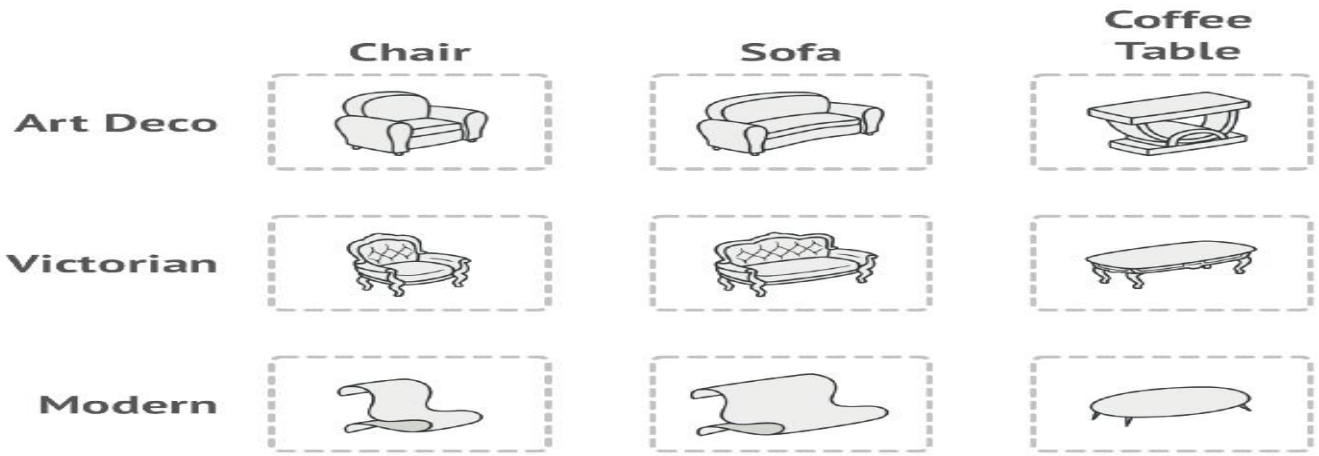
Abstract Factory

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

Problem

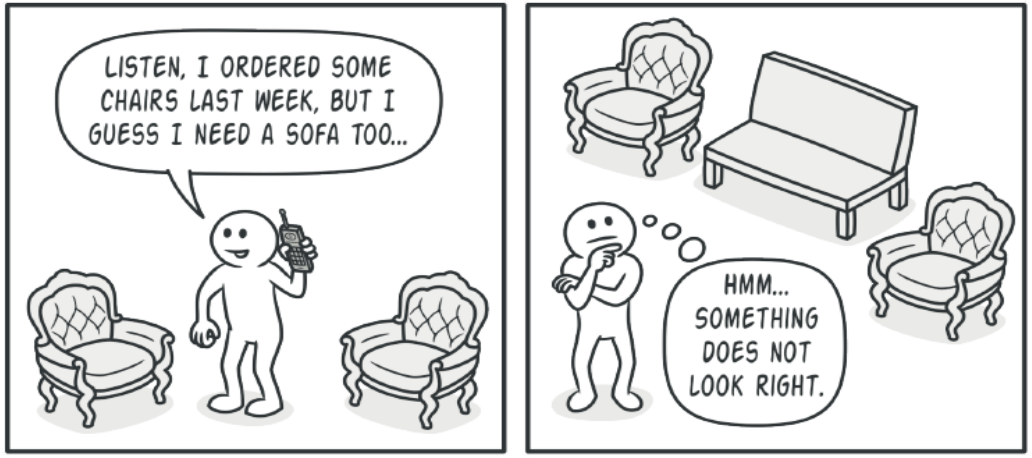
- Imagine that you’re creating a furniture shop simulator. Your code consists of classes that represent:
- 1. A family of related products, say: Chair + Sofa + CoffeeTable .
- 2. Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants: Modern , Victorian , ArtDeco .



Product families and their variants.

Contd.

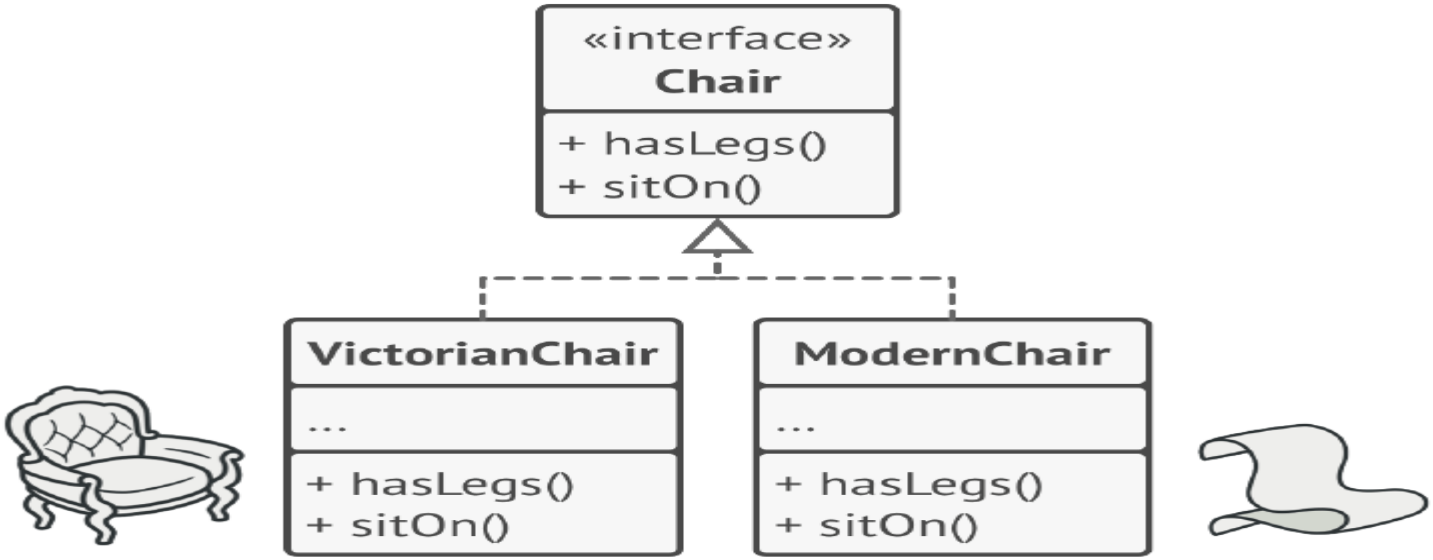
- You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.
- Also, you don’t want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn’t want to change the core code each time it happens.



A Modern-style sofa doesn’t match Victorian-style chairs.

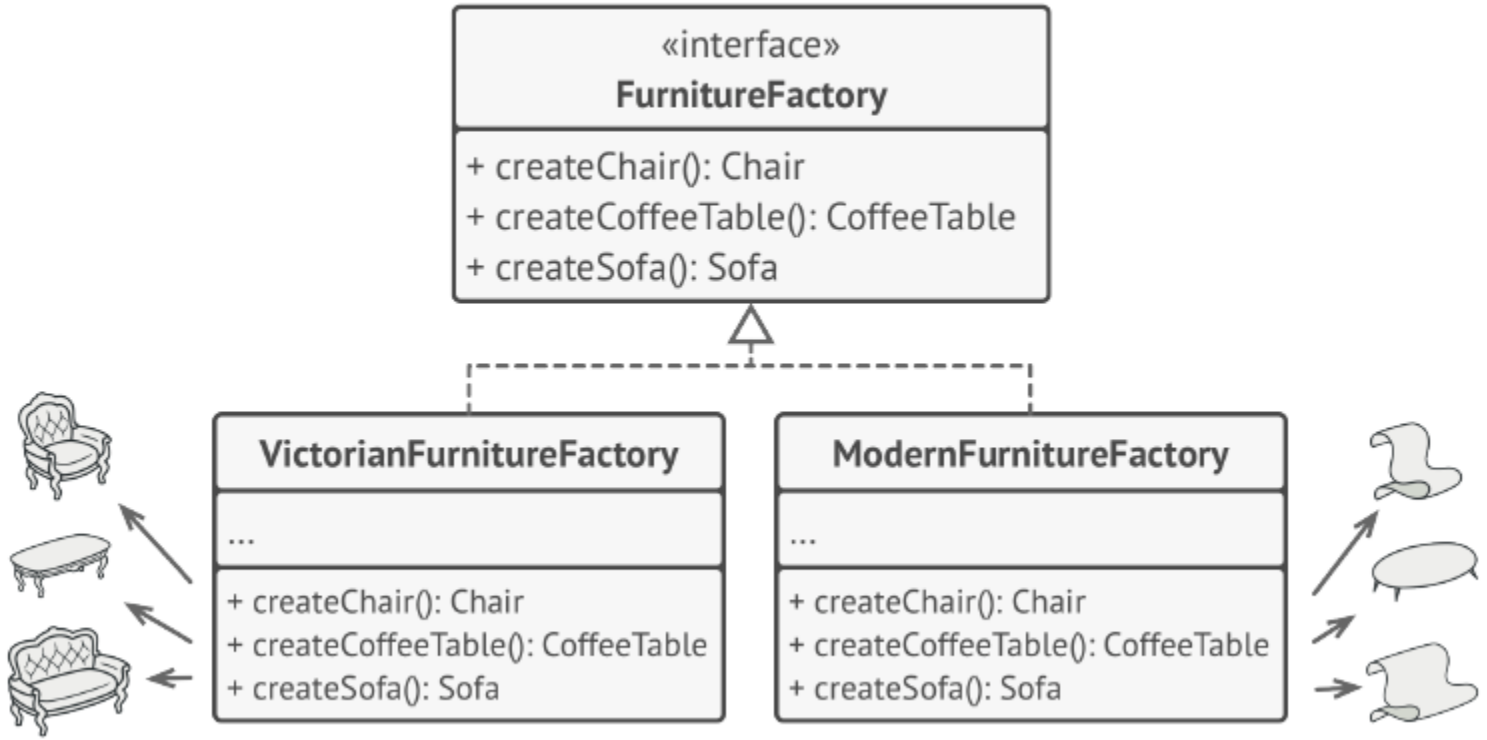
Solution

- 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



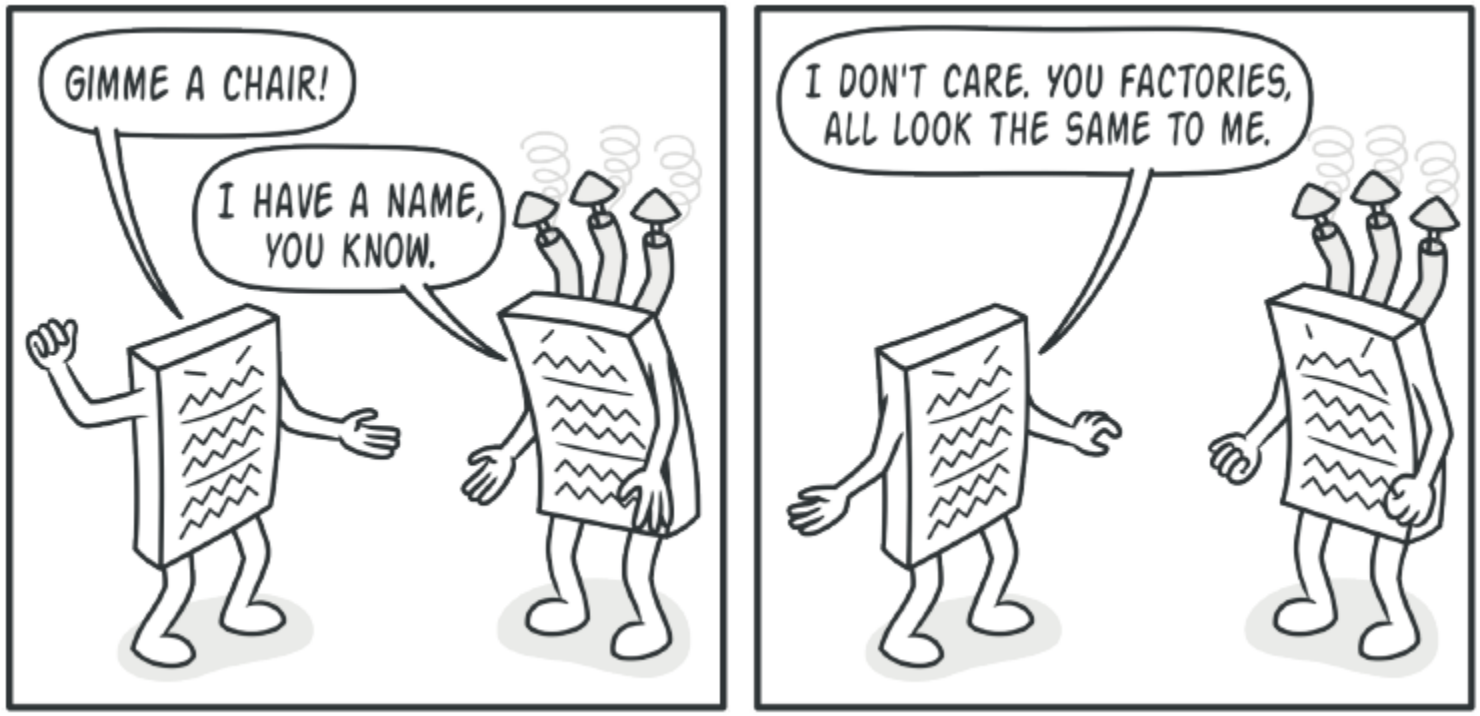
All variants of the same object must be moved to a single class hierarchy.

Contd.



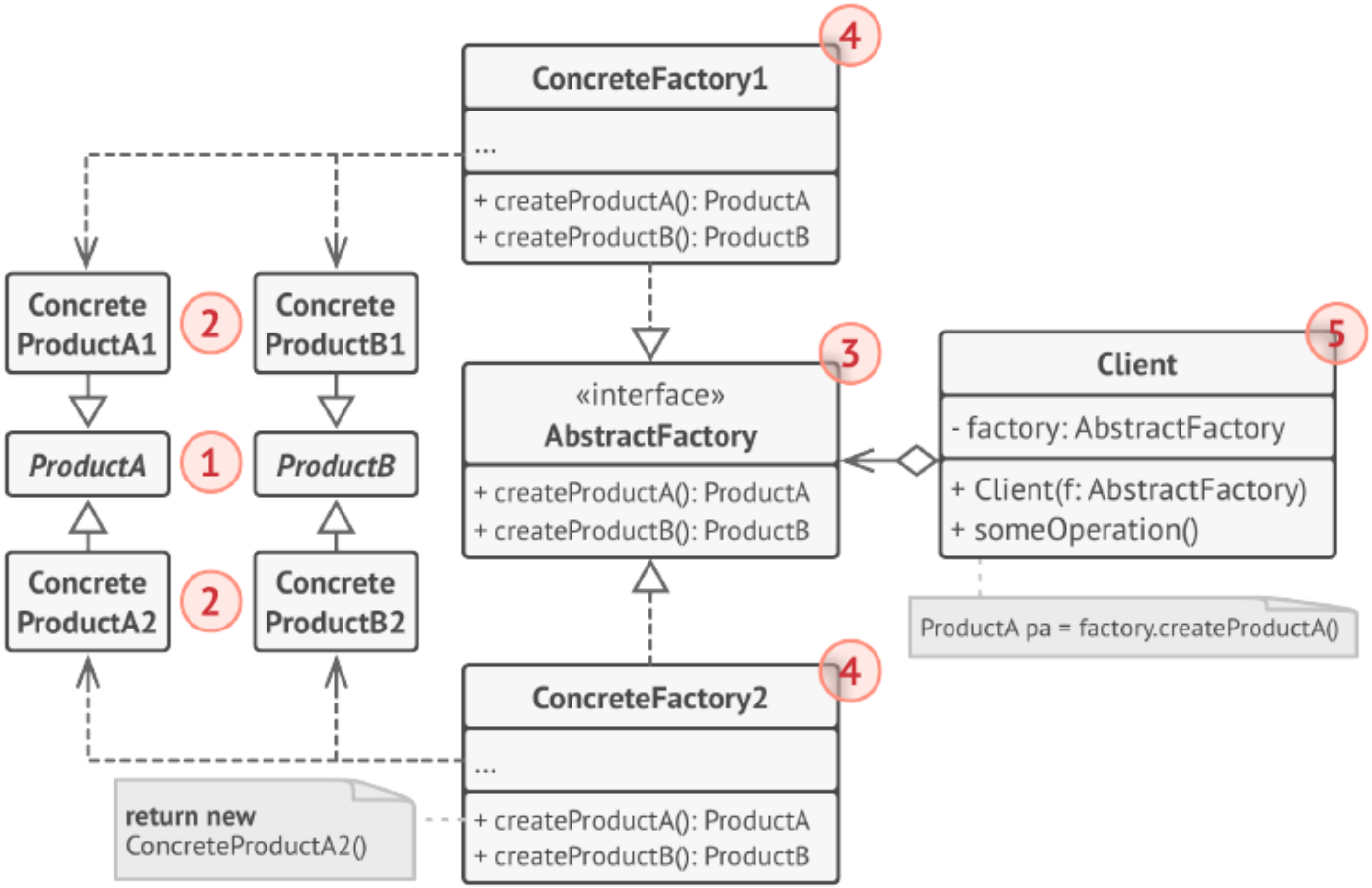
Each concrete factory corresponds to a specific product variant.

Contd.



The client shouldn’t care about the concrete class of the factory it works with.

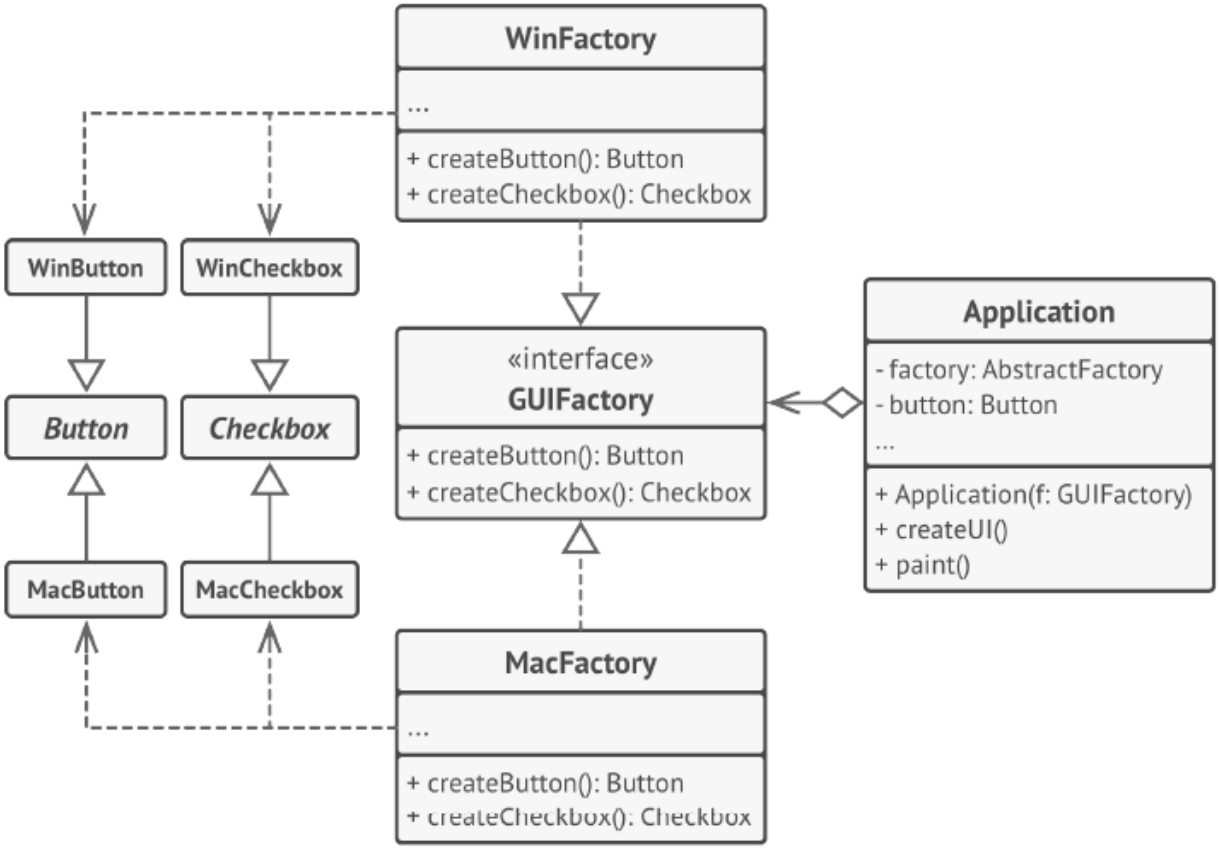
Structure.


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

Contd.

- **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.
- **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).
- The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.
- **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.
- 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.

Implementation



The cross-platform UI classes example.

Applicability

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

How to implement?

- Map out a matrix of distinct product types versus variants of these products.
- Declare abstract product interfaces for all product types. Then make all concrete product classes implement these interfaces.
- Declare the abstract factory interface with a set of creation methods for all abstract products.
- Implement a set of concrete factory classes, one for each product variant.
- Create factory initialization code somewhere in the app. It should instantiate one of the concrete factory classes, depending on the application configuration or the current environment. Pass this factory object to all classes that construct products.
- Scan through the code and find all direct calls to product constructors. Replace them with calls to the appropriate creation method on the factory object.

Pros and Cons

- One can be sure that the products you're getting from a factory are compatible with each other.
- Avoidance of tight coupling between concrete products and client code.
- *Single Responsibility Principle*. One can extract the product creation code into one place, making the code easier to support.
- *Open/Closed Principle*. One can introduce new variants of products without breaking existing client code.
- The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.