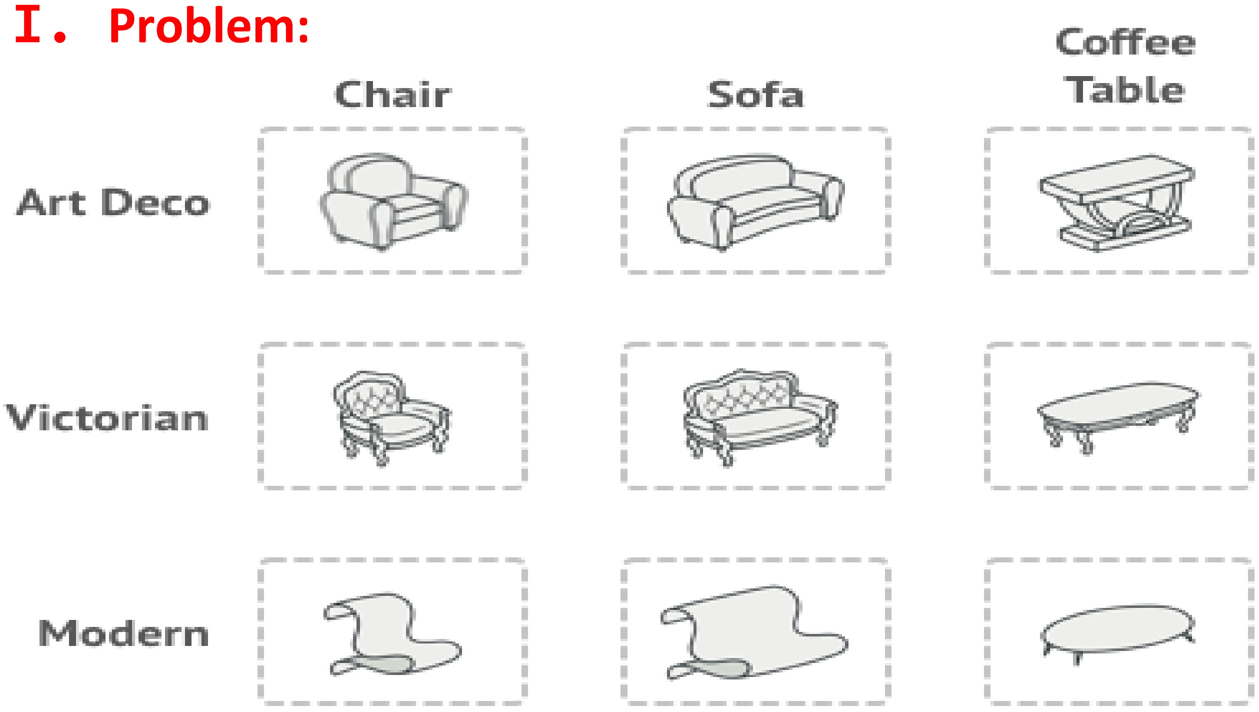
ABSTRACT FACTORY

I. Problem:



- <u>Without Abstract Factory</u>:

+If users creating a furniture shop simulator → users will have 3 class represent:

+A family of related products: Chair + Sofa + CoffeeTable

+Several variants of this family (art deco, victorian, modern)

→Need a way to create individual furniture objects → they can match other objects of the same family

LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...
HMM... SOMETHING DOES NOT LOOK RIGHT.

-More than that, users won't wanna change existing code when adding new products or families of products to the program.And furniture vendors update their catalogs very often and users, definitely, won't want to change users' core code each time changing.

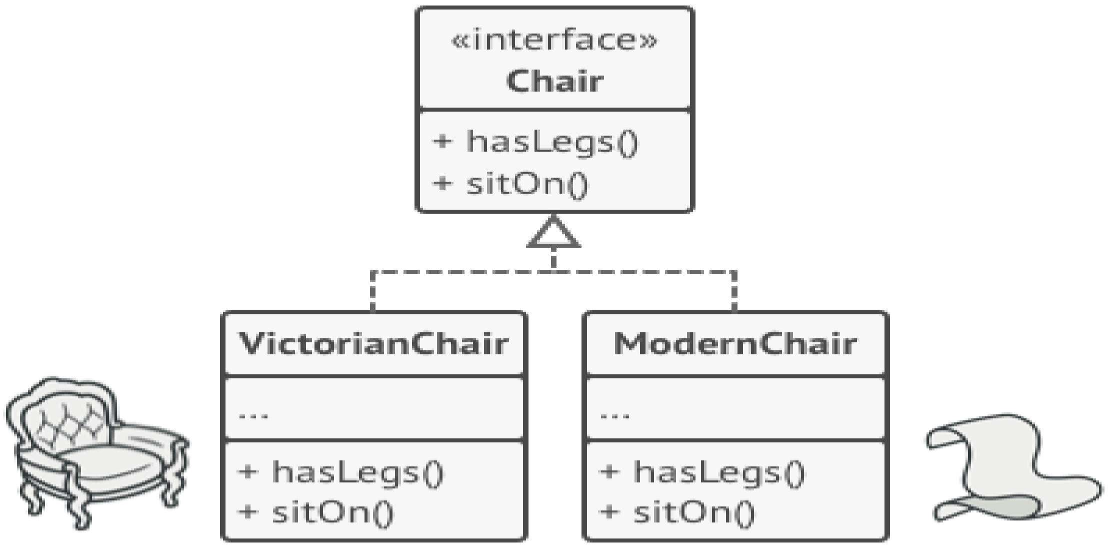
- ❖ The disadvantage of this example: The users have to change codebase when adding new classes.

  - <u>With abstract factory:</u>

The Abstract Factory pattern suggests:

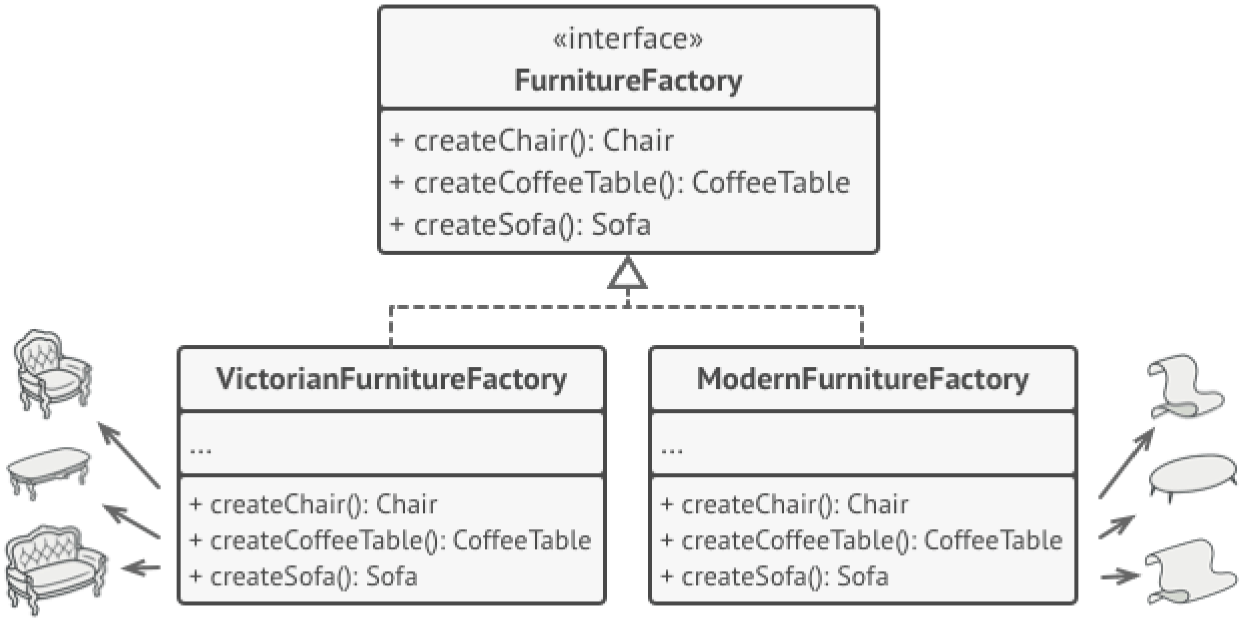
+explicitly declare interfaces for each distinct product (chair or sofa or coffeetable) of the product family

+make all variants of products follow those interface.



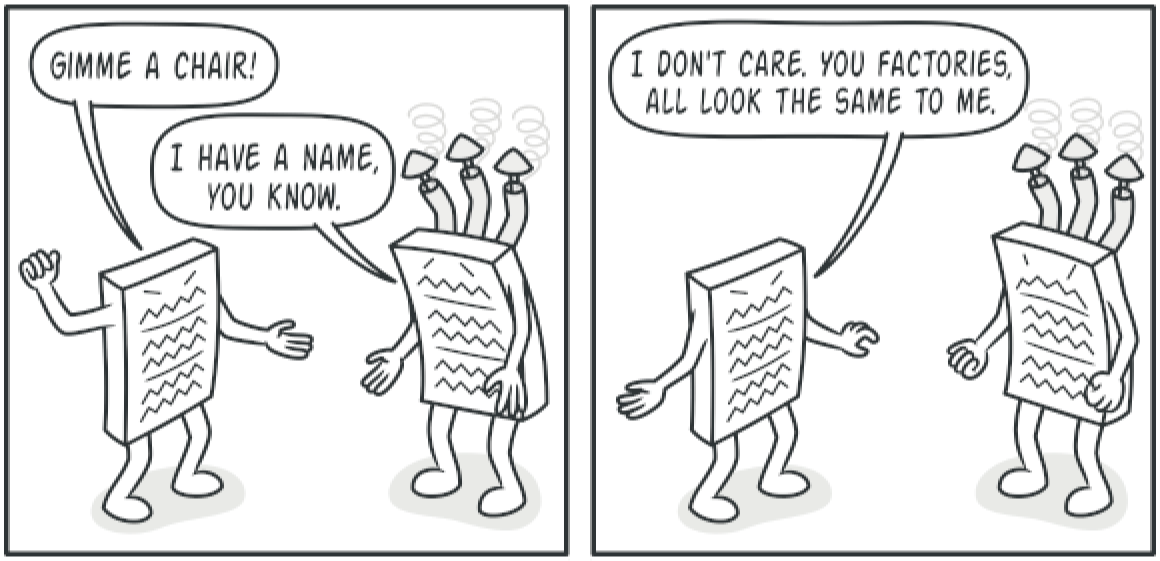
+declare the Abstract Factory – an interface with a list of creation methods (createChair or CreateSofa or createCoffeeTable) for all products that are part of product family

→Must return abstract product types represented by the interfaces (Chair or Sofa or CoffeeTable)



-Each concrete factory corressponds to a specific product variant

-For each variant of a product family → create a seperate factory class based on AbstractFactory interface (a factory is a class returns products of a particular kind)



    + Example: ArtDecoFurnitureFactory can only create ArtDecoChair, ArtDecoSofa, ArtDecoCoffeeTable objects

The clients don’t have to be aware of the factory’s class nor does it matter what kind of objects it gets → the clients must treat all chairs in the same manner using the abstract interface.

The application creates a concrete factory object at the initialization → app must select the factory type depending on the configuration or the environment settings.

→Definition: An Abstract Factory is a creational design patterns let users product families of related objects without specifying their concrete classes

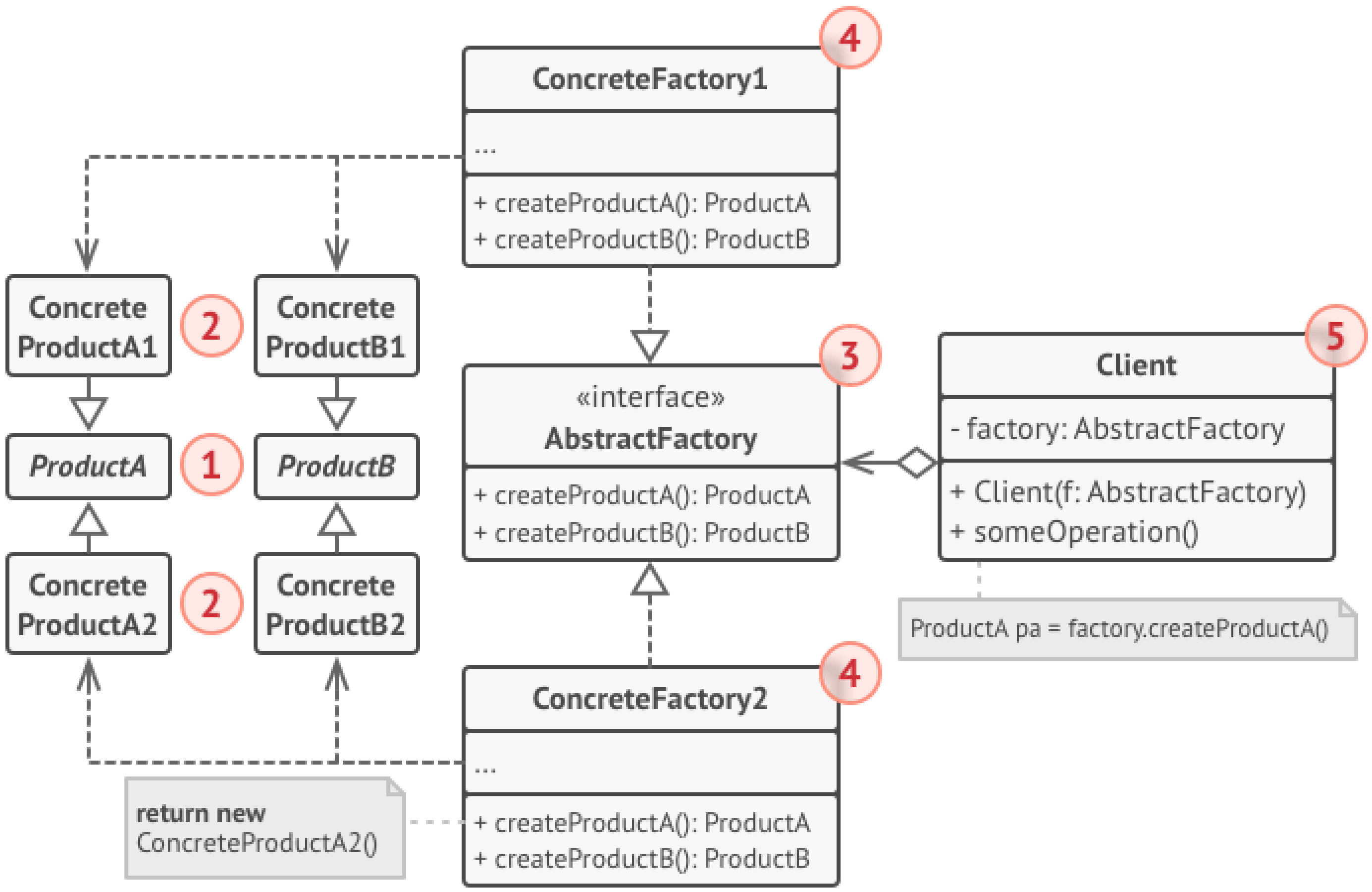
II. Structure

Concrete products: various implementations of abstract products. Each abstract product must be implement in all given variants

Abstract factory: declares a set of methods for creating each of the abstract products

Client: work with any concrete factory/product variant as long as it communicates with their objects via abstract interfaces

Abstract products: Declares interface for a set of distinct but related products which make up a product family



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

Concrete factories: implement creation methods of the abstract factory. Each concrete factory corresspond to a specific variant of products + creates those product variants

III. How to implement:

- Map out a matrix of distinct product types versus variants of these products
- Declare abstract product interfaces for all product types → make all concrete product classes implement these classes

- Declare the abstract factory interface with a set of creation methods for all abstract products
- Create factory initialization code in the app → instantiate one of the concrete factory classes depend on the application configuration or the current environement → pass factory object to all classes that construct products

1. Scan through the code and find all the direct calls to product constructors → replace them with calls to the approiate creationg method

IV. Applicability

- When the users' code needs to work with various familiies of relatied products. However, users don't want it to depend on the concrete classes of those products – they might be unknown beforehand or users simply want to allow for future extensibility

→Abstract Factory provides an interface for creating objects from each class of the product family

V. Advantage and Disadvantage:

Pros of Abstract Factory Pattern	Cons of Abstract Factory Pattern
• Can be sure that the products the users are getting from a factory are compatible with each other • Users avoid tight coupling between concrete products and client code • Single Responsibility Principle: the user can extract the product creation code into 1 place only → making the code easier • Open/Closed Principle: The user can introduce new variants of products without	• A lot of new interface and classes are introduced along with pattern → The code may become more complicated than it should be

breaking existing client code	

VI. Relations with Other patterns

- Factory Method: Many designs start by using Factory Method (less complicated and more customizable via subclasses) and envolve 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.
- Prototype: classes are often based on a set of Factory Methods, but you can also use Prototype to compose the methods on these classes.
- Facade: 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.
- Bridge: 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.