Creational

Abstract Factory

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Solves:

Imagine you create a program that deals with related objects of several variants and you need a way to create individual objects which match other objects of the same variant. You also need a way to add new products to the program without modifying the codebase significantly

Implementation:

The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products follow those interfaces.

For example, all chair variants can implement the Chair interface; all coffee table variants can implement the CoffeeTable interface, and so on.

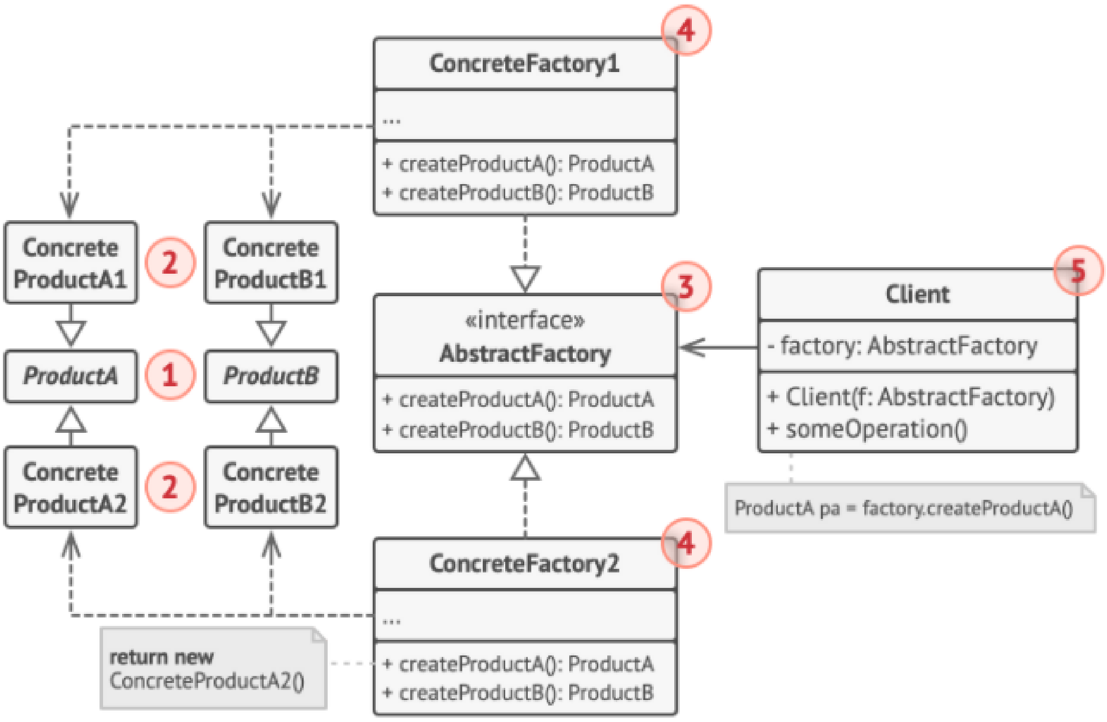
The next move is to declare the Abstract Factory—an interface with a list of creation methods for all products that are part of the product family (for example, createChair, createSofa and createCoffeeTable). These methods must return abstract product types represented by the interfaces we extracted previously: Chair, Sofa, CoffeeTable and so on.

For each variant of a product family, we create a separate factory class based on the AbstractFactory interface. A factory is a class that returns products of a particular kind. For example, the ModernFurnitureFactory can only create ModernChair, ModernSofa and ModernCoffeeTable objects.

The client code has to work with both factories and products via their respective abstract interfaces. This lets you change the type of a factory that you pass to the client code, as well as the product variant that the client code receives, without breaking the actual client code.

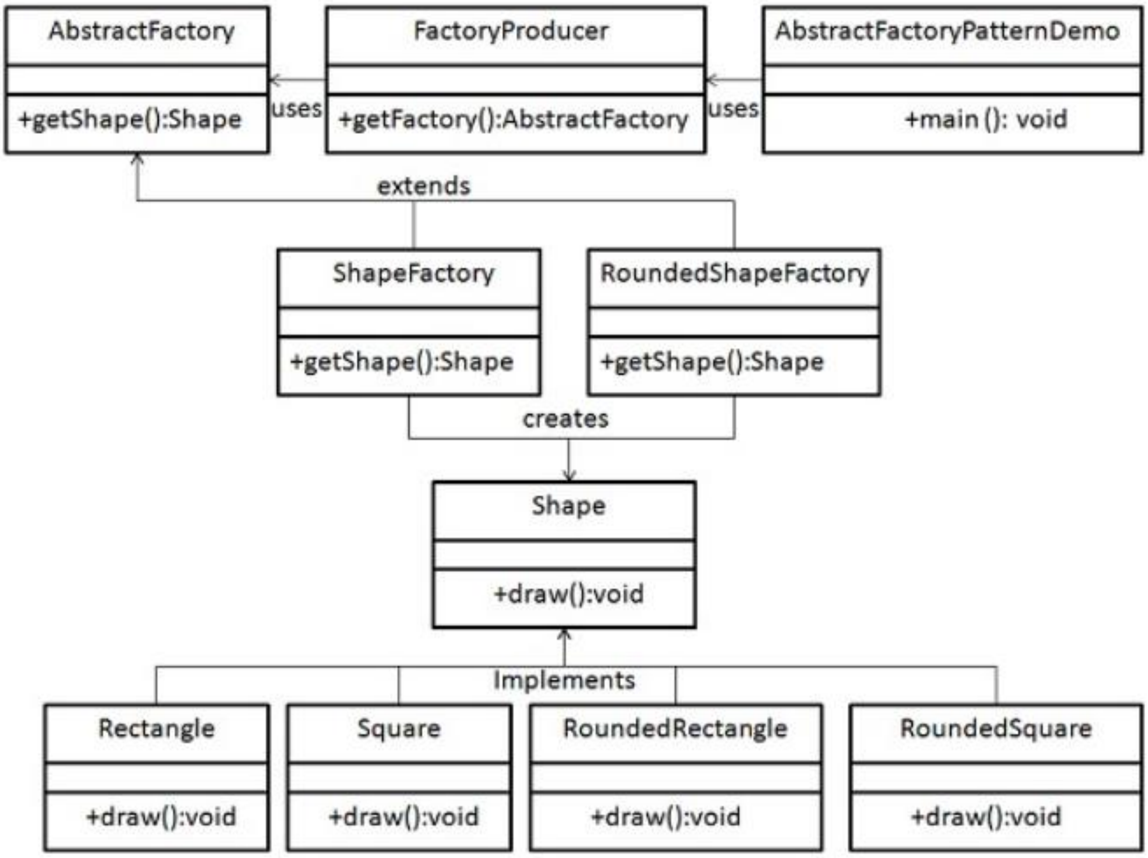
if the client is only exposed to the abstract interfaces, what creates the actual factory objects? Usually, the application creates a concrete factory object at the initialization stage. Just before that, the app must select the factory type depending on the configuration or the environment settings.

Structure


4
ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
2
Concrete ProductB1
ProductA
1
ProductB
Concrete ProductA2
2
Concrete ProductB2
3
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
5
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
4
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

1. **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.
2. **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).
3. The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.
4. **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.
5. Although concrete factories instantiate concrete products, signatures of their creation methods must return corresponding *abstract* products. This way the client code that uses a factory doesn't get coupled to the specific variant of the product it gets from a factory. The **Client** can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

Code Structure



Instances of Application

>Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.

The Abstract Factory provides you with an interface for creating objects from each class of the product family. As long as your code creates objects via this interface, you don't have to worry about creating the wrong variant of a product which doesn't match the products already created by your app.

- Consider implementing the Abstract Factory when you have a class with a set of **Factory Methods** that blur its primary responsibility.
- In a well-designed program *each class is responsible only for one thing*. When a class deals with multiple product types, it may be worth extracting its factory methods into a stand-alone factory class or a full-blown Abstract Factory implementation.

Pros and Cons

Pros:

- You can be sure that the products you're getting from a factory are compatible with each other.
- You avoid tight coupling between concrete products and client code.

- *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.
- *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.

Cons:

- The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

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).
- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. *Abstract Factory* returns the product immediately, whereas *Builder* lets you run some additional construction steps before fetching the product.
- **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.
- **Abstract Factory** can serve as an alternative to **Facade** when you only want to hide the way the subsystem objects are created from the client code.
- You can use **Abstract Factory** along with **Bridge**. This pairing is useful when some abstractions defined by *Bridge* can only work with specific implementations. In this case, *Abstract Factory* can encapsulate these relations and hide the complexity from the client code.
- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.