# Factory Design Pattern

# What is Factory Design Pattern

Factory method is a creational design pattern, i.e., related to object creation.

In Factory pattern, we create objects without exposing the creation logic to the client and the client uses the same common interface to create a new type of object.

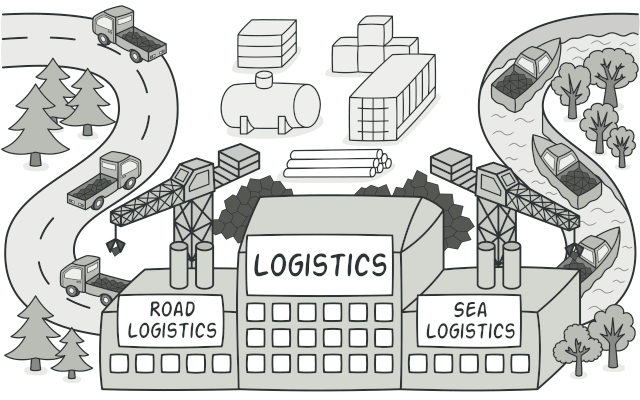
The idea is to use a static member-function (static factory method) that creates & returns instances, hiding the details of class modules from the user.  
A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library (explained below) in a way such that it doesn’t have tight coupling with the class hierarchy of the library.

Taken from:

https://refactoring.guru/design-patterns/factory-method

# Intent

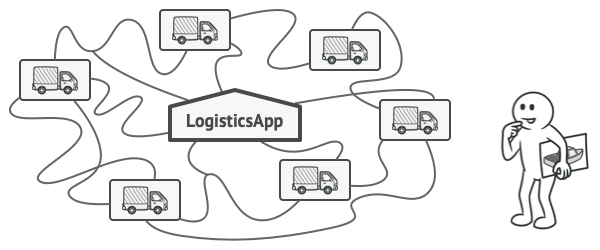
**Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



# Problem

Imagine that you’re creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.

After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app

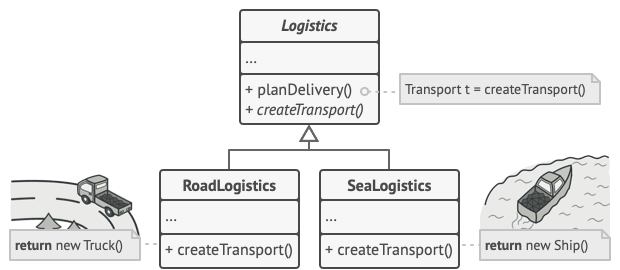


# Solution

The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method.

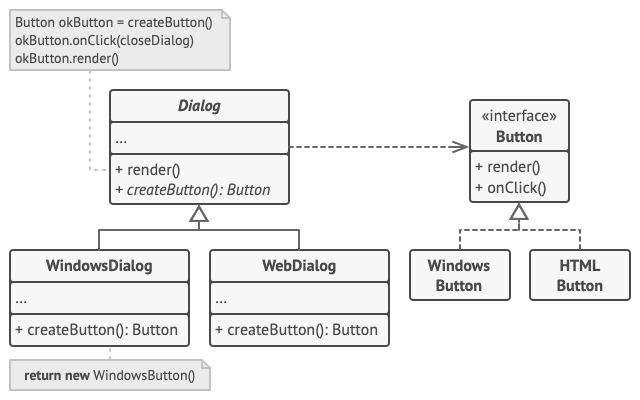
Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method.

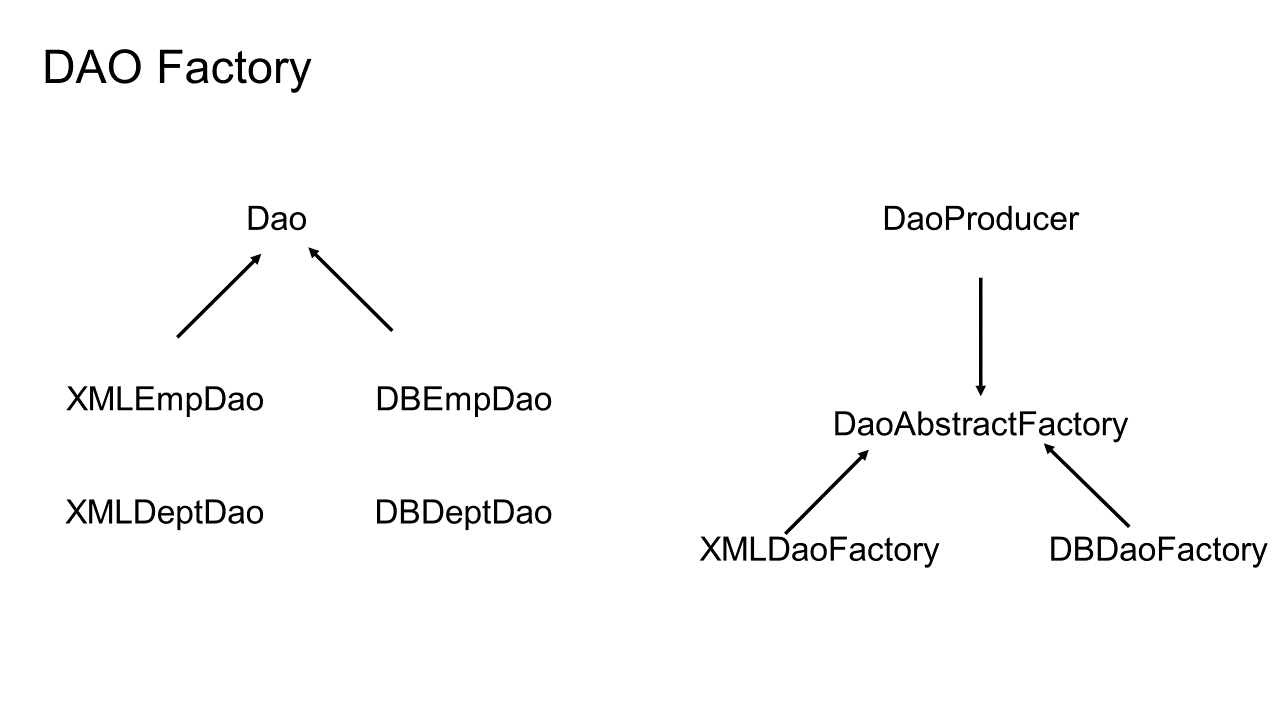
Objects returned by a factory method are often referred to as products.

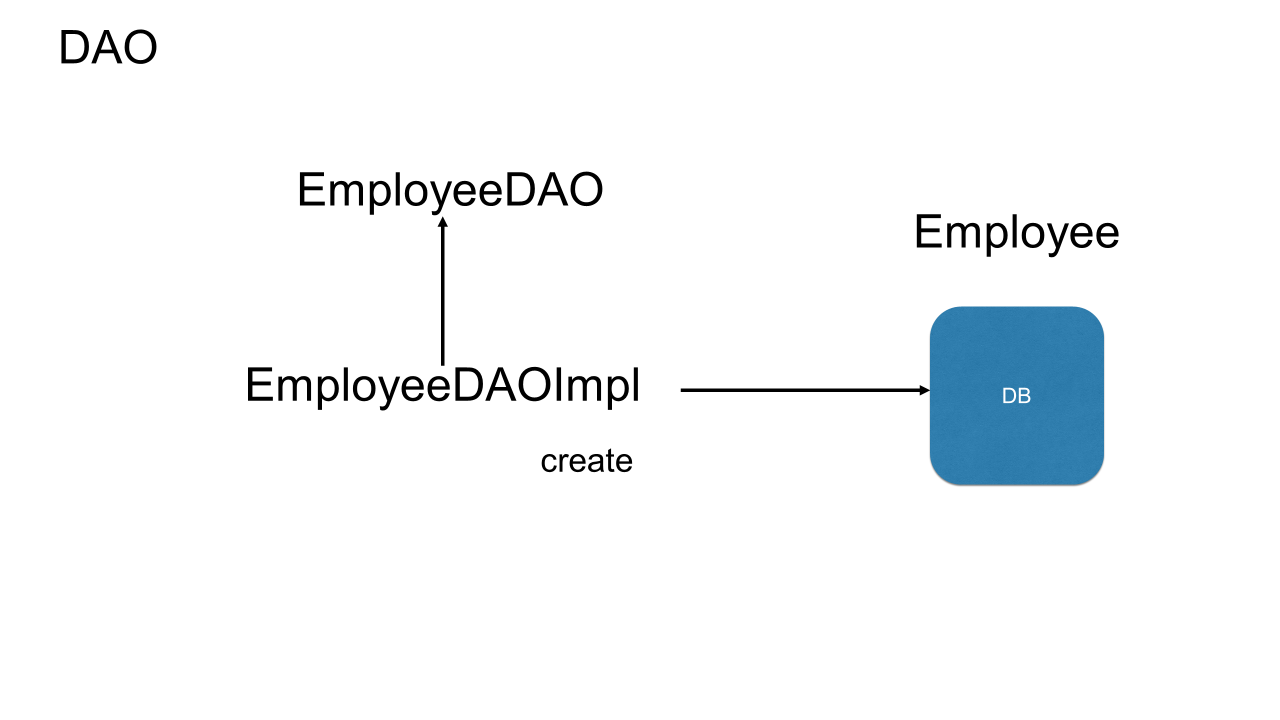


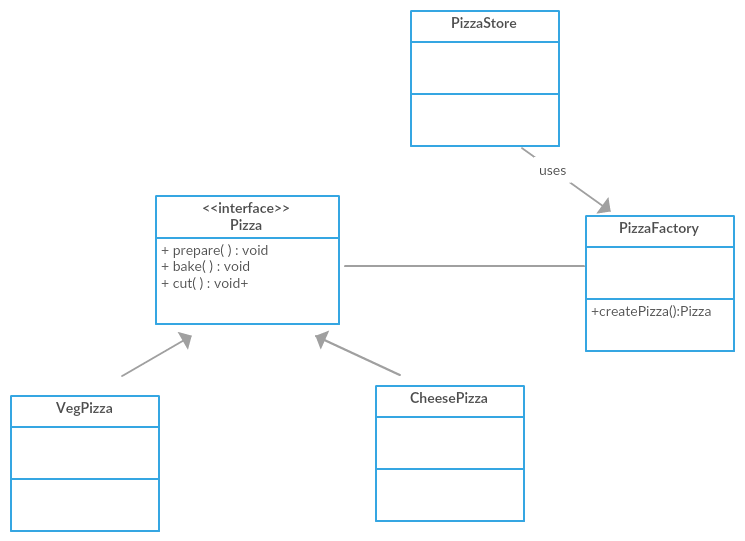


Another UML









# Applicability

* **Use the Factory Method when you don’t know 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.**

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.

* **Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.**

 Inheritance is probably the easiest way to extend the default behaviour of a library or framework. But how would the framework recognize that your subclass should be used instead of a standard component?

The solution is to reduce the code that constructs components across the framework into a single factory method and let anyone override this method in addition to extending the component itself.

Let’s see how that would work. Imagine that you write an app using an open source UI framework. Your app should have round buttons, but the framework only provides square ones. You extend the standard Button class with a glorious RoundButton subclass. But now you need to tell the main UIFramework class to use the new button subclass instead of a default one.

To achieve this, you create a subclass UIWithRoundButtons from a base framework class and override its createButton method. While this method returns Button objects in the base class, you make your subclass return RoundButton objects. Now use the UIWithRoundButtons class instead of UIFramework. And that’s about it!

# **Relations with Other Patterns**

* Many designs start by using [**Factory Method**](https://refactoring.guru/design-patterns/factory-method) (less complicated and more customizable via subclasses) and evolve toward [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory), [**Prototype**](https://refactoring.guru/design-patterns/prototype), or [**Builder**](https://refactoring.guru/design-patterns/builder) (more flexible, but more complicated).
* [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory) classes are often based on a set of [**Factory Methods**](https://refactoring.guru/design-patterns/factory-method), but you can also use [**Prototype**](https://refactoring.guru/design-patterns/prototype) to compose the methods on these classes.
* You can use [**Factory Method**](https://refactoring.guru/design-patterns/factory-method) along with [**Iterator**](https://refactoring.guru/design-patterns/iterator) to let collection subclasses return different types of iterators that are compatible with the collections.
* [**Prototype**](https://refactoring.guru/design-patterns/prototype) isn’t based on inheritance, so it doesn’t have its drawbacks. On the other hand, Prototype requires a complicated initialization of the cloned object. [**Factory Method**](https://refactoring.guru/design-patterns/factory-method) is based on inheritance but doesn’t require an initialization step.
* [**Factory Method**](https://refactoring.guru/design-patterns/factory-method) is a specialization of [**Template Method**](https://refactoring.guru/design-patterns/template-method). At the same time, a Factory Method may serve as a step in a large Template Method.