**Design Pattern**

Soluções que foram criadas para resolver problemas repetitivos (Para não precisar reinventar a roda) (Com o passar do tempo, desenvolvedores experientes, passaram por problemas, e resolveram criar padrões para resolvê-los). São padrões seguidos para tornar seu código mais legível, reaproveitável e o tornando mais fácil para mudanças. Facilitando a comunicação da equipe, e a ingressão de novos integrantes ao time (É um padrão de código para soluções de determinado problema na criação de software).

Abstração – Olhar parar o mundo real e transformar em objetos dentro do desenvolvimento, ‘conseguir olhar para uma cadeira e conseguir abstrair suas propriedades’.

Herança – Compartilhamento de características entre objetos.

Encapsulamento – Definição de níveis de acesso.

Polimorfismo – Um único objeto tem a capacidade de se comportar de maneiras diferentes.

**Princípios de Programação**

“Verdade básica imutável”, um dos principais objetivos da orientação a objeto é a abstração.

Ajudar a criar códigos mais flexíveis e de fácil manutenção, e mais legíveis.

\*Solid não é um desing pattern, e sim um princípio de programação

**Princípios SOLID**

A missão de todo desenvolvedor de software é criar um código de qualidade durante toda a fase de desenvolvimento.

Esse termo Solid foi criado por Michael Feathers apor observar um ensaio escrito em 2000 por Robert Martin(Uncle Bob), que reuniu um conjunto de boas pratinhas na codificação que já existiam em torno de um tema centrar de gerenciamento de dependências, e discutiu a importância da utilização das mesmas em conjuntos para garantir a longevidade de um software de sucesso, melhorando sua manutenção, coesão etc...

Esses princípios ajudam o programador a escrever [códigos mais limpos](https://medium.com/joaorobertopb/1-clean-code-o-que-%C3%A9-porque-usar-1e4f9f4454c6), separando responsabilidades, diminuindo acoplamentos, facilitando na refatoração e estimulando o reaproveitamento do código.

Conceitos importantes

**Acoplamento –** Acoplamento forte uma classe depende muito da outra, e uma alteração em uma se propaga na outra, assim se tornando mais difícil de entender as mesmas, pois as duas estão muito interconectadas, assim sendo mais difícil de ser reutilizada.

**Coesão** – Mede o grau que a classe e o método fazem sentido, medindo o quão claro é o entendimento dela.

Assim é possível identificar que o código é preciso ter um fraco acoplamento e forte coesão

**SRP – Single Responsability Principle – Princípio da Responsabilidade única** “Nunca deve haver mais de um motivo para uma classe mudar”

Uma classe deve ter apenas uma responsabilidade, e seus métodos e atributos devem existir para atender apenas o objetivo principal da classe. (Não utilizar “God Class”, ou seja, uma classe deus com todas as informações do sistema) Para facilitar a manutenção dessa classe, pois caso ela contenha mais de uma responsabilidade, com uma alteração, será atingido mais de um lugar.

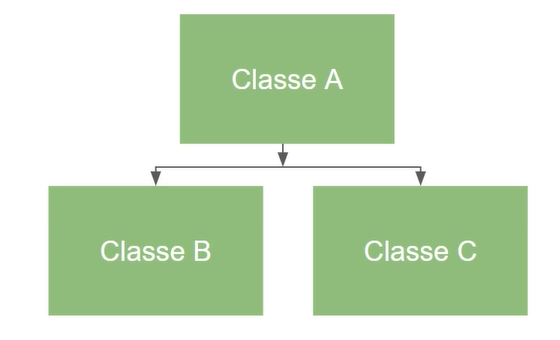
Esse princípio declara que uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.

**OCP – Open/Closed Principle – Princípio Aberto/Fechado**

Trabalhar com código extensível, porém evitando quebrar o que já foi implementado.

“Uma entidade (que pode ser uma classe, modulo ou método) deve ser fechado para modificação e aberto para extensão”. Ou seja, a entidade pode ser estendida (Adicionando mais recursos), mas o que já foi implementado não pode ser alterado, pois as funcionalidades dela já devem estar sendo utilizadas por outras entidades.

Exemplo:



Como pode ser visto, a Classe A é a classe pai, e a Classe B e Classe C são heranças dela. Se for alterado algo na classe A que interfira no funcionamento da B e C, está ocorrendo uma violação do princípio OCP, onde não se deve modificar coisas que alterem ou interfiram nas classes que já implementam a mesma.

Mas mudanças acontecem sempre, dessa forma caso ocorra uma mudança significativa na entidade é importante utilizar uma forma de versionamento, para não interferir no funcionamento de fluxos antigos.

Esse princípio se resume a mudanças, e utilizando é criado um código legível, extensível e de fácil manutenção.

Herança e interface são maneiras de aplicar esse princípio.

Uma maneira também de exemplificar o princípio de open/close é a questão de atributos e métodos privados/públicos, sempre é importante identificar esse controle da classe, se aquele atributo será apenas acessado/alterado por métodos da classe, ou se ele será aberto para qualquer um. Dificultando a mudança do estado dos objetos da classe. (O que eu devo deixar aberto para modificação do comportamento da classe, e o que devo deixar fechado, essa é uma pergunta muito importante), (Tudo que é deixado visível para todos, uma vez implementado, é difícil voltar atras, por isso é muito importante identificar o que deixar público de privado de uma classe etc...)

**LSP – Liskov Substitution Principle – Princípio da Substituição de Liskov**

“Uma classe derivada (herança) pode ser substituível por sua classe base sem quebrar a aplicação” (Os objetos de uma subclasse devem ser substituídos por objetos da superclasse sem parar o funcionamento do código) ou seja, os objetos da subclasse devem se comportar da mesma maneira que os objetos da superclasse.

Uma maneira de visualizar esse princípio é quebra de um contrato, por exemplo existe uma classe onde contém um método que recebe um int e retorna um int, se esse envio e retorno forem alterados por exemplo para um double, irá impactar em todos os clientes que utilizam esse método. Por exemplo, você disponibiliza uma API, e após um tempo altera os parâmetros de entrada e saída, isso irá impactar dos sistemas clientes

A utilização de interfaces (Contratos) é uma maneira para que esse princípio seja cumprido, pois classes relacionadas iram utilizar a mesma assinatura.

//if (empregado.GetType() != typeof(EmpregadoVoluntario))

**ISP – Interface Segregation Principle – Princípio da Segregação de Interface**

“Os clientes não devem ser forçados a depender de métodos que eles não utilizam” dessa forma devem ser identificados interfaces ou classes abstratas que estão muito infladas (Forçando a criação de muitos metodos) e transformarem em interfaces mais resumidas, que declaram métodos para cada situação que realmente são necessários.

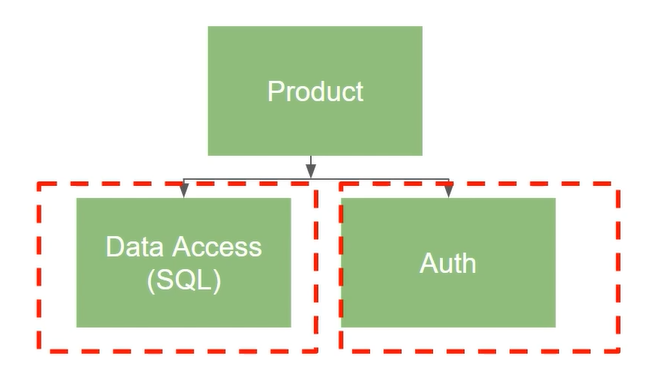
Assim minimizando a dependências de membros não utilizados e reduzindo o acoplamento de código.

**DIP – Dependency Inversion Principle – Princípio da Inversão de Dependência**

“Um modulo de alto nível não pode depender de um modulo de baixo nível, ambos devem depender apenas de abstrações”. “Abstrações não devem depender de detalhes (Implementação). Os detalhes devem depender de abstrações”.

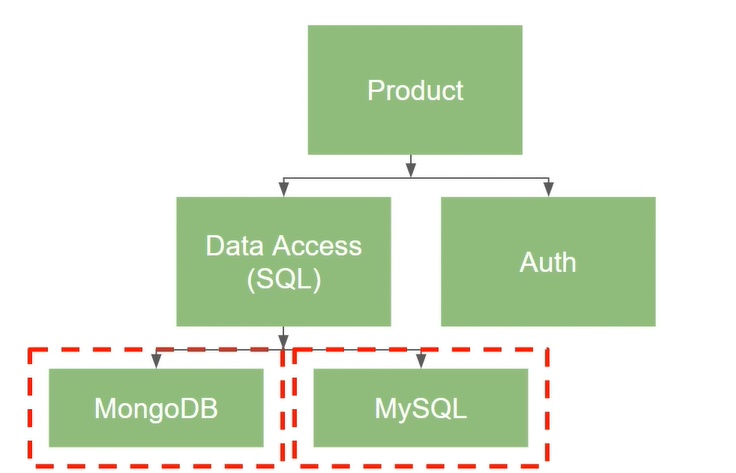
Primeiramente módulos de alto nível é o lugar onde estão as regras de negócio da aplicação (mostrando o que precisa ser feito), já os módulos de baixo nível são as tarefas internas realizadas pela aplicação para cumprir as regras de negócio (determinado como será feito).

Exemplo:



Nesse caso, o modulo de alto nível seria “Product” da aplicação, contendo as regras de negócio dos produtos, como os mesmos serão gerenciados etc...

E abaixo temos os módulos de baixo nível em relação a “Product” definindo como será acessados os dados desses produtos etc.



Porem essa situação não é imutável, pode ser visualizado na imagem acima que existem dois módulos de baixo nível em relação a “Data Access”.

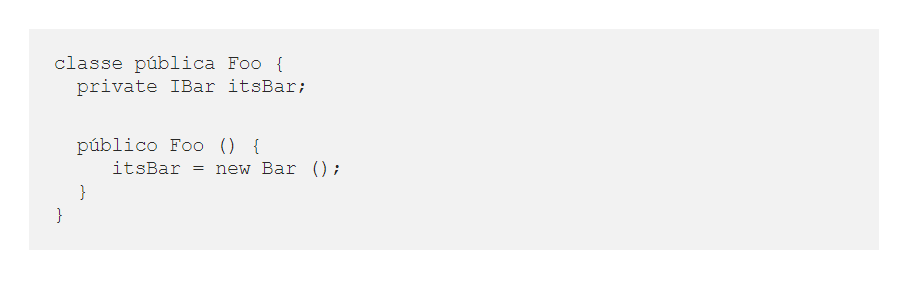
Dessa forma, o modulo “Data Access” é o modulo de alto nível em relação aos módulos “MongoDB” e “MySQL”, mas também é um modulo de baixo nível em relação a “Product”

ConfigurationManager.AppSettings["DB"] == "SQLSERVER"

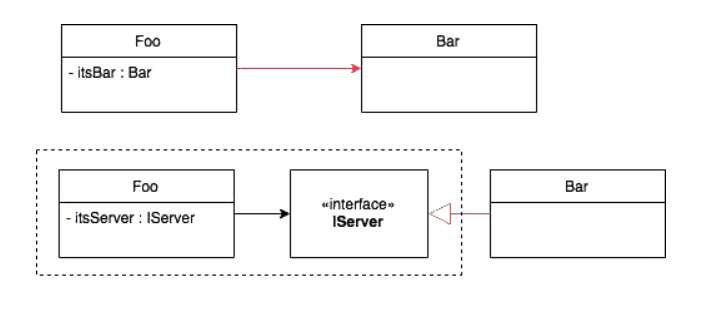
**Dependency Inversion vs Injection**

Vamos utilizar o código abaixo para exemplificar o princípio de inversão de dependência.

A classe Foo está acoplada a classe Bar ou seja, dependendo dela e violando o principio



Para que o princípio seja cumprido, a classe Foo deve depender de uma abstração, é não da classe concreta em si, resolvemos isso fazendo com que ela dependa de uma interface, e não da classe Bar.



Muitas pessoas acabam visualizando o IServer e o Bar como uma entidade só, mas na verdade não, pois nesse caso o IServer pertence a classe Foo, e a classe Bar deve obedecer ao contrato para efetuar a comunicação com Foo.

Dessa forma, podemos ver no código abaixo, onde a classe Foo passa a depender de uma abstração, e não da classe em si. Porem a mesma ainda é responsável pelo instanciamento do objeto. Através do ServiceLocator.



Utilizando a Injeção de dependência ocorre uma inversão de controle, onde é injetado no método (Através do construtor, propriedade etc...) o objeto necessário de maneira externa. Ou seja, a classe Foo não tem nenhuma obrigação do instanciamento desse objeto.



Usamos Dependency Inversion para separar os módulos por abstração e Dependency Injection para eliminar a instanciação manual. [Implementamos](https://distillery.com/contact/) tudo através de um framework de acordo com o princípio da Inversão de Controle

**Padrão GoF – Criacional, Estrutural e Comportamental**

**Classificação por proposito.**

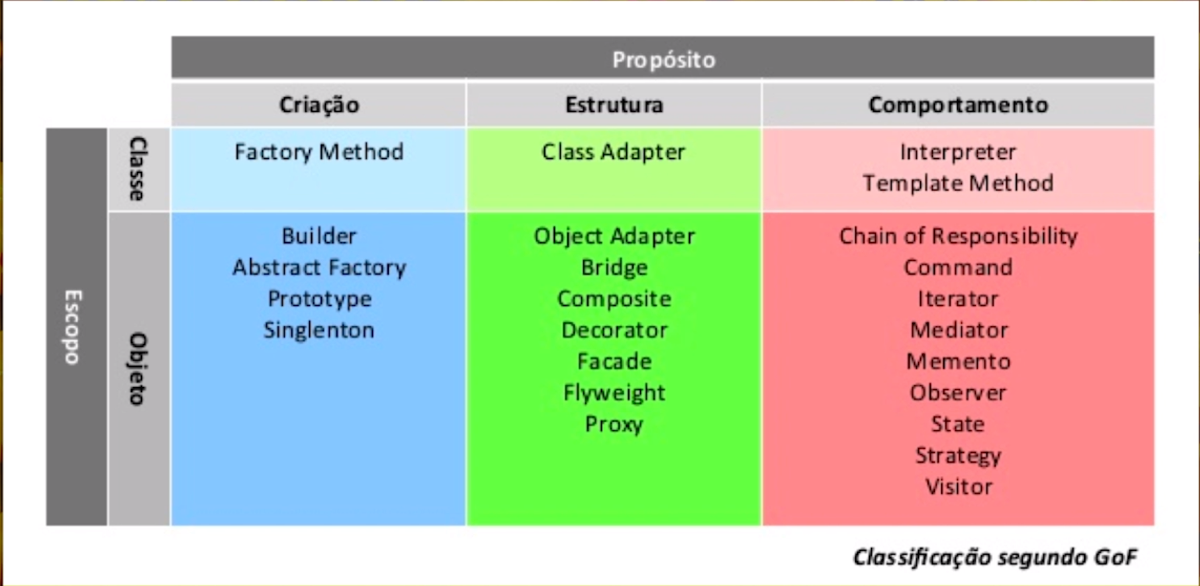
**Criacional** – Trabalham com a criação de objetos flexíveis e reutilizáveis

**Estrutural** – Define como trabalhar com objetos e classes em estruturas maiores e complexas, e ainda assim flexíveis e eficientes.

Como será criada e feita a estrutura do projeto

**Comportamental –** Trabalham com a comunicação eficiente e designam as responsabilidades comuns aos objetos.

**Classificação por Escopo** (Se o padrão se aplica principalmente a classes ou objetos).



**Criacional**

Essa categoria tem como objetivo a instanciação de objetos. E sustenta o princípio mais importante “Programe para interfaces e não para implementação”.

* Abstraem e/ou adiam o processo criação dos objetos.
* Eles ajudam a tornar um sistema independente de como sues objetos são criados, compostos e representados.
* Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegara a instanciação para outro objeto.
* Os padrões de criação tornam-se importantes à medida que os sistemas evoluem no sentido de dependerem mais da composição de objetos do que a herança de classes.
* A configuração pode ser estática (específica em tempo de compilação) ou dinâmica (Em tempo de execução).

**Singleton –**

O problema que irá resolver – 1° Precisamos controlar o acesso as instancias da classe, garantindo que seja instanciada apenas uma vez. 2° Reduzir a utilização de memória. 3° - Fornecer mais flexibilidade que a utilização de estruturas estáticas.

Objetivo – A utilização e manutenção de apenas uma instancia.

Característica – 1° Define uma operação responsável por garantir o acesso a uma única instancia. 2° Responsável por criar e manter sua própria instancia única.

Um exemplo de abstrair esse problema é pensar em um jogo de futebol. No mesmo existe apenas uma bola em campo, que é tocada pelos jogadores de ambos os times, caso outra bola caia em campo, o jogo é parado imediatamente.

**Factory Method** –

O problema que irá resolver – 1° Eu não posso antecipar o tipo da classe de objeto que deve ser criada. Pois, tenho objetos de vários tipos. 2° Preciso delegar a responsabilidade para que as subclasses especifiquem os objetos que criam.

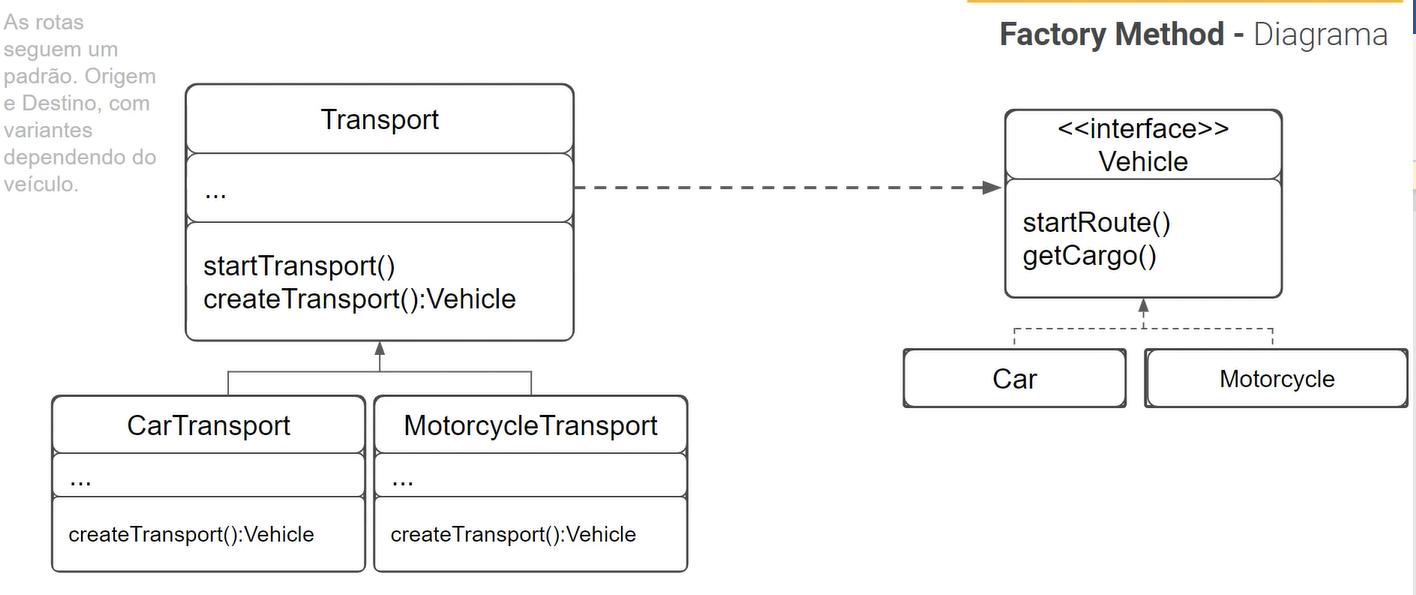
Objetivo – Também conhecido como Construtor Virtual, o objetivo desse padrão é a possibilidade de criar uma fábrica de objetos em tempo de execução e deixar que o objeto decida seu tipo dinamicamente.

Características – O Factory Method dispões uma interface que cria um objeto, mas são as subclasses, que irão decidir qual classe concreta instanciar.

Exemplo, um aplicativo de viagem, onde os usuários contratam carros, para levar o cliente até determinado lugar.

Depois de um tempo a empresa aumenta, e uma nova funcionalidade deverá ficar disponível, entrega de delivery, com motos. Dessa forma podemos visualizar que ocorre uma mudança de regra de negócios enorme. O que acontece caso o código do sistema esteja engessado para atender apenas carros (Um código muito engessado, com difícil manutenção)?

O Factory method vem para resolver esse problema. Nesse padrão deverá ser substituído as chamadas diretas de carros, para um método intermediário que fara o papel de “fábrica”, esse método será responsável por controlar a criação de objetos instanciados. Ou seja, irá controlar quando deverá retornar um objeto de carro ou moto (Classes concretas), e a classe que está referenciando essa factory estará dependendo apenas de uma abstração de veículos e não da classe concreta em questão.



**Abstract Factory** –

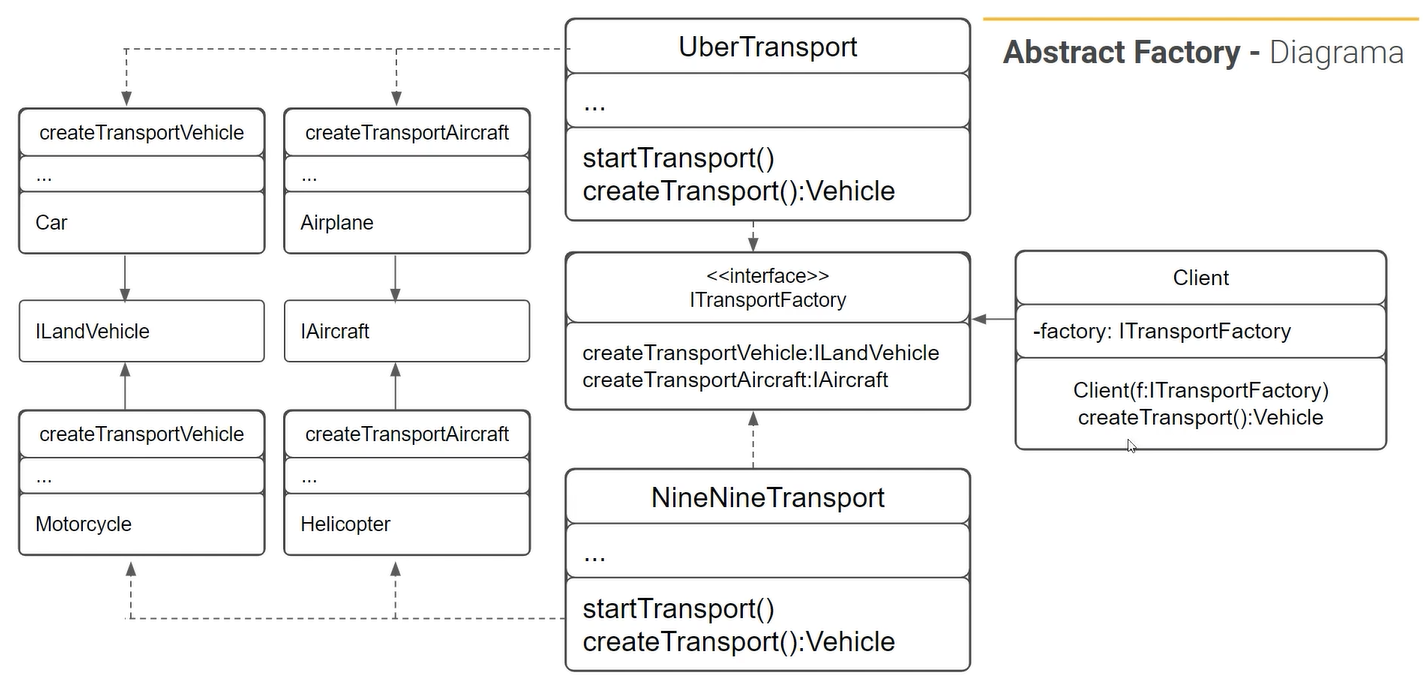
O problema que irá resolver – O princípio de Factory Method e Abstract Factory são parecidos, a diferença é que Abstract cria uma instancia de várias famílias de classes, já o Factory cria uma instancia de várias classes derivadas.

Objetivo – Fornecer uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Características – 1º Elimina a necessidade de anexar classes especificas das aplicações no código. 2° Fornece um gancho para as subclasses. A criação de classes e subclasses se torna mais flexível. 3° Porem, como desvantagem podemos citar o fato de que podemos ter que criar varias subclasses da classe Creator apenas para criar um ConcretProduct.

Utilizando o exemplo anterior, foi corrigido a regra onde a aplicação só funcionava para carros, agora ela está funcionando através de abstrações de veículos, porém agora temos outro problema, digamos que a aplicação anterior era o Uber, e agora entrara o cliente 99, e nele não será disponibilizado apenas veículos terrestres, também serão disponibilizadas aeronaves divididas em helicópteros e jatos, entre outros. E também agora serão dois clientes (Uber, 99), e cada um poderá disponibilizar 1 ou mais dos tipos de veículo disponíveis. Ou seja, o problema agora é que teremos famílias de produtos.

Com Abstract Factory primeiro devemos declarar interfaces para cada produto distinto da família de produtos (Ex: Veículos e Aeronaves), e assim fazer com que todas as variantes desses produtos seguirem as interfaces. Em seguida declaramos uma interface com uma lista de métodos de criação para todos os produtos que fazem parte da família de produtos (Ex: CriarTransporteVeiculo, CriarTransporteAeronave). E esses métodos deveram retornar tipos abstratos de produtos representados pelas interfaces (Veículo, Aeronave etc...).



**Builder** –

Objetivo – Separar a construção de um objeto complexo de sua representação para que o mesmo processo de construção possa criar diferentes representações.

Características – 1° Permite variar a representação interna de um produto. 2° Encapsula o código entre construção e representação. 3° Provê controle durante o processo de construção. 4° Como desvantagem, requer criar um concrete builder especifico para cada instancia diferente do produto.

Quando criamos um objeto com muitas regras de negócio, métodos, é sempre um risco termos construtores com inúmeros parâmetros. Este é um problema, uma vez que situações que não precisem de tantos parâmetros podem fazer uma serie de parâmetros nulll, ou vazio. Na verdade, métodos que passam de 3 a 4 parâmetros podem confundir a clareza do código e se enquadrar no que é chamado de Long Parameter List. Para ilustrar imagine o objeto Carro. Para construir um carro, você precisa de motor, carroceria, rodas, combustível. Mas e se você quiser uma Caminhonete? Ou um Caminhão.

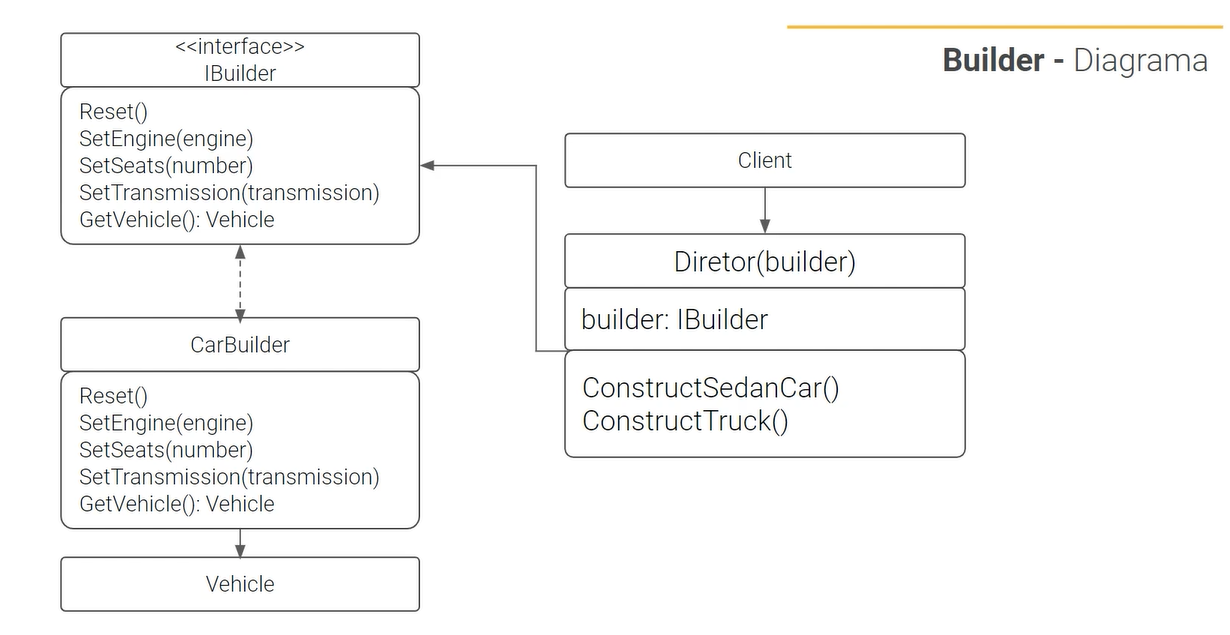
Você pode pensar em criar uma classe para cada tipo de carro, mas terá uma infinidade de classes, e o Builder vem para ajudar a resolver isso.

O Builder é sempre utilizado quando vamos construir objetos complexos. Nesse padrão iremos quebrar o processo de criação do objeto em etapas, e em vários métodos, como ferramentas e componentes. Preciso de motor, ok a classe Motor estará disponível, preciso de uma caçamba, temos o Componente Caçamba, assim estamos criando uma caminhonete.

Esses componentes serão chamados builders, que são os itens para construir o objeto complexo.

A sequência da construção, será guiada por um Diretor, que recebe quais os métodos ele precisa, e sempre que solicitado, dispara a sequência para os builders, que iram construir o objeto.

Assim, temos uma biblioteca de componentes e ferramentas, que são acessados pelo diretor quando provocado a construir algum modelo pré-estabelecido por meio de uma interface.

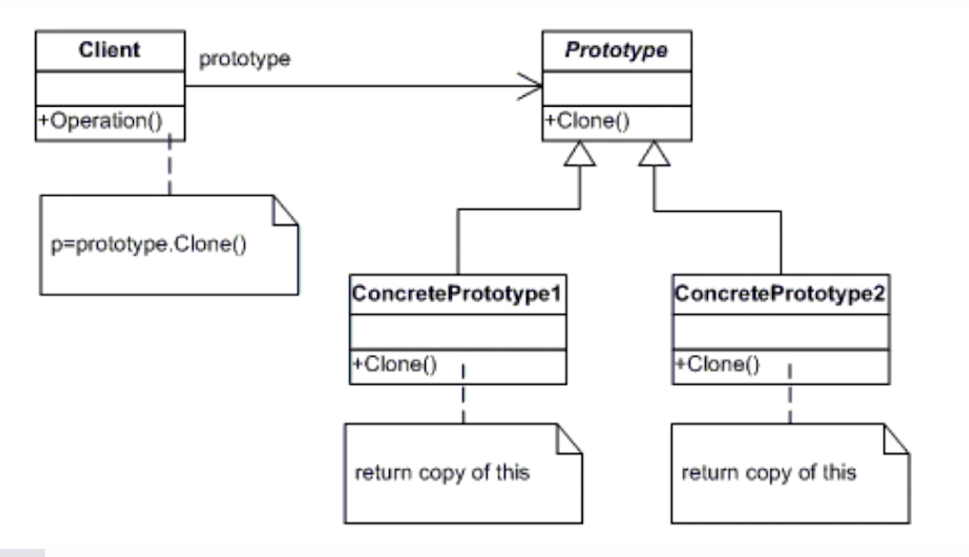


**Prototype** –

Objetivo – Especificar o tipo de objetos a criar usando uma instancia prototípica, e criar novos objetos copiando esse protótipo.

Características – 1° Uma cópia exata carrega consigo espécie de “fotografia” do objeto: não apenas seu comportamento é “clonado”, mas também seu estado, ou seja, um novo objeto com os mesmos campos do objeto original. 2° Entretanto, quando se quer copiar um objeto, quase sempre se deseja um “novo objeto” e não um “objeto novo”, o que significa uma copia não instanciada daquele objeto, ou com os campos ajustados para valores iniciais, que sejam default para toda aplicação. 3° Como desvantagem, o programador deve ter em mente esta diferença ao implementar este padrão, pois objetos clonados podem apresentar problemas de inclusão de classes não existentes e/ou referencias (dependências de parâmetros ou resultados que sejam internos aos objetos e que não sejam levados em consideração na clonagem).

Um exemplo de situação para imaginamos a utilização desse factory é uma situação de confirmação de alteração. No caso efetuamos a alteração de alguns dados de um usuário, no momento de salvar é exibido o antes e o depois da alteração. Dessa forma, antes do objeto sofre uma alteração foi gerado um clone do mesmo, para guardar o seus estado.



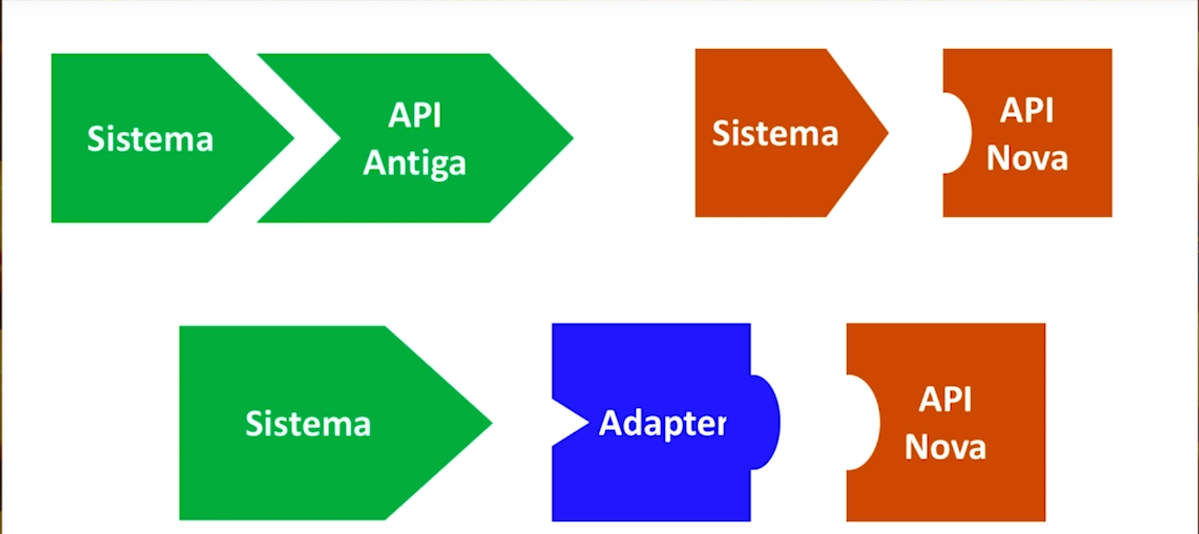
**Estrutural**

* Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores.
* Os de classes utilizam a herança para compor interfaces ou implementações, e os de objeto ao invés de compor interfaces ou implementações, eles descrevem maneiras de compor objetos para obter novas funcionalidades.
* A flexibilidade obtida pela composição de objetos provem da capacidade de mudar a composição em tempo de execução o que não é possível com a composição estática (herança de classes).

**Adapter -**

Objetivo – Tem como proposito ajustar interfaces para que sejam compatíveis entre si. Convertendo a interface de uma classe em outra interface que os clientes esperam. O adaptador permite que as classes trabalhem juntas, o que não poderia ocorrer devido a interfaces incompatíveis.

Características – 1° Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. 2° Padrão particularmente útil para fazer bibliotecas de classes desenvolvidas independentemente trabalharem juntas. 3° Cria uma classe que faça uma adaptação. 4° Adapter faz com que uma interface adaptada seja compatível com outra. 5° Adapter de classes e Adapter de objetos.



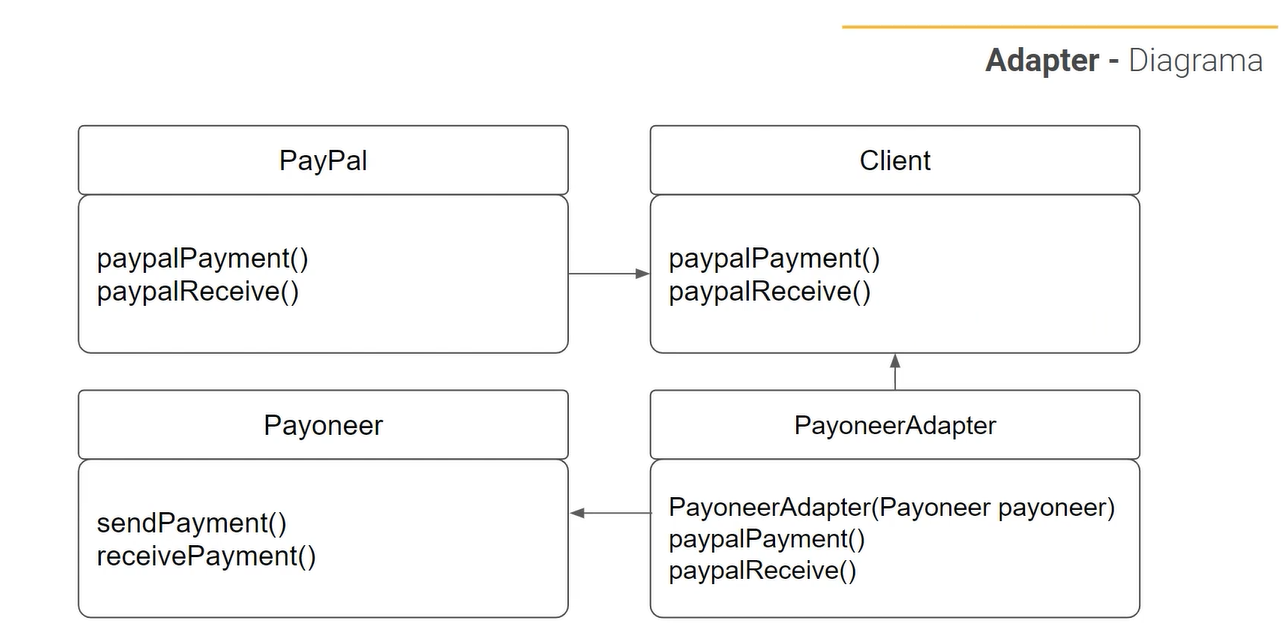
Vantagens – Adapter de Classe: Permite que o adaptador sobrescreva algumas funções do adaptado. Adapter de Objeto: Permite o uso de um único adaptador para uma hierarquia de classes adaptadas.

Desvantagens – Adapter de Classe: Não funciona bem quando se quer adaptar uma hierarquia de classes. Adapter de Objeto: E mais difícil sobrescrever funções do adaptado.

Para entender o Adapter iremos utilizar o seguinte exemplo, digamos que teremos que integrar um sistema de pagamentos novo no sistema. Anteriormente o sistema já era integrado com o PayPal, e agora iremos implementar o Payonner, porém o problema é que os nomes dos métodos são diferentes, e também tem estruturas diferentes. Agora precisamos adaptar o nosso cliente para consumir tanto do PayPal quanto do Payoneer, só que não pode quebrar o que já está funcionando.

Podemos utilizar o padrão ADAPTER para converter a interface de um objeto de maneira que outro objeto possa compreender as informações. É um adaptador. Um adaptador encobre um dos objetos para esconder as regras de negócio e qualquer dificuldade de conversão, isso fica oculto. O objeto encoberto neste caso o PayPal nem fica sabendo que existe um adapter, o adapter só consome a informação.

Adapter: Converter a interface de uma classe para outra interface que o código cliente espera encontrar. A entidade adaptadora permite que classes com interfaces incompatíveis trabalhem juntas.



**Bridge –**

Objetivo – Desacoplar uma abstração de sua implementação para que os dois possam variar independentemente.

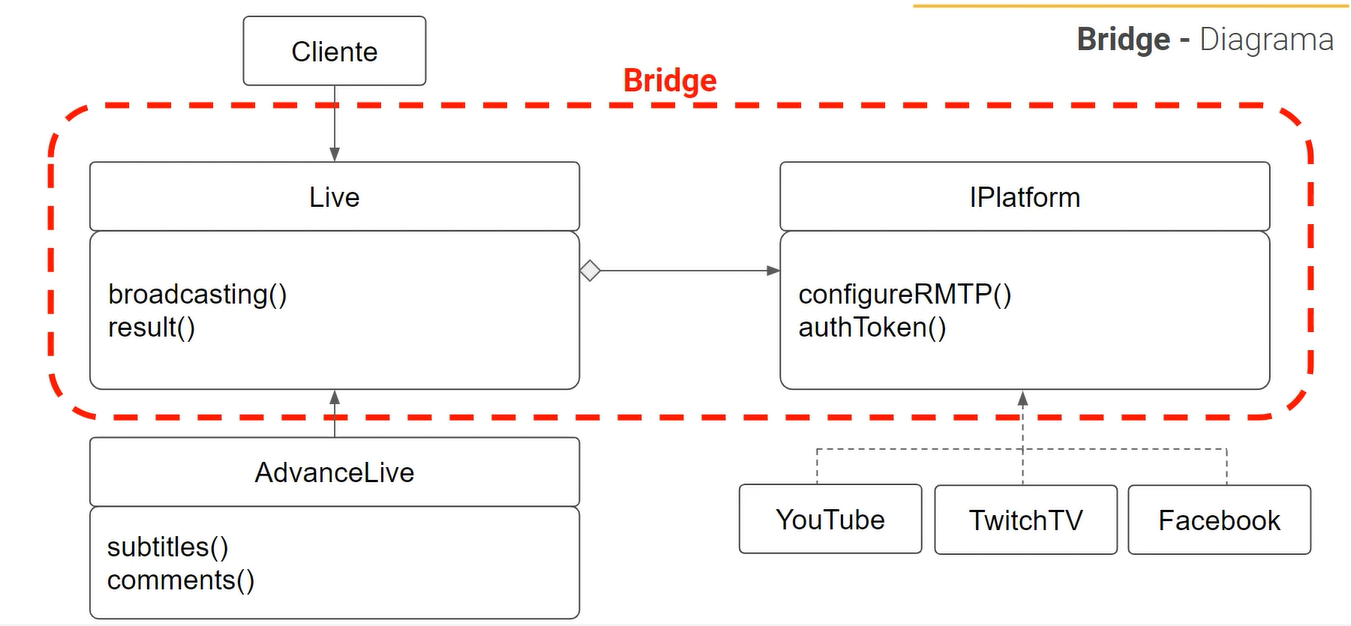
Para entender o Bridge iremos utilizar o seguinte exemplo, será necessário criar um sistema para transmissão de lives, mas não sera transmitido apenas no Youtube, e sim na Twitch e no Facebook. E cada plataforma tem suas próprias especificações. Poderia ser criado uma classe enorme para atender todos os requisitos das três plataformas, porém não é uma boa saída pelo que vimos até agora.

Precisamos quebrar essa classe em várias classes, mas pense bem, Youtube Live, Facabook Live e Twitch Live, ou seja, todas as plataformas utilizam o mesmo conceito. Então como podemos quebrar uma classe grande em várias? E ainda assim compartilhar os recursos da transmissão como configuração RMTP, autenticação e outros métodos necessários para fazer a Live?

O Bridger é um padrão de projeto estrutural que permite você dividir uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas: Abstração e Implementação.

E elas podem ser desenvolvidas independentemente umas das outras. Abstração neste caso não tem a ver diretamente com interface ou classe abstrata.

Abstração é uma camada de controle de alto nível para alguma entidade. Nesse caso sera nossa Live. Essa camada não deve fazer nenhum tipo de trabalho por conta própria. Ela deve delegar o trabalho para a camada de implementação (também chamada de plataforma) – sera Youtube, Twitch e Facebook.



**Comportamental**

**Chain of Responsibility –**

A esse pattern dá a ideia de uma corrente com vários elos interligados, da mesma maneira quando temos uma tarefa que depende de vários passos ou manipuladores, nos podemos interligá-los, fazendo que cada um seja executado de forma independente, assim se um deles tiver algum problema, ou cumprir o objetivo que precisamos não precisamos seguir adiante.

Exemplo Chain of Responsibility(Corrente de responsabilidade) “A aplicação pode tentar autenticar um usuário ao sistema sempre que receber um pedido que contém as credenciais do usuário. Contudo se essas credenciais não estão corretas e a autenticação falha, não há razão para continuar com outras checagens. Com o tempo você implementa mais diversas checagens sequenciais”.

Como muitos outros padrões de projeto comportamental, Chain of Responsibility se baseia na ideai de transformar comportamentos em objetos independentes denominados handlers. Assim as verificações extraídas para sua própria classe com um único método que faz a checagem. O pedido, junto com seus dados, é passado para esse método como um argumento.

O padrão sugere que os handlers sejam ligados como uma corrente. Cada handler ligado tem um campo para armazenar uma referência ao próximo handler da corrente. Além de processar o pedido, os handlers também o passam adiante na corrente. O pedido viaja através da corrente até que todos os handlers tiveram uma chance de processá-lo. E aqui está a melhor parte: um handler pode decidir não passar o pedido adiante na corrente e efetivamente interromper um próximo processamento.

