What is the Factory pattern, and what problem does it solve in software design?	Describe the structure of the Factory pattern. What are the main components involved?	What are the benefits of using the Factory pattern in software development? Provide some practical examples.
The Factory pattern is responsible for producing many copies of the same class / subclasses, allowing for new subclasses of a superclass.	The factory class will contain methods to create new objects, which they will return to the main script. It can be given different responsibilities with some adaptations, like including a list with all the created objects, also becoming a manager of those objects	The factory pattern allows us to create new classes if we see fit, without needing to alter the main script, and adheres to the single responsibility principle. If there is a new class created then all we need to do is add the method to create the class in the Factory, opposed to needing to factor it in the main script
O Factory Pattern é um padrão creacional que esconde a implementação da instanciação de uma nova instância de uma classe	Interface do Produto: comum a todos os objetos que pode ser produzida pelo criador e as suas subclasses; Concrete Products: diferentes implementações da interface do produto; Classe Creator: Declara o método factory que retorna novos objetos produto; Concrete creators: override do método factory base para retornar o tipo de produto;	Evitar high coupling entre o criador e os concrete products; Single responsibility principle -> mover o codigo da criação do produto para um lugar no código tornando-o mais fácil de suportar; Open/Closed Principle -> introduzir novos tipos de produto no programa sem quebrar código de cliente existente.
Factory Pattern é um padrão de projeto de software que permite às classes delegar para subclasses decidirem, isso é feito através da criação de objetos que chamam o método fabrica especificado numa interface e implementado por um classe filha ou implementado numa classe abstrata e opcionalmente sobrescrito por classes derivadas.	Creator — declara o factory method que retorna o objeto da classe Product . Este elemento também pode definir uma implementação básica que retorna um objeto de uma classe ConcreteProduct básica; ConcreteCreator— sobrescreve o factory method e retorna um objeto da classe ConcreteProduct; Product — define uma interface para os objectos criados pelo factory method; ConcreteProduct — uma implementação para a interface Product.	Baixo acoplamento, maior flexibilidade e elimina a necessidade de acoplar classes específicas para aplicação em nível de código. Utilizar para especificar tipos de transporte(barco, carro, autocarro) ou animais(gato, cão)
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.	Product, ConcreteProduct, ConcreteCreator, Creator	The Factory Method separates product construction code from the code that actually uses the product. Therefore it's easier to extend the product construction code independently from the rest of the code. For example, to add a new product type to the app, you'll only need to create a new creator subclass and override the factory method in it.
O factory pattern serve para facilitar a criacao de objetos atraves da criacao de uma inteface.	O factory tem uma interface que sera comum para todos os objetos criados.	O factory method separa a criacao de objetos da classe tornando o codigo mais simples e mais legivel.

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. For example, we don't use "new" commands in main and let other functions do that. This can help making the code more extensible, for example, if we have a project that provides transport to clients (cars) and after sometimes we want to add (motorbikes) it is easier to do it, and we don't need to change our original code	The Product declares the interface, which will be common to all objects that can be produced by the creator and respective subclasses. Concrete Products will be different implementations of the product interface. The Creator class will declare the factory method that will return new Product objects. The return type SHOULD match the product interface. You can declare the factory method as abstract to force all subclasses to implement their own versions of the method. You can also the base factory method can also return some default product type. Despise its name, product creation is not the primary responsibility of the creator. Concrete Creators override the base factory method so it returns a different type of product. The factory method doesn't have to create new instances all the time. It can also return existing objects from a cache, an object pool, or another source	We avoid the tight coupling between the creator and the concrete products. We also follow the Single Responsability Principle, because we can move the product creation code into one place in the program, making it easier to support and the Open/Closed Principle because we can introduce new types of products into the program without messing or breaking existing client code. Some pratical examples would be: Document generation, a "factory" can generate different document formates depending on user selection UI Component Generation, we can create various UI elements like buttons or text boxes.
The Factory pattern provides an interface to write objects in a superclass, allowing their subclasses to change the type of objects that will be created without damaging the program.	The structure consists of: - The Product, a basic interface that has general info about all the objects. - The Concrete Products, more specific implementations of the Product. - The Creator, which implements the interfaces and declares the methods necessary (it is the superclass). - The Concrete Creators, the subclasses of Creator that override the previous method to return different products.	Having a Factory pattern prevents code confusion at long term development, making it way more organized.
O padrão Factory, ou padrão de fábrica, resolve o problema da criação de objetos sem expor a lógica de instanciação ao cliente. Abstrai o processo de criação de objetos, permitindo que o sistema seja independente de como os seus produtos são criados e representados	A estrutura do padrão Factory inclui uma interface para criar um objeto, mas deixa a classe que implementa a interface decidir qual classe instanciar. Os componentes principais são a interface Factory e as classes concretas que implementam essa interface, criando objetos de diferentes tipos.	Os benefícios de usar o padrão Factory incluem a promoção do baixo acoplamento, maior facilidade para adicionar novos tipos de objetos sem alterar o código existente, e a criação de famílias de objetos relacionados sem precisar especificar suas classes concretas. Um exemplo prático é um aplicativo de gestão de documentos que pode usar o padrão Factory para criar diferentes tipos de visualizadores de documentos, dependendo do formato do arquivo, sem que o código cliente saiba qual visualizador está sendo usado.

O padrão Factory é um padrão de design criacional na engenharia de software que fornece uma maneira de criar objetos sem especificar sua classe exata, utilizando uma interface comum ou classe base e permitindo que subclasses alterem o tipo de objetos que serão criados.

Além disso, este padrão resolve ainda o problema de criar objetos sem expor a lógica de instanciação ao cliente. Isto é, o padrão encapsula o processo de criação de objetos e desacopla o código do cliente da implementação real dos objetos. Tudo isto promove a flexibilidade e a manutenibilidade do código, uma vez que permite alterações mais fáceis nos tipos de objetos sendo criados ou adicionados no futuro sem exigir modificações no código do cliente. Além disso, ajuda a aderir ao princípio Aberto/Fechado, permitindo a adição de novos tipos de objetos sem modificar o código existente.

A estrutura do padrão Factory consiste nos seguintes componentes principais:

Product: Define a interface ou classe base para os objetos que serão criados pelo Factory.

ConcreteProduct: Implementações específicas do produto que serão criadas pelo Factory.

Factory: Interface ou classe base que declara um método para criar objetos. Esta é a parte central do padrão, onde a lógica de criação é encapsulada.

ConcreteFactory: Implementação da fábrica que instancia e retorna objetos ConcreteProduct.

Em resumo, o padrão Factory envolve a criação de uma fábrica que produz objetos conforme uma interface comum ou classe base, permitindo que diferentes implementações sejam criadas sem expor a lógica de criação ao cliente.

Os benefícios do uso do padrão Factory no desenvolvimento de software incluem:

Encapsulamento da lógica de criação: A lógica de criação de objetos é encapsulada na fábrica, o que ajuda a ocultar os detalhes de implementação dos clientes.

Desacoplamento entre cliente e classes concretas: Os clientes não precisam conhecer as classes concretas dos objetos que estão sendo criados, apenas a interface ou classe base.

Flexibilidade e extensibilidade: É fácil adicionar novas implementações de produtos ou modificar a lógica de criação sem alterar o código do cliente.

Promoção da consistência: O padrão Factory ajuda a garantir que os objetos sejam criados de maneira consistente em todo o sistema.

Exemplos práticos incluem:

-Uma fábrica de carros que produz diferentes modelos de carros (ex: Sedan, SUV, Hatchback) com base nas preferências do cliente.

O padrão Factory, ou padrão de fábrica, resolve o problema da criação de objetos sem expor a lógica de instanciação ao cliente. Abstrai o processo de criação de objetos, permitindo que o sistema seja independente de como os seus produtos são criados e representados

A estrutura do padrão Factory inclui uma interface para criar um objeto, mas deixa a classe que implementa a interface decidir qual classe instanciar. Os componentes principais são a interface Factory e as classes concretas que implementam essa interface, criando objetos de diferentes tipos.

Os benefícios de usar o padrão Factory incluem a promoção do baixo acoplamento, maior facilidade para adicionar novos tipos de objetos sem alterar o código existente, e a criação de famílias de objetos relacionados sem precisar especificar suas classes concretas. Um exemplo prático é um aplicativo de gestão de documentos que pode usar o padrão Factory para criar diferentes tipos de visualizadores de documentos, dependendo do formato do arquivo, sem que o código cliente saiba qual visualizador está sendo usado.

	Product Interface/Abstract Class: This defines the common interface or abstract class that all the concrete products (the objects being created) will implement or inherit from. It specifies the operations that all products must support. Concrete Products: These are the actual classes that implement the Product interface and provide the specific functionality for each type of object that can be created. Creator/Factory Class: This class defines the factory method that clients will use to request object creation. It can be an interface or an abstract class that subclasses inherit from. The factory method usually takes some input (like a type parameter) to determine which concrete product to create. Subclasses will typically override the factory method to provide logic for creating their specific concrete product.	You avoid tight coupling between the creator and the concrete products. Single Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support. Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code.
Factory pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.	Client: This is the part of your code that uses the factory to create objects. It interacts with the factory class through the factory method, requesting objects of the desired type without needing to	O código fica mais limpo e organzado, segue o Single Responsability principle, reduz o acouplamento e aumenta a coesão, também ajuda a manter o código a seguir o Open/Closed principle. No exercicío dos voos de PDS, para adicionar um novo voo, pode-se criar uma factory que receba os argumentos do voo e do avião e que retorne um novo voo.
O factory pattern é uma maneira de instanciar objetos sem ter de especificar as classes exatas. Recebe argumentos e cria um novo objeto consuante. Evita high coupling	Tem de haver uma interface dos métodos dos objetos que este produz, e tem de dar return de um new object	
The factory pattern allows creating specialized subclasses from the superclass without leaking the concrete type. This allows for easy specialization of the object depending on the arguments passed to the factory.	The factory consists of two or more classes, first the superclass which has the object interface implemented by it's subclasses and the factory method (the factory method can also be hijacked into it's own class), then we have the concrete implementations that are subtypes of the superclass.	The factory pattern allows for abstracting differences of implementations on different environments, for example we could have a factory method that depending on the OS it's running returns the native class responsible for handling user input without coupling the rest of the application to it.

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Imagine you have a base class called Character, and you want subclasses to create specific types of characters, such as Warrior, Mage, and Archer.
In a traditional approach, you might directly create instances of these subclasses in your code.
However what if you want to allow for flexibility in creating these characters?
This is where the Factory Method pattern comes in. Instead of directly creating instances of subclasses, you define a factory method within the Character class or interface.
Each subclass then implements this factory method to create its own instances.

The Factory pattern is a creational design pattern that provides an interface for creating objects in a super class but allows subclasses to alter the type of objects that will be created.
The main components of the Factory pattern include: Product, ConcreteProduct, Creator and ConcreteCreator.

Some benefits of using the Factory pattern in software development: Flexibility and Extensibility, Encapsulation, Reduced Code Duplication, Abstraction, Centralized Control and Consistency.

Some practical examples:
Database Connection Factory: A factory that creates database connection objects based on the type of database being used.

GUI Widget Factory: A factory that creates different types of GUI widgets (like buttons, text fields) based on user input or application configuration.

Document Generator Factory: A factory that creates document generator objects for generating documents in various formats.

By employing the factory pattern in these scenarios, developers can achieve better code organization, maintainability, and flexibility in managing object creation.

É a criação de uma entidade responsável pela instanciação de objetos. Resolve o problema da standarização da criação de objetos

Metodo estático de uma classe que retorna um objeto dessa mesma classe. Ela deve chamar o construtor da classe ou da subclassse indicada e retorna-la de modo a estar pronta a ser usada.
Componentes:
- interface comum aos objetos que se pretende criar.
- a implementação desses objetos
- Uma classe creator que está encarregue de instanciar esses objetos
- Os metodos da classe creator que retornam os objetos em si

Abstração do processo de criação de objetos (Single Responsibility Principle)
Podes introduzir novos tipos de objetos com o esforço minimo (Open/Close Principle)
Evita um coupling alto entre o criador e o objeto criado

The Factory pattern, specifically the Factory Method pattern, is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
This pattern is used to deal with the problem of creating objects without having to specify the exact class of the object that will be created. It achieves this by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor

The factory pattern says that class creation and instantiation can and should be handled by different classes.

Factory pattern não especifica uma classe de um objeto. define uma interface para criar um objeto mas deixa que as subclasses.
Os problemas são acoplamento rigido, subclasses complexas, codicoes de criacao complexas.
Algumas solucoes sao encapsulamento da criacao, flexibilidade e a reutilizacao, descoplamento.

The Factory pattern, specifically the Factory Method pattern, is structured around several key components:

Product: This is the interface or abstract class that declares the common interface for all objects that can be produced by the creator and its subclasses. It defines the methods that make sense for every product.

Concrete Products: These are the different implementations of the product interface. Each concrete product represents a specific type of product that can be created by the factory method.

Creator: This is the class that declares the factory method. The factory method is responsible for creating and returning new product objects. The return type of this method matches the product interface. The creator class can be abstract, forcing all subclasses to implement their own versions of the factory method, or it can be a concrete class with a default factory method.

Concrete Creators: These are subclasses of the creator class that override the base factory method to

Creator - creates the class, calls Creator 1 to instantiate it
Creator 1 - instantiates the class
Product - The class
Product 1 - An instance of Product, created by Creator 1

A estrutura concentra em o utilizo do metodo em uma classe para ccriar objeto em vez de um construtor.

Reduced Coupling and Improved Cohesion: By decoupling the creation process from the rest of the code, the Factory pattern allows each class or method to have a single responsibility and purpose. This separation leads to more maintainable and scalable code

Promotes Reusability and Extensibility: The Factory pattern provides a consistent interface or abstract class for different implementations or variations of an object, reducing code duplication and making the code more extensible. This means that adding new types of objects or changing existing ones becomes easier without affecting the rest of the application

Increased Flexibility and Scalability: It allows for the modification of the behavior or appearance of objects by changing the parameters or input of the factory method or class. This flexibility enables switching between different subclasses or implementations without altering the code that uses the object, thereby improving the scalability of the application

Practical Examples:
Creating Shapes in a Drawing Application: Based on the name of the shape and some dimensions, a factory method or

It reduces the need for duplicated code when creating the class.

As principais componentes são a separacao entre a criacao e utilizacao de objetos, pricipio de aberto/fecho e tembem o descoplamento.
Por exemplo um pagamento de mbway:
pagamento de produto: temos a interface para os objetos.
produtos concretos

Consiste em definir uma interface para criar um objeto numa superclasse, permitindo às subclasses alterar o tipo de objeto a ser criado.

É um padrão de Design criacional que permite a criação de objetos sem que seja necessário especificar a classe do objeto que está sendo criado. Assim, não é necessário utilizar um construtor, mas sim um método que retorna o objeto já criado. O Factory Pattern permite o desacoplamento entre os componentes do código.

The Factory Pattern is an Interface used for creating objects in a superclass, but allows subclasses to alter the type of object that will be created.
When the creation of the object is incorporated in the class it makes the code hard to read and not maintainable.

Esconder os instanciadores dos construtores (MyClass ob = new MyClass();) da main(), recorrendo a "wrappers" ou a funções similares.
O problema principal que este resolve é a inflexibilidade da criação de objetos.

Uma interface que define as operações que todos os tipos diferentes de objetos podem executar.

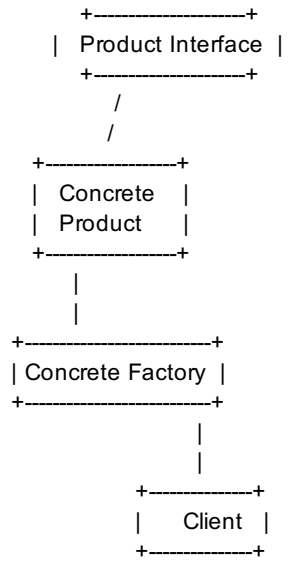
Classes Concretas que são as classes que implementam a interface e definem a funcionalidade específica de cada produto.

Classe Fábrica é responsável por criar objetos. Ela usa um método de criação para decidir qual classe concreta instanciar com base na entrada que recebe.

Consiste em uma interface,um criador, que implementa essa interface, um produto abstrato e um produto concreto.

The structure of this pattern is all about an interface for a product and a creator that is applied to the Factory and the Product itself. This needs a Client as well that relies on the Factory to create the Products.

Product Interface or Abstract Class;
Concrete Products; Concrete Factory:



Permite a reutilização de objetos já criados, ter nomes mais descritivos para o métodos do que os construtores. É possível adicionar novos métodos para novos produtos sem alterações para o cliente.

Exemplo:
Num Viveiro, para criar uma Árvore, podemos ter um método de fábrica que aceite como argumento uma string e devolva um objeto do tipo Árvore, mas de uma subclasse. Assim, o programa principal fica liberto de fazer new, limitando-se a invocar o método estático no Viveiro.

Uma classe pode criar um objeto sem a necessidade de saber qual objeto está a criar, permitindo o desacoplamento e reutilização de código.
Imagine um caso em que seja necessário criar uma classe "cadeira" com diferentes especificações, cria-se uma interface cadeira e a partir dessa interface, é possível criar diferentes tipos de cadeira sem alterar o código existente.

Some of the beneficts are respecting the Single Resposability Principle and the Open Closed Principle. It helps make the code more clean and easy to read.

We could have used this principle in the last classes exercise of the Voos.

Permite a flexibilidade na criação e instânciação de objetos.

MyClass ob = new MyClass();

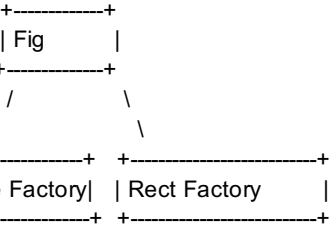


Fig ob = circle();
Fig ob = rect();

		The factory design pattern provides various benefits, among them being providing an approach to code for interface rather than implementation, removing the instantiation of the actual implementation classes from client code, making more robust, less coupled and easy to extend. An example of this is easily change PC class implementation, due to the client's program being unaware of this. Finally, the factory design pattern provides an abstraction between the implementation and client classes through inheritance
Factory pattern is used when we have a superclass with multiple sub-classes, and we want to return a specific sub-class based on the input. This way, it takes away the responsibility of the instantiation of a class from the client program to the factory class	There are four main components. The interface Product, Concrete Product(s), a Creator class, and Concrete Creator(s)	
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created, it tries to solve: How can an object be created so that subclasses can redefine its subsequent and distinct implementation? How can an object's instantiation be deferred to a subclass?	Product: This is an interface for objects that the factory method creates. ConcreteProduct: These are classes that implement the Product interface. Creator: This is an abstract class that declares the factory method, which returns an object of type Product. ConcreteCreator: These are subclasses of Creator that override the factory method to return an instance of a ConcreteProduct.	reduces coupling and improves cohesion by decoupling the creation process from the rest of the code, allowing each class or method to have a single responsibility and purpose.
The factory pattern make sit easier to create objects by taking that responsibility away from the main class and instead assigning it to a dedicated class.	The main components are the interface, that creates a base for all the objects, the concrete class that creates the individual object itself and the Factory, that manages which object must be created.	By using the Factory design pattern, you can easily modify your code to suit new needs. If you need a new object for a new purpose you only need to add that object and slightly change the object Factory. That way the creation logic doesn't need to be explicit to the user.
It is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. Objects returned by a factory method are often referred to as products.	The main components involved in the Factory pattern are: 1 - Creator -> declares the factory method that returns new product objects 2 - Concrete creators -> override the base factory method so it returns a different type of product 3 - Product -> declares the interface which is common to all objects 4 - Concrete product -> different implementations of the product interface	Now we have a code that only creates circle objects but now we want to expand the code to other figures. It is a better solution to define a superclass called Figure. We can also define an interface that all products must follow (ex: calculateArea(), calculatePerimeter()).

A factory method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
This pattern solves the problem of creating objects without specifying the exact class of the object that will be created.

Structure:
Client: The client is the code that needs to create objects but doesn't want to specify the exact class of the object.
Factory interface: This defines the interface or abstract class for creating objects.
Concrete factories: These are the classes that implement the factory interface or extend the abstract class, each concrete factory is responsible for creating instances of a specific type of object.
Product: The product is the object being created.

You use the Factory pattern when you're not sure beforehand the exact types and dependencies of the objects your code should work with.
The Factory Method separates product construction code from the code that actually uses the product. So it's easier to extend the product construction code independently from the rest of the code.
Inheritance is easy to be extended. Easy to extend internal components.
Example: Implementation of interfaces onto subclasses because it provides versatility by allowing you to instantiate concrete objects types through a more abstract concept.

O padrão Factory é uma ferramenta valiosa para o desenvolvimento de software no sentido em que oferece flexibilidade e simplificação do código.
Este padrão visa a resolução de problemas relacionados à criação de objetos em Java.

A estrutura tem os seguintes elementos:
- Objeto -> Interface que contém os métodos implementados pelos objetos concretos
- Objeto Concreto -> Especifica as funcionalidades de cada objeto
- Criador -> Define o método de criação do tipo do produto
- Criador Concreto -> Através da interface do Criador cria objetos concretos

Alguns do benefícios são:
- Eliminar a necessidade de usar o operador new
- Torna o código mais flexível, legível e mais fácil de manter
- Facilita a criação de objetos de acordo com o seu contexto ou ambiente de execução
- Permite criação de código low coupling e permite-nos usar o principio Open/Closed de padrão SOLID.

É um padrão Creational que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.
O problema que resolve é o seguinte: Como um objeto pode ser criado para que as subclasses possam redefinir sua implementação subsequente e distinta? Como a instanciação de um objeto pode ser adiada para uma subclasse?

Product - declara a interface, que é comum em todos os objetos que podem ser produzidos pelo criador e pelas suas subclasses.
Concrete Products - são diferentes implementações da interface do produto.
Creator - declara o método de fábrica abstrato que retorna novos objetos de produto. É importante que o tipo de retorno desse método corresponda à interface do produto.
Concrete Creators - São subclasses de produtos que implementam o método de fábrica para criar objetos de produtos específicos

Evita acoplamentos entre o criador e os produtos concretos.
Princípio de responsabilidade única- é possível mover o código da criação do produto para um único local do programa, facilitando a manutenção do código.
Princípio aberto/fechado- podemos introduzir novos tipos de produtos no programa sem modificar o código cliente existente.

Exemplo:
1. No caso de ser importante economizar recursos do sistema reutilizando objetos existentes em vez de recriá-los sempre.
2. Quando não se sabe ao certo os tipos e dependências exatas dos objetos com os quais o codigo vai funcionar.

The Factory pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. In other words, it provides a way to delegate the instantiation logic to subclasses, rather than having it defined in the base class.
The problem that the Factory pattern addresses is tightly coupled code. In traditional object creation scenarios, client code directly instantiates concrete classes, meaning it's tightly bound to those specific implementations. This tight coupling can lead to various issues. For instance, if the concrete classes change or if new classes need to be added, every place where instantiation occurs must be modified. Furthermore, if there are multiple places in the codebase where objects are created, any changes to the instantiation logic would need to be replicated across all those locations, leading to redundancy and potential inconsistencies.
By introducing the Factory pattern, the client code no longer needs to concern itself with the concrete classes of the objects it creates.

The main components related to the factory pattern are the product, the concrete products, the creator and the concrete creator. The product is responsible for creating an interface that is common to all objects that can be produced by the creator class and their subclasses. From this interface we can create concrete products, that implement the product interface. The creator returns new product objects, that have to match the product interface, if the factory method is declared as abstracts each of the subclasses have to implement their own version of the method. As an alternative, the base factory method can return some sort of default product type. Concrete Creators override the base factory method so it returns a different type of product.

Using the Factory pattern in software development provides significant benefits. It decouples client code from concrete implementations, enhancing maintainability and flexibility. By working with abstract interfaces or base classes, the pattern promotes abstraction and facilitates higher-level design. Factories offer flexibility by dynamically determining object instantiation based on runtime conditions. Additionally, encapsulating common object creation logic promotes code reuse across different parts of the application, contributing to better organization and efficiency in software development.
The Factory pattern proves beneficial in numerous practical scenarios, such as database connection management, GUI component creation, logging frameworks, dependency injection containers, and plugin systems. For instance, in a database access layer, a factory can dynamically create database connections for various vendors without altering client code, ensuring flexibility and maintainability. Similarly, in a GUI framework, different implementations of a factory can generate UI components tailored for different platforms or styles, facilitating platform-agnostic development. Whether managing dependencies in a dependency injection container or

O que é: É um padrão criacional que providencia uma interface para criar objetos numa super classe, permitindo que as superclasses alterem os objetos que vão ser instanciados.

Problema:A manutenção ou alterações na estrutura do código podem ser muito custosas, pois por exemplo se tivermos uma aplicação de transportes em que apenas trabalha com camiões, e posteriormente passar também a trabalhar com carros, em termos de implementação, teremos de alterar o código em muitos sítios, de modo a conseguir integrar esta alteração, nomeadamente, poder instanciar objetos do tipo carro, para além do tipo camião.

Substituir a instanciação de objetos usando o operador "new" diretamente, com chamadas a um método especial da factory.
Componentes: Ter uma superclasse ou interface com um método de criação que vai ser sobrescrito nas subclasses e aí sim, instancia um objeto do tipo determinado usando o operador "new".

Simplifica a integração de novas subclasses num projeto já existente.
O princípio do Open/Closed está presente, uma vez que a classe está aberta para extensão e fechada para alterações.
É evitado o alto acoplamento entre o criador e os objetos concretos.
O princípio do Single Responsability, uma vez que dá para mover o código responsável pela criação de objetos para um espaço específico do programa, permitindo que a codebase seja mais fácil de manter.

O Factory Pattern resolve um forma de criar objetos sem ter que especificar a classe exata do objeto que será criado.

o Factory pattern consiste em quatro componentes principais um produto, produto concreto, criador, criadores concretos, a principal componentes e o produto porque e constituído por uma interface ou classe abstrata que define o objeto a ser criado.

um beneficio de usar o padrão factory e que torna o projeto mais simples

O padrão Factory é um padrão de design que fornece uma interface para a criação de objetos. Este padrão resolve a questão da criação de objetos sem juntar fortemente o código do cliente às classes concretas.

Estrutura do padrao Factory:
- Product -> Representa o comportamento e os atributos comuns que todos os produtos concretos devem implementar;
- Concrete Products -> Contém as implementações específicas da interface/classe Product;
- Factory -> responsável por instanciar o produtor concreto apropriado com base em certas condições ou parâmetros;
- Client -> solicita a criação de um objeto da Factory chamando o método Factory.

Benefícios do padrão Factory:
- aumenta a flexibilidade;
- aumenta a extensibilidade;
- aumenta a capacidade de manutenção do código.

Exemplo pratico:
- no acesso aos bancos de dados: ou seja, o código do cliente pode então usar o objeto do banco de dados criado sem a necessidade de conhecer os detalhes específicos da implementação do mesmo.

O padrão Factory é um padrão de desenho criacional que abstrai o processo de criação de objetos, permitindo que suas subclasses decidam quais objetos criar. Ele resolve o problema da criação direta de objetos em código, o que pode levar a dependências rígidas e dificuldades na manutenção e expansão do código.

O factory pattern tem como suas componentes o Product (que declara a interface comum a todos os objetos), o Concrete Product (que são as varias implementações da interface Product), o Creator (retorna um objeto de uma classe Product, pode ser abstrato ou ter uma implementação padrão) e o Concrete creators(subscreve o método fábrica para retornar uma instância de um ConcreteProduct específico)

Evita o acoplamento e simplifica a criação de objetos.
Exemplos:
Um Factory pode criar conexões específicas para diferentes tipos de bancos de dados (MySQL, PostgreSQL, etc.) sem alterar o código que utiliza essas conexões.
Um Factory pode servir para o desenvolvimento de jogos pois serve para criar diferentes tipos de personagens com base em condições ou configurações, facilitando a expansão do jogo.

The Factory pattern, in the context of software design, refers to a creational design pattern that defines an interface for creating an object but lets subclasses decide which class to instantiate. This pattern falls under the category of class-creation patterns and object-creation patterns, depending on whether the class instantiation logic is deferred to subclasses or encapsulated within a single function.

In software design, direct instantiation of objects often leads to tight coupling between classes, making the system less flexible and more difficult to extend or modify. When a class is directly dependent on a specific object, any change in the construction process of that object or its class hierarchy requires changes in the class that uses it. This violates the principles of modular, scalable, and maintainable code.

The Factory pattern solves these problems by:

Encapsulating Object Creation: By moving the creation logic into a separate method or class, the system

The main components involved in the Factory Method pattern are:

Product:
The Product component represents the interface or an abstract class used to define the contract for the types of objects the factory method will create. Any concrete product will adhere to this contract, ensuring that the products are used interchangeably within the system that expects an object conforming to the Product interface.

Concrete Product:
Concrete Products are specific implementations of the Product interface. Each Concrete Product corresponds to a specific kind of product that the factory can produce. These are the objects actually constructed by the factory method. Each one has its unique implementation details, but they all conform to the Product interface.

Creator:
The Creator is an abstract class or an interface that declares the factory method. This method is what creates the Product objects. Often, the Creator class contains some standard business

It promotes low coupling between classes because it separates the construction of objects from the objects' class hierarchy. It also promotes flexibility and scalability because it encapsulates object creation. new types of objects without changing the client code, making the application more flexible and easier to scale. Also, it also simplifies the creation of subclasses since changes to object creation can be made extending the factory class and overriding the creation method, without having to change the client code itself. Finally, since creation is defined with the single purpose of creating classes, it follows the singles responsibility principle, resulting in a easier to maintain system.

Examples:

Document creation -> The implementation of a Factory pattern for handling the creation of documents (Word, Excel, PDF, etc)

Login / Sign up method -> The implementation of a Factory pattern would be useful for handling the type of Login / Sign up the user chooses (gmail, outlook, etc)

Use the factory method when you want to save system resources by reusing existing objects, instead of rebuilding them each time.
You avoid tight coupling between the creator and the concrete products.
Single Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support.
Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code.

Provides an interface for creating objects in a superclass, but allows sub classes to alter the type of classes that will be created

There is an interface class that refers to a simple version of an object, is common to all objects and can be produced by the creator and it's subclasses.

Diminui o acoplamento entre a classe criadora e o produto, aplica o Single Responsibility Principle e o Open/Close Principle.
A classe calendar do Java, para criarmos um novo calendar usamos um metodo estatico da classe para o criar, escondendo a criação do mesmo.

Resolve essencialmente o problema de escalabilidade.

Creator, Concrete Creators e Product

O Factory Design Pattern foca-se em criacao de objetos. Permite uma camada de abstracao entre o usuário e a classe criadora de objetos. Permite tambem ao usuario criar classes expecificas sem que este se tenha de preocupar com isto.
Os pricipais componentes sao:

Através de Factory Design Patterns é possivel obter abstraçao, introduz uma camada entre o código do cliente e a implementação real da criação do objeto. Ao fazer isso, permite alterar ou adicionar novos tipos de objetos sem modificar o código do cliente. Essa abstração promove flexibilidade e modularidade.

Flexibilidade, é possivel criar objetos de tipos diferentes atraves de condicoes ou parametros passados ao construtor.

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Produto (classe abstrata ou interface):
Representa as operações comuns que a fábrica criará.
Define a interface para classes de produtos concretas.
Produtos concretos implementam essa interface.
Por exemplo, em um sistema de pedido de pizza, Pizza poderia ser a classe ou interface abstrata que representa diferentes tipos de pizza.

Reusabilidade, é possivel criar objetos de modo reusavel e modular, permitindo melhor manutencao e uso em difrentes projetos.

The Factory Method pattern solves the problem of creating objects without specifying the exact class of object that will be created. By providing a separate method for object creation, the Factory Method pattern allows for more flexibility and extensibility in the software design. It also helps to decouple the client code from the concrete implementations of the objects, making it easier to adapt to changing requirements or to switch between different implementations without affecting the client code.

Produtos de concreto:

Classes reais que implementam a interface do produto.
Cada produto concreto representa um tipo específico de objeto a ser criado.
No nosso exemplo da pizza, MargheritaPizza, PepperoniPizza e MeatLovers são produtos concretos.

É comummente usado em sistemas de Logging (file loggers, database loggers, and console loggers), Componetes de interface grafica, conexoes com databases (permite diferentes tipos de conecao) e em processamento de pagamentos.

Aqui esta um scketch de um exemplo:

Por exemplo uma Pizza Factory, aplicamos o Factory Design Pattern. O padrão introduz uma camada de abstração entre o código do cliente e a criação do objeto. Classes concretas de pizza (como

The Factory Method is a software design pattern that provides an approach to object creation without directly specifying their concrete classes. Instead, it defines an interface or abstract class for object creation and delegates the responsibility of instantiating objects to subclasses that implement this factory method. This allows client code to work with the object creation interface, decoupling it from concrete implementations.

This pattern resolves several issues in software design, including excessive coupling between client code and concrete classes, thereby facilitating code maintenance and extensibility. Additionally, it promotes code reuse since client classes can interact with the object creation interface without needing to know specific details about the concrete implementation of objects. Thus, the Factory Method enables the creation of more flexible, scalable, and easily maintainable systems.

É um padrão de projeto de software que permite a classes delegar para subclasses decidirem, isso é feito através da criação de objetos que chamam o método fabrica especificado numa interface e implementado por um classe filha ou implementado numa classe abstrata e opcionalmente sobrescrito por classes derivadas.
É utilizado para resolver problemas recorrentes com design flexível, reutilizável e object-oriented software, ou seja, os objetos são mais facilmente implementados, alterados, testados e reutilizados.

The Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. The main components involving the structure of the Factory Pattern are:

Product: This is the interface or abstract class that defines the type of objects that the factory method will produce.

Concrete Product: These are the actual implementations of the Product interface or class.

Factory: This is an interface or abstract class responsible for defining the method(s) to create objects. It can also contain common logic for object creation.

Concrete Factory: These are the subclasses of the Factory class that implement the factory method(s) to create specific instances of products.

Factory Pattern contem 4 elementos:
Creator: declara o factory method que retoma o objeto da classe product
ConcreteCreator: sobrescreve o factory method e retorna um objeto da classe ConcreteProduct
Product: define uma interface para os objectos criados pelo factory method
ConcreteProduct: uma implementação para a interface Product

The Factory Method pattern provides a way to create objects without tightly coupling client code to specific implementations. It centralizes object creation logic, promoting code reuse and allowing for easy addition of new object types without modifying existing code. This pattern enhances flexibility in scenarios like cross-platform development and testing, where different implementations or mock objects are required. Overall, the Factory Method pattern improves code organization and simplifies the process of extending object creation within software systems.

Practical examples:
Game Development: In a video game, the Factory Method pattern can be used to create different types of game characters (e.g., warriors, mages, archers) based on player choices or game scenarios. This allows for easy addition of new character types without modifying existing game logic.

Web Application Development: In a web application, the Factory Method pattern can be employed to create instances of database access objects (e.g., SQL database service, NoSQL database service) based on configuration settings or

Factory pattern oferece vários beneficios, aumenta a coesão e reduz o acopolamento.
Segue ainda o Single Responsibility Principle pois separa o construction logic do business da aplicação.
Um exemplo seria as seguintes classes:

(interface) Veículo
Carro (normal)
Bicicleta
SUV
ClasseDeEscolha
ClasseDeFabrico

Neste exemplo, ClasseDeEscolha, devolve o tipo de carro sem o "código cliente" saber das classes referentes a cada tipo de carro. Apenas é necessário conhecer a ClasseDeEscolha e
da interface Veículo.

É um creational design pattern que providencia uma interface para criar objetos em uma superclasse, mas permite que subclasses alterem o tipo de objetos que serão criados. Resolve o problema de acoplamento rígido entre o código que usa objetos e as classes desses objetos.

É um padrão de desenho do tipo creational que dá uma interface para criar objetos numa surperclasse, mas permite sublcasses alterar o tipo de objeto que é criado. Substitui a chamada de construtores de objetos por metodos de construção especiais deste padrão.

O Factory pattern é um padrão de design de software que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados. No entanto este padrão é usado quando um sistema precisa ser independente de como seus objetos são criados, compostos e representados. O Factory pattern é uma forma de implementar o princípio da inversão de dependência, que diz que as entidades devem depender de abstrações, não de concreções no seu todo.

Product: interface que define operações comum em todos os objetos que podem ser criados pela fábrica.
Concrete Products: as próprias classes que herdam do Product e implementa seus métodos.
Creator: interface que declara o método fábrica.
Concrete creators: são subclasses da classe Creator que subscreve os métodos da fábrica para especificar o produto específico que eles criam.

Uma interface pai que força a implementação de um método construtor e metodos que fazem sentidos a todas as subclasses de novos objetos dessa classe, dos quais as subclasses relevantes irão implementar.
Um criador que irá utilizar esse método para criar novos objetos dessa interface não importando a estrutura de cada um das subclasses especificas

A estrutura do Factory Pattern é composta por quatro componentes principais:

1- Product: Uma interface ou classe abstrata que define o tipo de objetos que a fábrica pode criar.
2- ConcreteProduct: Implementações concretas da interface ou classe abstrata do Product.
3- Factory: Uma interface ou classe abstrata que define o método para criar um objeto do tipo Product.
3- ConcreteFactory: Implementações concretas da interface ou classe abstrata da Factory que criam e retornam instâncias de ConcreteProduct.

Providencia uma forma de criar objetos sem diretamente especificar a classe do objeto que está sendo criado.
Imagine que vc cria uma classe caminhão e a sua classe passa a ser utilizado por muitas pessoas. Caso alguém queira fazer uma classe barco, haveria a necessidade de fazer alteração total da classe caminhão. A utilização do Factory pattern sugere a criação de uma superclasse transporte, que teria as formas de transporte como filhos.

O "Factory pattern" esconde o código do construtor do código do produto. Ou seja, torna mais fácil estender a construção do código independentemente do resto do código. Por exemplo, se estamos a adicionar um novo tipo de produto a uma app, nos só precisamos de criar uma nova subclasse criadora e dar overwrite ao método de fabrica da classe mãe.

O Factory Pattern oferece vários benefícios no desenvolvimento de software, especialmente em sistemas complexos onde a criação de objetos de diferentes classes é necessária. Vou aqui mencionar alguns dos principais benefícios:

1- Abstração da Criação de Objetos: O Factory Pattern abstrai o processo de criação de objetos, permitindo que o código cliente não precise saber os detalhes de como os objetos são criados. Isso torna o código mais limpo e mais fácil de manter.

2- Flexibilidade: Permite que o sistema seja flexível em relação às classes de objetos que podem ser criadas. Isso é especialmente útil em sistemas que podem precisar criar diferentes tipos de objetos em diferentes circunstâncias.

3- Desacoplamento: Reduz o acoplamento entre o código cliente e as classes de objetos concretos. Isso significa que o código cliente não precisa depender diretamente das classes concretas, o que facilita a modificação ou substituição de implementações sem afetar o código cliente.

		Uma maior flexibilidade para mudanças pois reduz as dependencias diretas na criação dos objetos.
	Separar o objeto da sua criação.	E.g., assumindo que temos classes Circle e Rectangle derivadas da classe Figure e temos uma Main que gere estes objetos.
Factory pattern é um padrão de desenho que propoem uma interface para a criação de objetos, ou seja, esconder a criação direta do objeto. O principal problema que este resolve é a inflexibilidade de mudança, pois mesmo mudando o objeto, a interface de criação mantem-se.	A classe "Product" que declara a interface que é comum para todos os objetos e a sua respetiva implementação em produtos concretos. Por outro lado, a classe "Creator" que contem a interface para o método de criação dos objetos (abstrata) - "Factory method". E as respetivas implementações de "creators" para cada tipo de produto - e.g., CreatorCircle(). Ou seja, 1 Product -> 1 Creator, ...	Se na Main ao criarmos um novo objeto usarmos "new Circle()" estamos a criar uma dependencia direta na Main com a classe em especifico. Por outro lado, com este padrão de criação, nós utilizariamos "createCircle()", o que produzia uma menor dependencia. Ou seja, um exemplo positivo deste padrão seria, se nós mudarmos o nome da classe Circle para MyCircle não teriamos de alterar o código da nossa Main.
Na criação de um objeto para adicionarmos uma nova variantes dele, temos de fazer varias verificações, se tentarmos adicionar multiplos novos objetos de varião entre si então podemos ficar com um codigo muito complexo e dificil de iterar e modificar.Outro problema que factory pattern tenta resolver é não utilizar diretamente o "new" na criação de um novo objeto, mas sim um metodo numa superclass que permite subclasses alterarem o tipo de objeto que é criado.	Uma interface que é comum aos objetos.Objetos concretos que implementam a interface. A classe "Creator" retorna os objetos.Concrete Creators que se subrepoem ao criador de objetos para retornar um objeto diferente.	Um beneficio é evitar ter "tight coupling" entre os criadores e os seus objetos.Tambem podemos simplificar o codigo separando o processo de criação de objetos uma uma classe e metodo diferente seguindo o principio de Single Responsibility.Outro beneficio é para a extensão do codigo, pois se quiseremos adicionar novas instancias de objetos, em vez de adicionar requisitos para criação de um objeto, podemos simplemente adicionar um interface nova que faz a criação dele.

		You avoid tight coupling between the creator and the concrete products; Single Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support; Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code. Regarding the code on Refactoring Guru: Creator: The Dialog class acts as the Creator. It declares the factory method createButton() that returns an object of the Button class. The createButton() method is abstract, indicating that the Creator doesn't provide an implementation but leaves it to its subclasses. Concrete Creators: The WindowsDialog and WebDialog classes are Concrete Creators. They inherit from the Dialog class and override the createButton() method to return specific types of Button objects (WindowsButton and HTMLButton, respectively). Product: The Button interface defines the operations that all concrete products must implement. It declares the render() and onClick() methods.
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. It avoids the problem of having coupled code to existing classes.	1 - The Product declares the interface, which is common to all objects that can be produced by the creator and its subclasses. 2 - Concrete Products are different implementations of the product interface. 3 - The Creator class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface. 4 - Concrete Creators override the base factory method so it returns a different type of product.	
O Factory pattern é um padrão de design Creacional em que é criada uma interface para a criação de objetos numa super classe, mas que permite que subclasses alterem o tipo de objeto que é criado. Desta forma ajuda a resolver o problema da dificuldade de adicionar novos tipos de objetos referentes a uma classe já existente, permitindo a expansão do código sem necessidade de grandes alterações, o que melhora a sua flexibilidade. Também retira a dificuldade de implementar aplicações diferentes a cada tipo de objeto.	Na estrutura do padrão Factory, a mesma tem um conjunto de componentes principais. Para a criação dos diferentes objetos, deve existir uma superclasse (colocá-la como abstrata ajuda a que cada subclasse implemente cada método da sua maneira), que depois é expandida por cada sublcasse, com um método de criação e outras operações importantes. Cada subclasse a ela ligada deve implementar o seu próprio método de criação e que deve dar override ao da superclasse, cada um retornando um diferente tipo de produto. Por último, a superclasse deve implementar uma interface com todos os métodos necessários, sendo que cada classe de diferentes objetos implementa a mesma de forma adequada ao tipo de objeto.	Os benefícios são a possibilidade de expansão dos componentes internos do código e a possibilidade de reuso de objetos já existentes, ao invés de os reconstruirmos. Por último permite a separação do código de construção do produto (objetos), do código que o usa. Desta forma é útil quando, por exemplo, queremos criar um código para a gestão de uma empresa de transportes da qual não sabemos ainda quantos tipos de transporte a mesma vai implementar (aereo, terrestre, etc), permitindo a expansão para qualquer um através de uma superclasse geral transporte, por exemplo.
Factory Method Design Pattern defines an interface or abstract class for creating an object, allowing the subclasses to call which class to instantiate.	The main components consist of the interface or the abstract class, the class that implements the interface or abstract class. The abstract class that declares the Factory Method and the class that implements the factory method.	Allows the subclasses to choose which objects they want to create, streamlines the though process and usage of class instatiates

Some of the advantages are: Loose Coupling, Flexibility and Extensibility, Encapsulated Object Creation Logic and Improved Testability.

Reduces repeated codes to choose and instantiate an implementation of the interface for a combination of arguments and for the environment of the machines.

It is a creational design pattern and is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class.

An example of a practical example would be a Logging Framework in which the the client code requests a logger from the factory with the preferred destination and the factory returns the appropriate concrete product for handling log messages.

It solves the problem of having to tightly couple the creation of objects to the code that uses them.

The main components involved are Product Interface, Concrete Products, Creator and Client.

O Produto é a interface ou classe abstrata que define os métodos que o ConcreteProduct deve implementar. O ConcreteProduct é a classe que implementa a interface Product. O Creator é a classe abstrata que declara o Factory Method. O ConcreteCreator é a classe que implementa o Factory Method e retorna o ConcreteProduct

Continuação do outro:
Produto (mbway): interface para todos os objetos;
Produtos concretos que implementa a interface
O criador que é uma classe abstrata que retorna o pagamento

O padrão Factory Method sugere que substitua chamadas diretas de construção de objetos (operador new) por chamadas para um método. O seu problema consiste em resolver problemas de software relacionados a criação de classes.

Permite devolver uma instância de uma subclasse e assim a reutilização de um objeto já
criado. Outro benefício é muioto útil quando não se sabe o tipo ou classe do obejto que se quer criar.
É possível adicionar novos métodos de fábrica para novos produtos sem alterações
para o cliente, fazendo o programa aberto a extensão.
Um exemplo seria fazer uma fábrica de bicicletas e motas, onde temos a classe abstrata Veiculo que define a interface para os produtos (Mota e Bicicleta). A classe abstrata VeiculoFactory define a interface para as fábricas (MotaFactory e BicicletaFactory). Os clientes utilizam as fábricas para criar veículos sem se preocupar com os detalhes específicos da criação.

O Product declara a interface que é comum a todos os objetos que podem ser produzidos pelo criador e pelas suas subclasses.
O Concrete Product são diferentes implementações da interface do produto.
A classe Creator declara o método fábrica que retorna novos objetos de produto. O tipo de retorno deste método corresponde à interface do Product.
O Concrete Creators faz override do metodo fábrica básico para retornar um tipo diferente de produto.

Dada a necessidade de conformizar a arquitetura, no factory pattern define-se uma interface para criar um objeto numa superclasse, permitindo às suas subclasses alterar o tipo de objeto a ser criado através de um construtor virtual.
Para isso temos de, substitui-se as chamadas dos construtores de cada um dos objetos a criar, por chamadas ao método de fábrica, indicando, num argumento, o tipo de objeto a construir, retornando um objeto desse tipo (product).

The Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
Overall, the factory pattern promotes better software design by improving modularity, flexibility, maintainability, and consistency in object creation. It is particularly useful in scenarios where the creation of objects involves complex initialization logic or may vary based on runtime conditions.
In a simple way, if most of our code is coupled to a single class it is difficult to add other classes, requiring making changes to the entire codebase.

Este padrão é um padrão de design criacional, que tem como objetivo simplificar a criação de objetos, permitindo que esta criação seja feita por subclasses especificas através de uma mesma interface.
O problema que resolve é o de dependência direta na instanciação de classes, o que faz com que a flexibilidade e a possibilidade de alterar o código de forma significativa aumente.

It is the creation of new objects without using the "new" keyword and with this you solve the problem of initially name and implement the superClass with a specific theme or product. This way we can expand and turn scalable in the long term. An abstraction of the constructor.

The factory pattern typically involves several key components that work together to facilitate object creation. The main components involved in the structure of the factory pattern:

- The product
- Concrete Products
- Creator
- Concrete Creators

Product: Interface que vai ser comum a todos os objetos produzidos pelo Creator e as suas subclasses.
ConcreteProduct: São diferentes implementações da interface Product.
Creator: Classe que declara o método factory, que retorna um objeto do tipo Product.
ConcreteCreator: Implementa ou estende o Creator, sobrescrevendo o método factory para retornar uma instância de um ConcreteProduct.

We have the Creator that has the factory method that will return new products objects that it matches with the product interface. By turn this method abstract we force all subclasses to implement their own versions of the method. Concrete Creators overrides this factory method to return a diferent type of product.

Some benefits could be:

- Avoid tight coupling between the creator and the concrete products.

Example: In a system where a PaymentProcessor directly creates instances of payment gateways (for example PayPal), there is tight coupling between the PaymentProcessor and the concrete payment gateway classes. If the creation logic is moved into a factory, the PaymentProcessor only depends on the factory interface. This reduces coupling, as the PaymentProcessor doesn't need to know the specifics of each payment gateway implementation, adhering to the principle of avoiding tight coupling.

- You can move the product creation code into one place in the program, making the code easier to support (Single Responsability Principle).

Example: Initially, a DocumentCreator class directly creates documents of different formats (for example, PDF, Word). By moving the creation logic into a DocumentFactory, the responsibility of creating documents is separated from the DocumentCreator. Now, the DocumentFactory solely focuses on creating documents, adhering to the Single

Um dos beneficios seria adicionar novos tipos de objetos sem ter que alterar o código existente.
Mantendo a criação em apenas um lugar, em caso de necessidade de alteração, será mais facil manter.

Abstract the creation of new objects. This way, developers don't have to worry about internal object representations and constructors. Besides that, classes can change over time without having to update all constructions (using 'new').

É um padrão Criacional que fornece uma maneira de criar objetos sem especificar a classe exata do objeto que será criado. O Factory delega a responsabilidade de criar o objeto para subclasses, permitindo que o código cliente seja independente de como os objetos são criados, compostos e representados. Isso é útil quando o sistema precisa ser independente de como seus objetos são criados, compostos e representados.
O problema que o padrão Factory resolve no design de software é a necessidade de criar objetos de uma classe específica sem expor a lógica de criação ao código cliente. Isso é especialmente útil quando o sistema precisa criar diferentes tipos de objetos que compartilham uma interface comum, mas são implementados de maneiras diferentes. O Factory permite que o código cliente solicite um objeto sem saber a classe exata do objeto que está sendo criado, tornando o sistema mais flexível e menos acoplado.

A implementação do padrão Factory geralmente envolve:
- Definir uma interface comum para todos os produtos que podem ser criados pelo Factory.
- Implementar métodos de fábrica dentro da classe criadora que retornam novos objetos de produto. Esses métodos podem ser abstratos, forçando subclasses a implementarem suas próprias versões do método, ou podem retornar um tipo de produto padrão.
- Subclasses da classe criadora podem sobrescrever o método de fábrica para criar e retornar diferentes tipos de produtos, permitindo que o sistema crie diferentes tipos de objetos sem alterar o código cliente.

O padrão Factory ajuda a manter o código cliente livre de responsabilidades de criação de objetos, promovendo a separação de preocupações e a reutilização de código. Ele também facilita a adição de novos tipos de produtos no futuro, pois a lógica de criação de objetos é encapsulada dentro das subclasses do Factory.

É um padrão que tem como objetivo substituir a utilização direta da operação "new" ao instanciar um objeto para a chamada de um método de uma classe que seria a fábrica.
Ele resolve o problema de quando a criação de objetos é complexa ou pode mudar ao longo do tempo, abstraindo a sua criação.

O factory pattern é estruturado em torno dos seguintes componentes principais: Product é a interface que é comum aos objetos produzido pelo criador; Concrete Products, são as diferentes implementações da interface product; Creator, é a classe abstrata que declara o método de fábrica e que retorna objetos de Product; Concrete Creator, subclasses da classe Creator que subscrevem os métodos de fabrica base, para retornar um tipo diferente de Product.

Os benefícios são: redução do acoplamento, melhoria da coesão, reutilização, flexibilidade e escalabilidade. Exemplo concreto: uma classe de figuras geométricas, em que podemos querer criar círculos, quadrados, triângulos etc; dependendo das dimensões e forma.

The Factory Method pattern is a creational design pattern used in object oriented programming. Its purpose is to create objects without specifying the exact class of the object that will be created.

Product: Represents the abstract interface or base class for objects. Defines common operations that all concrete products must implement.
Concrete Products: Concrete classes that implement the product interface. Represent specific types of objects created by the factory.
Creator (Factory): An abstract class or interface. Declares the factory method responsible for creating products. Provides default implementations for common operations related to the product.
Concrete Creators (Concrete Factories): Subclasses of the creator. Implement the factory method to create specific types of products.

The Factory pattern streamlines object creation, promoting code flexibility and encapsulation. By centralizing instantiation logic, it enhances code maintainability and allows for easy extension without modifying existing client code. This design pattern fosters modular and reusable software development practices.