Creational – Factory Method

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

Solves:

Without a superclass all the codebase sits within one class – when we want to modify the API to introduce a new object with similar/shared functionality – We would have to modify the entire code base.

As a result, you will end up with pretty nasty code, riddled with conditionals that switch the app's behaviour depending on the class of the objects.

Implement:

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

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.*

At first glance, this change may look pointless as we just moved the constructor call from one part of the program to another.

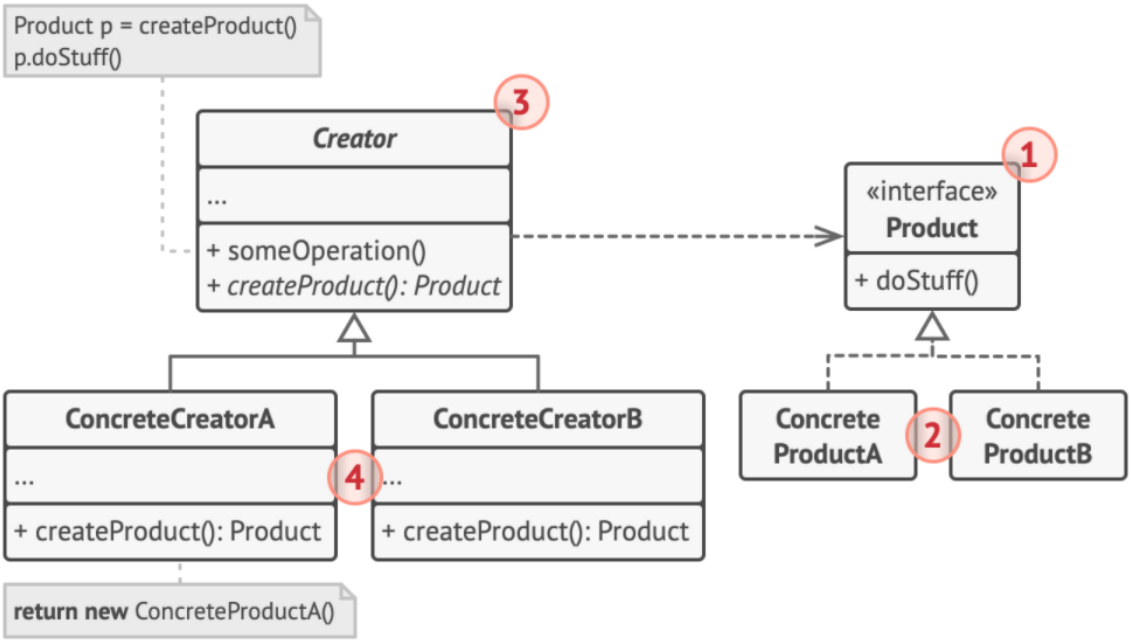
However, now you can override the factory method in a subclass and change the class of products being created by the method.

Limitation:

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.

The code that uses the factory method (often called the *client* code) doesn't see a difference between the actual products returned by various subclasses. The client treats all the products as an abstract `object`. The client knows that all objects are supposed to have the methods of the interface, but exactly how it works isn't important to the client.

Structure:



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.

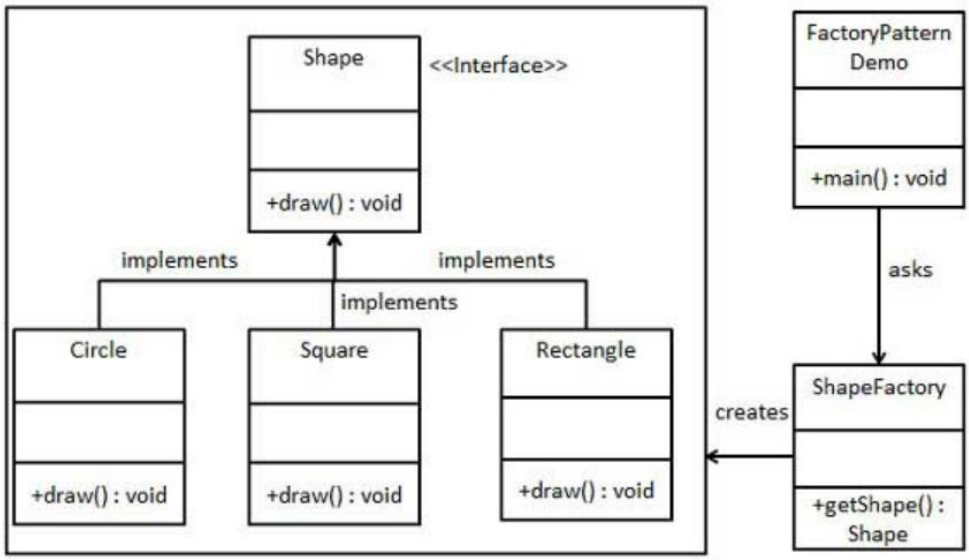
   You can declare the factory method as abstract to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.

   Note, despite its name, product creation is **not** the primary responsibility of the creator. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes. Here is an analogy: a large software development company can have a training department for programmers. However, the primary function of the company as a whole is still writing code, not producing programmers.

4. **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.

Code Structure:



Instances of Application:

>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 save system resources by reusing existing objects instead of rebuilding them each time.

Pros and 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.

Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).
- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.
- You can use **Factory Method** along with **Iterator** to let collection subclasses return different types of iterators that are compatible with the collections.
- **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** is based on inheritance but doesn't require an initialization step.
- **Factory Method** is a specialization of **Template Method**. At the same time, a *Factory Method* may serve as a step in a large *Template Method*.