Factory-method & Factory & Abstract Factory Design Patterns

University of Missouri – Columbia
Ekincan Ufuktepe

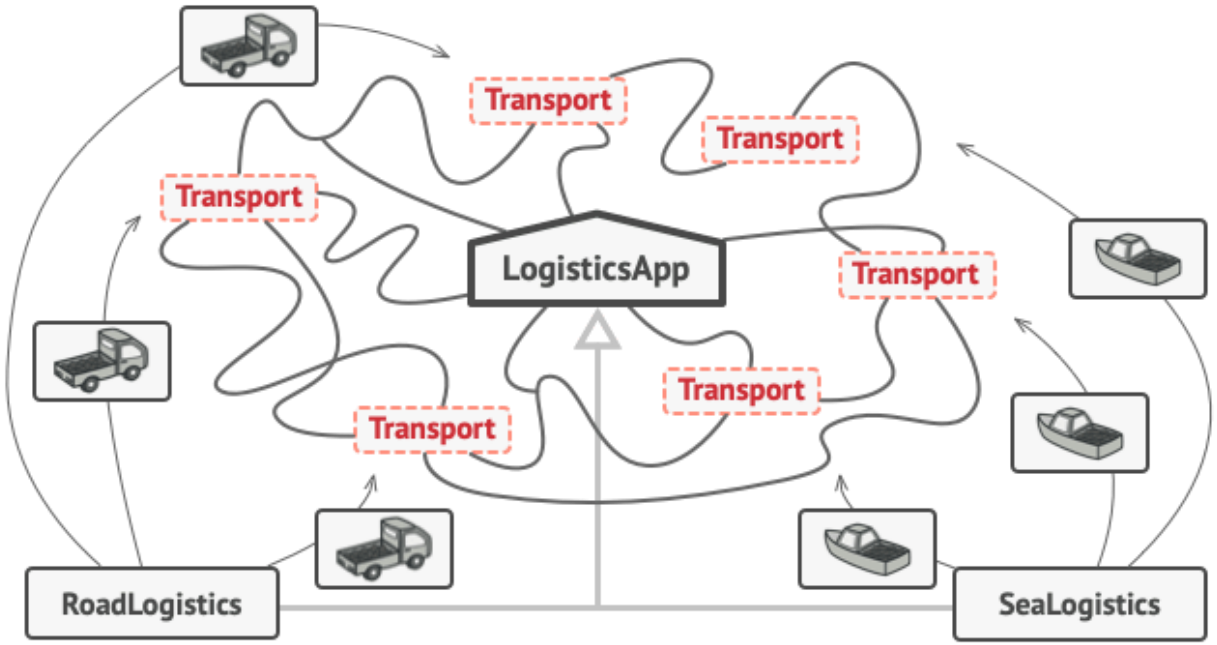
Factory-method Design Pattern: https://refactoring.guru/design-patterns/factory-method

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.

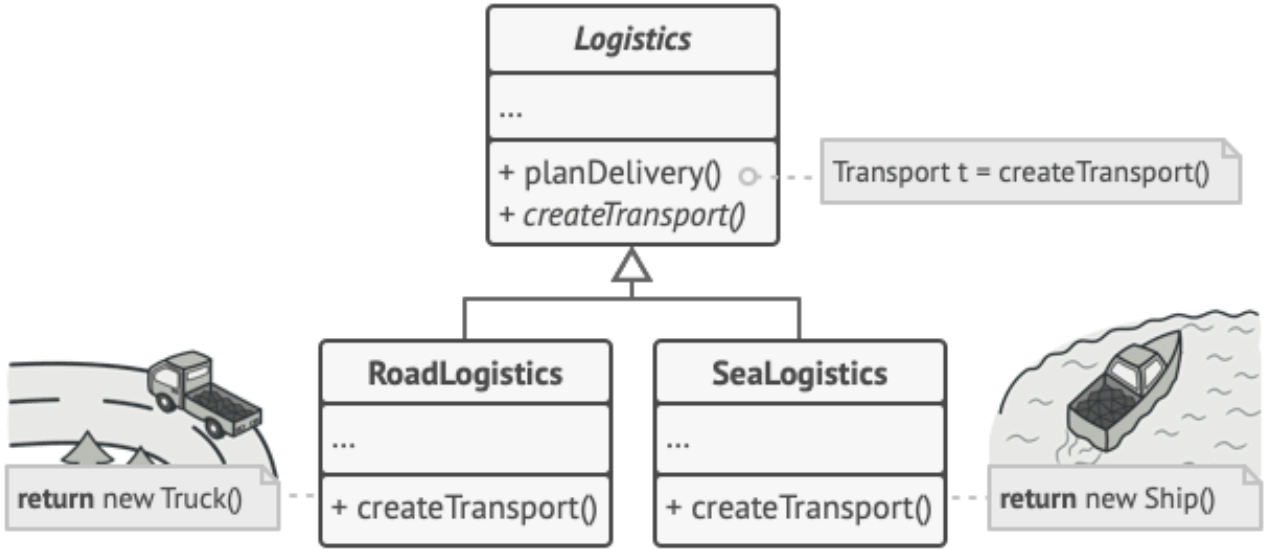
Problem

- Great news, right? But how about the code? At present, most of your code is coupled to the ***Truck*** class. Adding ***Ships*** into the app would require making changes to the entire codebase. **Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again**.
- As a result, you will ***end up with pretty nasty code***, riddled with conditionals that switch the app's behavior depending on the class of transportation objects.



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.

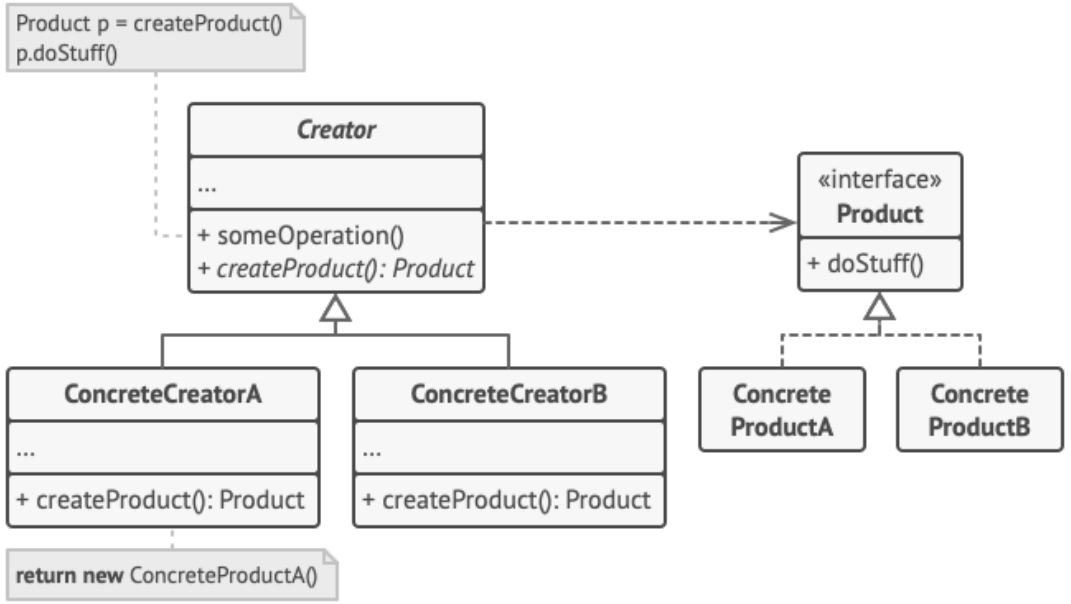


Solution

- At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another. However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.
- There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

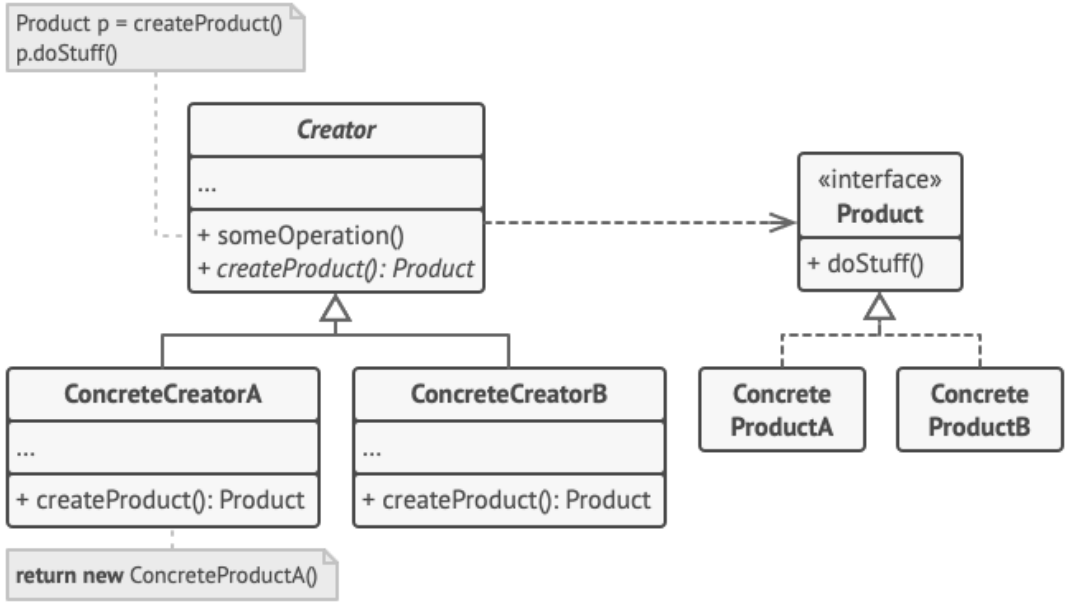
1 - Define the Interface

- The Product declares the interface, which is common to all objects that can be produced by the creator and its subclasses.



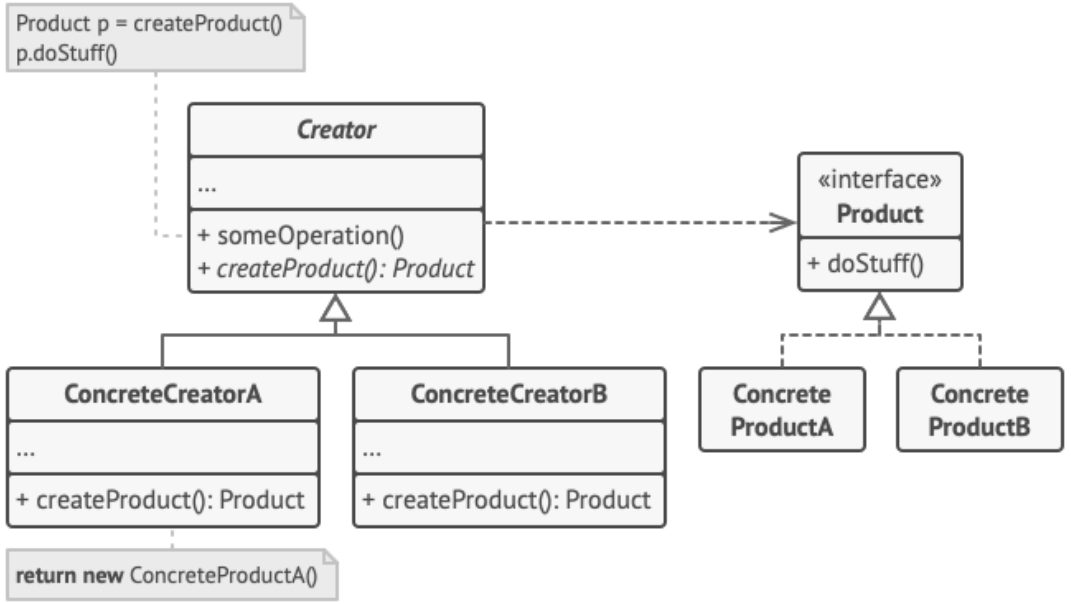
2 – Implement Concrete Classes of Interface

1. Concrete Products are different implementations of the product interface.



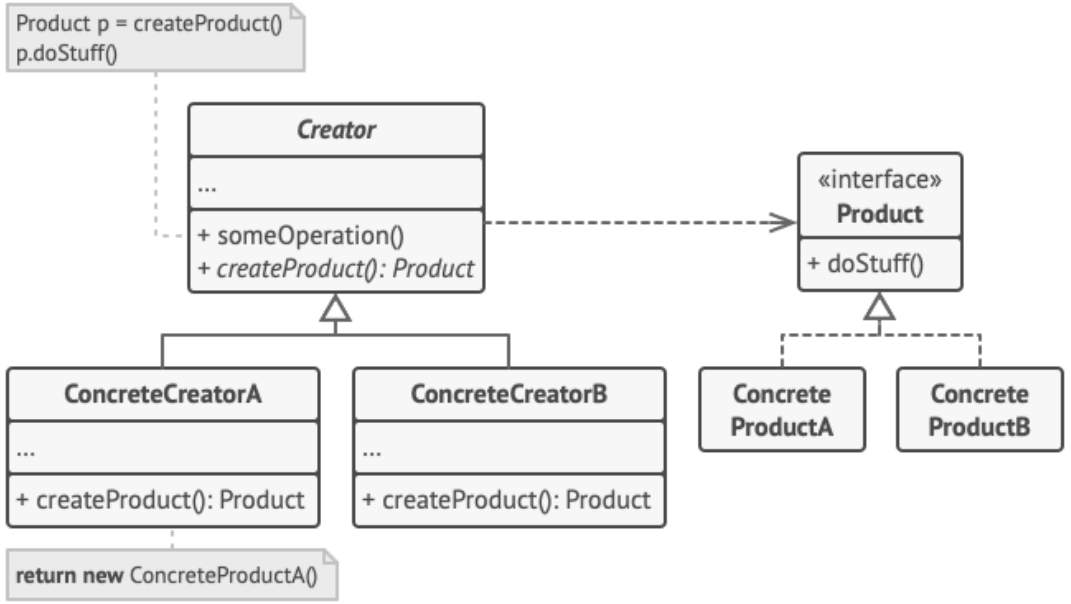
3 – Implement your Factory Class

- 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.
- You can declare the factory method as abstract to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.



4 – Override Factory method(s) for Creation

- Concrete Creators override the base factory method so it returns a different type of product.
- Note that 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.



Pros & Cons

Pros

- 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.

Cons

- The code may become more complicated since you need to **introduce a lot of new subclasses to implement** the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

Before Factory-method Design Pattern

- There was ***Factory Design Pattern (root of Factory design)***
- **Factory Design pattern** created its objects through **one single static function, that returns a super class type.**
- On the other hand, **Factory-method Design Pattern** promotes inheritance to create objects.
- So, if you have 100 types of product classes:
  - **Factory Design pattern**, will have one single function.
  - **Factory-method Design pattern**, will have 100 functions.
- Which, one do you think is better? Why?

What about Abstract Factory DP?

- The **Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes. It’s like a "factory of factories.“
- While:
  - The **Factory Pattern** provides a static method to create and return instances of different classes based on input parameters. It hides the instantiation logic from the client.
  - The **Factory Method Pattern** defines an interface for creating an object but lets subclasses alter the type of objects that will be created. It promotes *inheritance*.

Comparison

Feature	Factory (Simple)	Factory Method	Abstract Factory
Purpose	Centralizes object creation.	Allows subclasses to decide which class to instantiate.	Creates families of related objects.
Flexibility	Low – adding a new product requires modifying the factory.	Medium – new products via subclassing.	High – supports multiple product families.
Pattern Type	Creational (simple variant).	Creational, uses inheritance.	Creational, uses composition.
Client Knowledge	Client knows factory method.	Client uses factory subclass.	Client uses factory interface.
Key Use Case	When there's simple conditional creation.	When a class delegates instantiation to subclasses.	When systems need to be independent of how their products are created.

Aspect of Different Factory DP

Aspect	Factory Pattern	Factory Method Pattern	Abstract Factory Pattern
Object Creation	Static method in one class.	Subclass decides which object to create.	Factory of factories for related objects.
Flexibility	Low – changes need code edits.	Medium – add subclasses for new types.	High – supports multiple product families.
Design Principle	Encapsulation of instantiation.	Open/Closed principle (via inheritance).	Dependency inversion (via factory interfaces).