíche anexe diversas responsabilidades dinamicamente. **Dessa forma, em tempo de execução, à medida que se adicione um novo ingrediente, cria-se mais uma responsabilidade.**



Ao contrário da herança, que aplica funcionalidades a todos os seus objetos, o Decorator aplica funcionalidades apenas a um objeto específico.

## Façade

*Esse padrão oferece uma interface unificada para um conjunto de interfaces em um subsistema, definindo uma interface de alto nível que facilita a utilização do subsistema.*

O Padrão de Projeto Façade (também é um dos mais importantes) deve ser utilizado quando se desejar fornecer uma interface simples para um subsistema complexo ou também quando se deseja dividir em camadas os subsistemas. **É também utilizando quando existem diversas dependências entre clientes e as classes de implementação de uma abstração.** Considerem a hipótese de um banco com um sistema legado de informações de crédito muito complexo.

Esse sistema deve interagir com um sistema de banco de dados recém implementado e moderno. Para que ele acesse o sistema legado, pode-se utilizar o padrão Façade, uma vez que ele facilita a utilização por meio de uma interface de alto nível e, dessa forma, não há necessidade de se interagir diretamente com o sistema complexo.

## Flyweight

*Esse padrão utiliza compartilhamento para suportar eficientemente grandes quantidades de objetos de baixa granularidade.*

**Esse padrão de projeto deve ser utilizado quando uma aplicação usa grande número de objetos e os custos de armazenamento forem altos.** Deve ser utilizado, também, quando muitos grupos de objetos puderem ser substituídos por relativamente poucos objetos compartilhados, uma vez que os estados extrínsecos forem removidos. Considerem a hipótese de um texto escrito no Microsoft Word, em que cada caractere seja um objeto que contenha posição, formato e tamanho.

isso poderia ocupar toda memória disponível no sistema, mas percebam que vários caracteres se repetem diversas vezes (Ex: letra A) e eles são exatamente iguais, muda-se somente a posição (ex: 200 ocorrências da letra A em várias posições). Logo, cada caractere deve possuir uma referência para um objeto Flyweight, que deverá ser compartilhado por todas as instâncias do mesmo caractere do documento, exceto a posição que será variável.

Dessa forma, em vez de armazenarem 800 objetos com três atributos, armazenam-se 800 objetos com um atributo e ponteiro para o objeto Flyweight.

## Proxy

*Esse padrão provê um substituto ou ponto através do qual um objeto pode controlar o acesso a outro objeto.*



Esse padrão de projeto deve ser utilizado quando houver uma necessidade de uma referência mais versátil ou sofisticada para um objeto do que um simples ponteiro. **Por exemplo, proxies virtuais criam objetos caros por demanda e proxies de proteção controlam o acesso ao objeto original.** Considerem a hipótese de um sistema que acesse um banco de dados por meio de uma classe de conexão.

**No entanto, por medidas de segurança, vamos supor que se deseja que esse sistema não tenha acesso direto ao banco de dados referido.** Dessa forma, o usuário se conectará ao Proxy (isto é, a classe substituta ou suplente) e o Proxy que irá se conectar ao banco de dados. Claro, tudo isso ocorrendo de maneira transparente para o usuário.



## Padrões Comportamentais

INCIDÊNCIA EM PROVA: MÉDIA

### Chain of Responsability

*Esse padrão evita o acoplamento do remetente de uma requisição ao seu receptor ao dar a mais de um objeto a chance de lidar com a requisição.*

Esse padrão de projeto deve ser utilizado quando se deseja emitir uma solicitação para um dentre vários objetos, sem especificar explicitamente o receptor ou quando mais de um objeto é capaz de lidar com a requisição e ele não for conhecido *a priori*. **Esse padrão também é comumente utilizado quando um conjunto de objetos que podem lidar com uma requisição forem especificados dinamicamente.**

Esse padrão acaba com estruturas de decisão ao criar uma cadeia de objetos em que se passa a responsabilidade até encontrar aquele que pode respondê-la. Consideremos a hipótese de uma loja virtual que permite pagamento online por meio de diversos bancos. **Dado um parâmetro, deve-se identificar qual banco deve ser utilizado para o pagamento.** Esse problema pode ser resolvido com outros padrões, mas o Chain of Responsibility fornece uma solução com fraco acoplamento.

Assim, cada elemento da cadeia pode implementar a requisição da maneira que quiserem. **Assim, não há uma associação direta entre o remetente e o receptor que irá lidar com a requisição.**

### Command

*Esse padrão encapsula a requisição de um objeto, portanto permitindo que se parametrize os clientes com diferentes requisições.*

Esse padrão de projeto deve ser utilizado quando se deseja parametrizar objetos para realizar alguma execução ou também para especificar, enfileirar e executar requisições a qualquer momento. **Ele também é utilizado para suportar mudanças de log de maneira que possa ser reaplicado no caso de uma queda no sistema.** Entenderam? Considerem a hipótese de um interruptor que ligue ou desligue uma lâmpada.

**Esse interruptor encapsula uma requisição, de tal modo que se possa utilizá-lo para diferentes dispositivos.** Em outras palavras, se eu retirar um interruptor de uma lâmpada, conectar adequadamente aos fios de um computador, é possível ligar/desligar o computador com o mesmo interruptor. Logo, o interruptor tem sua interface encapsulada, logo pode ser utilizado em qualquer dispositivo que tenha uma interface Ligar/Desligar.



Imagine agora uma classe que faz diversas conexões a um banco de dados. Não é recomendável que ela tenha um método que se conecte diretamente ao banco, portanto encapsula-se essa conexão, diminuindo a dependência.

## Interpreter

*Esse padrão, dada uma linguagem, define uma representação para sua gramática em conjunto com um interpretador que utiliza a representação para interpretar sentenças na linguagem.*

Esse padrão de projeto deve ser utilizado quando houver uma linguagem para interpretar e quando se puder representar declarações nessa linguagem como árvores sintáticas abstratas. Ele funciona bem quando a gramática é simples, permitindo um fácil gerenciamento e quando a eficiência não é um fator crítico de sucesso. **Basta lembrar do Java, que é uma linguagem interpretada.**

*O que isso quer dizer esse negócio de interpretada, professor?* Bem, isso significa que o código fonte é compilado em um *bytecode*, que é então posteriormente interpretado por um interpretador. **Dessa forma, esse padrão de projeto é utilizado nos casos em que é possível definir uma linguagem com uma gramática que possa ser, posteriormente, interpretada.**

## Iterator

*Esse padrão fornece uma maneira de acessar elementos de um objeto agregado sequencialmente sem expor sua representação interna.*

Esse padrão de projeto deve ser utilizado quando se deseja acessar o conteúdo de um objeto agregado sem expor a sua representação interna e para fornecer uma interface uniforme para diferentes estruturas de agregação. **Ele também é recomendado para suportar múltiplos acessos a objetos agregados.** Entendido? Vamos ver um exemplo!

Considerem a hipótese de se desejar percorrer sequencialmente quatro coleções distintas (Lista, Arrays, Map e Set) de objetos bastante complexos. **Em uma situação normal, deve-se conhecer a representação interna de cada uma dessas coleções.** O Iterator permite que se percorra todas essas coleções sem precisar saber detalhes de seu funcionamento.

## Mediator

*Esse padrão define um objeto que encapsula a forma como um conjunto de objetos interagem, promovendo um fraco acoplamento ao evitar que objetos se refiram aos outros explicitamente.*

Esse padrão de projeto deve ser utilizado quando um conjunto de objetos se comunicar de maneira bem definida, porém complexa e quando o reúso de um objeto for difícil por referenciar e se comunicar com muitos outros objetos. **Ademais, ele é utilizado quando um comportamento distribuído entre diversas classes puder ser customizado sem a criação de muitas subclasses.**



**Considerem a hipótese de um software complexo com grandes quantidades de classes, de tal modo que a lógica de processamento está distribuída entre elas.** Todo mundo sabe que, há cada manutenção ou refatoração, o desenho do software pode se tornar mais complexo e a comunicação entre as classes pode ser mais difícil de ler entender.

**Logo, o Padrão Mediator permite que a comunicação entre objetos seja encapsulada em um outro objeto mediador.** Assim, não haverá mais uma comunicação direta entre as classes, reduzindo o acoplamento, garantindo que todos fiquem mais livres para serem modificados e diminuindo a complexidade de comunicação entre objetos.

## Memento

*Esse padrão captura e externaliza o estado interno de um objeto, sem violar seu encapsulamento, de maneira que o objeto possa ser restaurado posteriormente.*

**Esse padrão de projeto deve ser utilizado quando uma parte do estado de um objeto precisar ser armazenada, de forma que possa ser recuperada posteriormente.** Ele é utilizado, também, para evitar que uma interface direta para obter um estado exponha detalhes de implementação e quebrem o encapsulamento do objeto. Considerem a hipótese de um editor de texto que guarde o estado interno do objeto.

Em outras palavras, o que o usuário está digitando, aí ele deleta uma palavra, mas se arrepende e deseja voltar ao estado anterior. **Bem, o memento oferece uma maneira de desfazer ações sem ter acesso ao que estava sendo escrito, ele apenas retorna o estado anterior, mas sem olhar o que estava escrito.** Logo, o Padrão Memento permite a captura e externalização do estado interno de um objeto, sem violar seu encapsulamento.

Ele captura o que estava escrito e mostra ao usuário, mas sem acesso direto ao que estava escrito. **Assim, pode-se cancelar operações e desfazer alterações para retornar ao estado anterior.**

## Observer

*Esse padrão define uma dependência um-para-muitos entre objetos para que, quando um objeto mudar de estado, os seus dependentes sejam notificados e atualizados automaticamente.*

Esse padrão de projeto deve ser utilizado quando uma mudança em um objeto requisitar mudanças em outros objetos e não se souber quantos objetos necessitam ser modificados. **Ele também é utilizado quando uma abstração possuir dois aspectos, sendo um dependente do outro.** Além disso, sua utilização é recomendada quando um objeto for capaz de notificar outros sem assumir quem são. Certinho?



Considerem a hipótese de uma tabela de classificação do campeonato brasileiro com um gráfico de pizza informando vitórias, empates e derrotas de um determinado time, assim como um gráfico com a variação de posição do time do campeonato. **Aí chega o domingo, ocorrem 10 jogos e o estagiário altera a tabela com os novos dados.** E agora? Tem que atualizar os gráficos um a um? Os gráficos ficarão desatualizados? Não haverá nem uma notificação de novos dados?

Ele cria uma dependência dos gráficos em relação à tabela de modo que, quando a tabela muda de estado, os gráficos são atualizados automaticamente.

## State

*Esse padrão permite a um objeto alterar o seu comportamento quando o seu estado interno for modificado.*

Esse padrão de projeto deve ser utilizado quando o comportamento de um objeto depender de seu estado e ele deve mudar este comportamento em tempo de execução de acordo com este estado.

**Ademais, é recomendado quando operações tiverem declarações condicionais grandes que dependam do estado do objeto.** Considerem a hipótese do jogo *Super Mario Bros!* Lembram-se de quando ele conseguia uma flor de fogo?

**Se ele estivesse pequeno, ele ficava grande e pegando fogo! Se ele estivesse grande, ele ficava pegando fogo!** Se já estivesse pegando fogo, ganhava 1000 pontos! Se tivesse com uma capa de voo, perdia a capa e ficava pegando fogo! Logo, notem que o estado futuro depende de seu estado atual e o estado futuro é decidido em tempo de execução, isto é, durante o jogo. Esse padrão elimina a necessidade de condicionais complexos. *Por que?*

Porque pode haver dezenas ou centenas de estados possíveis. A grande vantagem é que esta solução torna mais simples adicionar estados e suas transições.

## Strategy

*Esse padrão define uma família de algoritmos, encapsula cada um e faz deles intercambiáveis.*

Esse padrão de projeto deve ser utilizado quando várias classes relacionadas diferirem apenas em seus comportamentos e que houver necessidade de diferentes variantes de um algoritmo. **Ele também é utilizado quando uma classe definir muitos comportamentos e eles aparecerem como declarações condicionais em suas operações.** Considerem a hipótese de uma escola querer organizar os meninos e as meninas por ordem de idade.

**Há uma família de algoritmos capaz de realizar essa operação (BubbleSort, QuickSort, Heapsort, etc).** Uma maneira de realizar essa operação é por meio de operadores condicionais (if-else), mas isso pode ser trabalhoso em grandes quantidades. Uma boa estratégia seria encapsular



os algoritmos de ordenação, de modo que se possa trocar de algoritmo sempre que quiser. Basta chamar o método OrdenaAluno com os parâmetros sexoAluno e algoritmoOrdenacao.

Assim, é possível chamar o método ordenaAluno(menino, bubbleSort) ou ordenaAluno(menina, mergeSort), etc, diminuindo o acoplamento.

## Template Method

*Esse padrão define o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses.*

Esse padrão de projeto deve ser utilizado quando se deseja implementar a parte invariante de um algoritmo e deixar que as subclasses implementem o comportamento variável. **Ele também é recomendado quando comportamentos comuns entre subclasses forem fatorados e localizados em uma classe comum, para evitar duplicação de código.** Considerem a hipótese de uma franquia de McDonald's fabricando um sanduíche.

**Bem, o cliente pode pedir para colocar mais queijo, tirar os picles, adicionar ketchup, não colocar sal, entre outras opções.** No entanto, observem que operações como: abrir o pão, posicionar o hambúrguer no meio e fechar o pão são operações que sempre irão ocorrer. Logo, esse esqueleto de algoritmo jamais irá mudar. Já as opções de condimentos, verduras, etc podem ser modificadas de acordo com o gosto do cliente.

Sendo assim, a parte invariável será implementada na classe abstrata e a parte variável será implementada pelas classes filhas de acordo com as necessidades do usuário.

## Visitor

*Esse padrão representa uma operação a ser realizada sobre elementos de uma estrutura de objetos e permite definir uma operação sem mudar as classes dos elementos sobre os quais opera.*

Esse padrão deve ser utilizado quando muitas operações distintas e não relacionadas precisarem ser executadas sobre uma estrutura de objetos e se quer evitar a poluição das classes com essas operações. **Ademais, são recomendadas quando as classes que definem a estrutura do objeto raramente forem modificadas.** Considerem a hipótese de uma compra em um supermercado. O carrinho de compras é a estrutura de dados que contém um conjunto de elementos.

Ao finalizar a compra, deve-se passar a responsabilidade para o caixa (Visitor). A partir deste momento, ele estará no comando, porque ele que realizará a soma dos preços, pesará as frutas, verduras, vegetais, etc (entre outras operações). **Logo, algumas operações se aplicam a alguns elementos, mas não se aplicam a outros, isto é, uma cerveja já possui um valor, mas uma maçã precisa ser pesada.** No entanto, ambas fazem parte da mesma estrutura de dados.



Galera, vou ser bem sincero com vocês: esse assunto é muuuuuuuuuuuuito chato – e também é muito decoreba! **Então, vou explicar o que eu recomendo para o estudo dessa aula – fiz uma análise estatístico rápida e cheguei a algumas conclusões!** Primeiro de tudo, decore as frases porque cerca de 15% das questões de prova exigem apenas saber qual Padrão de Projeto GOF pertence a qual categoria.



Vocês matam 15% das questões apenas decorando aquelas duas frases, certinho? Segundo, se você estiver sem tempo, ignore a categoria comportamental, visto a distribuição de questões segue a tabela mostrada acima – e ela é a que tem mais padrões. **Por fim, dos 23 Padrões de Projeto, seis correspondem à 55% das questões de prova (dos 85% restantes, isto é, sem as questões de categorias).** São eles, em ordem:

ADAPTER	1º
SINGLETON	2º
FAÇADE	3º
ABST. FACTORY	4º
ITERATOR	5º
BRIDGE	6º

Pessoal, se vocês estiverem com o tempo muito apertado, estudem apenas esses acima! **Do mesmo modo, podemos afirmar que, dos 23 Padrões de Projeto, onze correspondem à 15% das questões de prova:** Decorator, Flyweight, Proxy, Chain of Responsibility, Interpreter, Mediator, Memento, State, Strategy e Template Method. Os seis padrões restantes, estão no meio termo e correspondem a 30% das questões de prova.

FALOU EM...	... PROVAVELMENTE SERÁ:
Criar famílias de objetos relacionados...	ABSTRACT FACTORY
Construção de um objeto complexo com diferentes representações...	BUILDER
Deixar subclasses decidirem...	FACTORY METHOD
Criar uma instância prototípica...	PROTOTYPE
Apenas uma instância com um ponto global a ela...	SINGLETON



Converte uma interface em outra, por serem incompatíveis...	ADAPTER
Desacoplar interface da implementação...	BRIDGE
Estruturas de árvore em hierarquia parte-todo...	COMPOSITE
Anexa responsabilidades adicionais dinamicamente...	DECORATOR
Interface unificada de alto nível para simplificar outra complexa...	FAÇADE
Compartilhamento para suportar grandes quantidades de objetos...	FLYWEIGHT
Prover substituto para controlar um objeto	PROXY
Evitar o acoplamento dando oportunidade a outros objetos...	CHAIN OF RESPONSIBILITY
Encapsula requisição de objetos...	COMMAND
Representação de uma gramática...	INTERPRETER
Interface única para acessar coleções sequencialmente...	ITERATOR
Encapsula a forma como objetos interagem...	MEDIATOR
Captura o estado interno de um objeto...	MEMENTO
Quando objeto mudar de estado, notifica os dependentes...	OBSERVER
Altera comportamentos quando modificar o estado interno...	STATE
Família de algoritmos...	STRATEGY
Esqueleto de algoritmos...	TEMPLATE METHOD
Operação a ser realizada sobre uma estrutura de objetos...	VISITOR



## RESUMO

		Propósito		
		Criação	Estrutura	Comportamento
Escopo	Objeto	Builder Abstract Factory Prototype Singleton Factory Method	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor Interpreter Template Method

PADRÕES CRIACIONAIS	Descrição
BUILDER	Esse padrão separa a construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes tipos de representações.
ABSTRACT FACTORY	Esse padrão fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
PROTOTYPE	Esse padrão especifica os tipos de objetos para criar usando uma instância como protótipo e cria novos objetos copiando este protótipo.
SINGLETON	Esse padrão garante que uma classe tenha apenas uma instância e provê um ponto de acesso global a ela.
FACTORY METHOD	Esse padrão define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar.

PADRÕES ESTRUTURAIS	Descrição
ADAPTER	Esse padrão converte a interface de uma classe em outra interface que normalmente não poderiam trabalhar juntas por serem incompatíveis.
BRIDGE	Esse padrão desacopla uma interface de sua implementação, de forma que ambas possam variar independentemente.
COMPOSITE	Esse padrão compõe objetos em estruturas de árvore para representar hierarquias parte-todo, permitindo aos clientes tratarem objetos individuais e composições de objetos uniformemente.



<b>DECORATOR</b>	Esse padrão anexa responsabilidades adicionais a um objeto dinamicamente. Fornece uma alternativa flexível em relação à herança para estender funcionalidades.
<b>FACADE</b>	Esse padrão oferece uma interface unificada para um conjunto de interfaces em um subsistema, definindo uma interface de alto nível que facilita a utilização do subsistema.
<b>FLYWEIGHT</b>	Esse padrão utiliza compartilhamento para suportar eficientemente grandes quantidades de objetos de baixa granularidade.
<b>PROXY</b>	Esse padrão provê um substituto ou ponto através do qual um objeto pode controlar o acesso a outro objeto.

<b>PADRÕES COMPORTAMENTAIS</b>	<b>Descrição</b>
<b>CHAIN OF RESPONSIBILITY</b>	Esse padrão evita o acoplamento do remetente de uma requisição ao seu receptor ao dar a mais de um objeto a chance de lidar com a requisição.
<b>COMMAND</b>	Esse padrão encapsula a requisição de um objeto, portanto permitindo que se parametrize os clientes com diferentes requisições.
<b>ITERATOR</b>	Esse padrão fornece uma maneira de acessar elementos de um objeto agregado sequencialmente sem expor sua representação interna.
<b>MEDIATOR</b>	Esse padrão define um objeto que encapsula a forma como um conjunto de objetos interagem, promovendo um fraco acoplamento ao evitar que objetos se refiram aos outros explicitamente.
<b>MEMENTO</b>	Esse padrão captura e externaliza o estado interno de um objeto, sem violar seu encapsulamento, de maneira que o objeto possa ser restaurado posteriormente.
<b>OBSERVER</b>	Esse padrão define uma dependência um-para-muitos entre objetos para que, quando um objeto mudar de estado, os seus dependentes sejam notificados e atualizados automaticamente.
<b>STATE</b>	Esse padrão permite a um objeto alterar o seu comportamento quando o seu estado interno for modificado.
<b>STRATEGY</b>	Esse padrão define uma família de algoritmos, encapsula cada um e faz deles intercambiáveis.
<b>VISITOR</b>	Esse padrão representa uma operação a ser realizada sobre elementos de uma estrutura de objetos e permite definir uma operação sem mudar as classes dos elementos sobre os quais opera.
<b>INTERPRETER</b>	Esse padrão, dada uma linguagem, define uma representação para sua gramática em conjunto com um interpretador que utiliza a representação para interpretar sentenças na linguagem.
<b>TEMPLATE METHOD</b>	Esse padrão define o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses.

