Software Design Patterns

Lecture 3

Factory Method
Abstract Factory

Dr. Fan Hongfei

25 September 2025

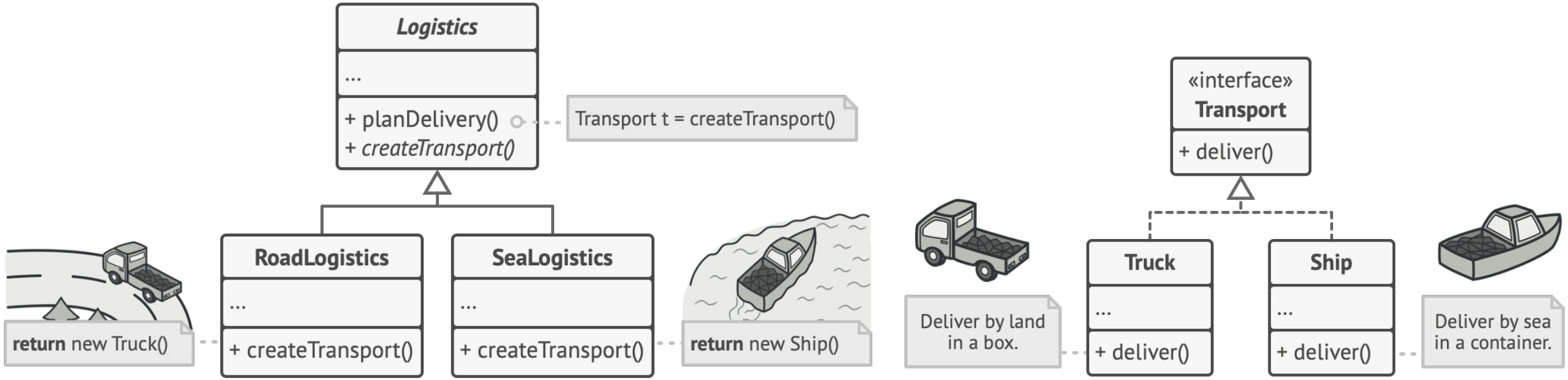
Factory Method: Problem

- **Example: a logistics management application**
  - First version: handling transportation by trucks, with a Truck class
  - Later: new requests to incorporate sea logistics, and more...

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/factory-method]

Factory Method: Solution

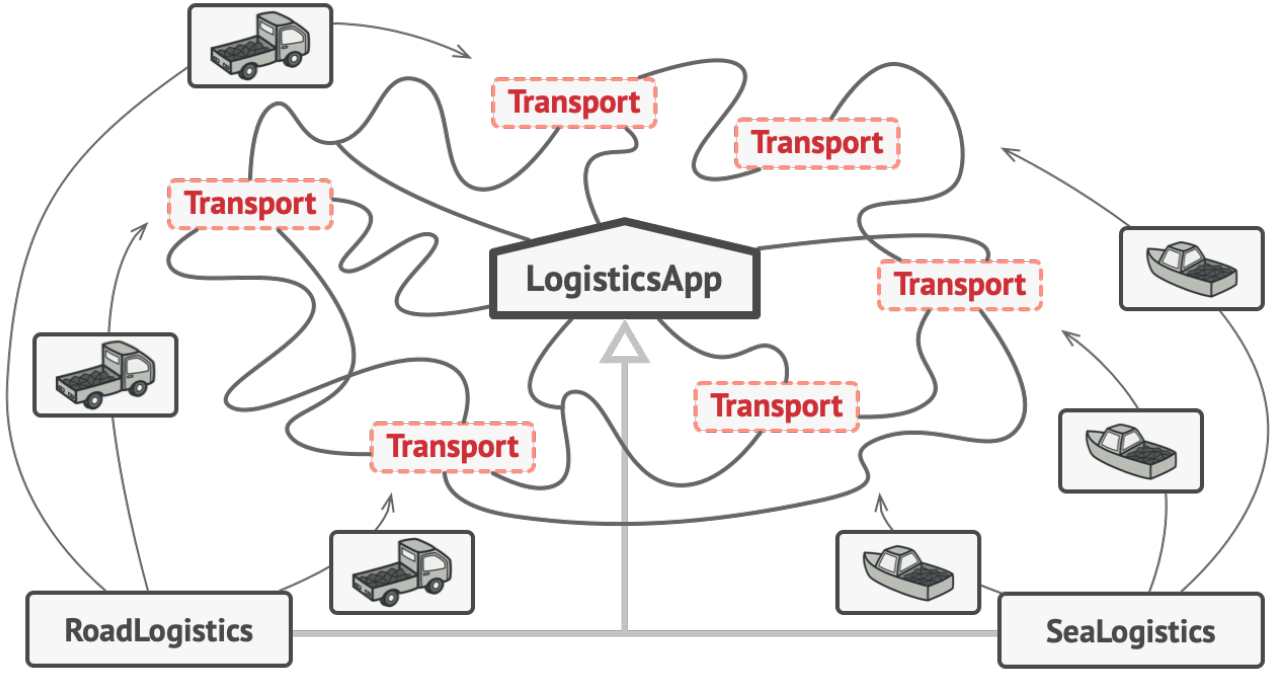
- Replace direct object construction calls (using the new operator) with calls to a special **factory method**
  - Objects returned are referred to as **products**
  - Override the factory method in a subclass, and **change** the class of products being created
  - All products must follow **the same base class or interface**



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/factory-method]

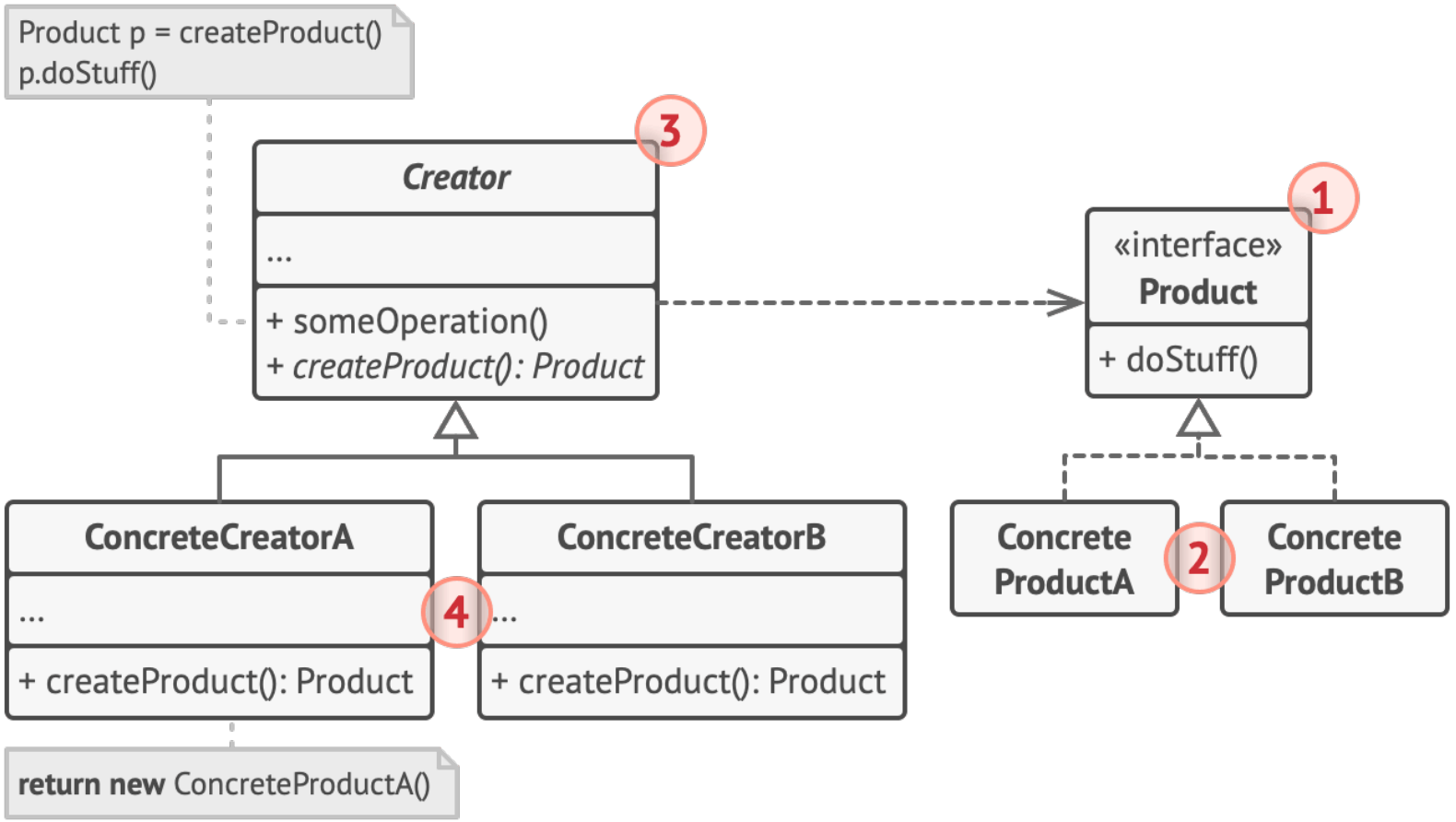
Factory Method: Solution (cont.)

- The client code does not see a difference between the actual products returned by various subclasses
  - It treats all products as **abstract *Transport***
  - It knows that all transport objects are supposed to have the ***deliver*** method, but does not care about how it exactly works



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/factory-method]

Factory Method: Structure

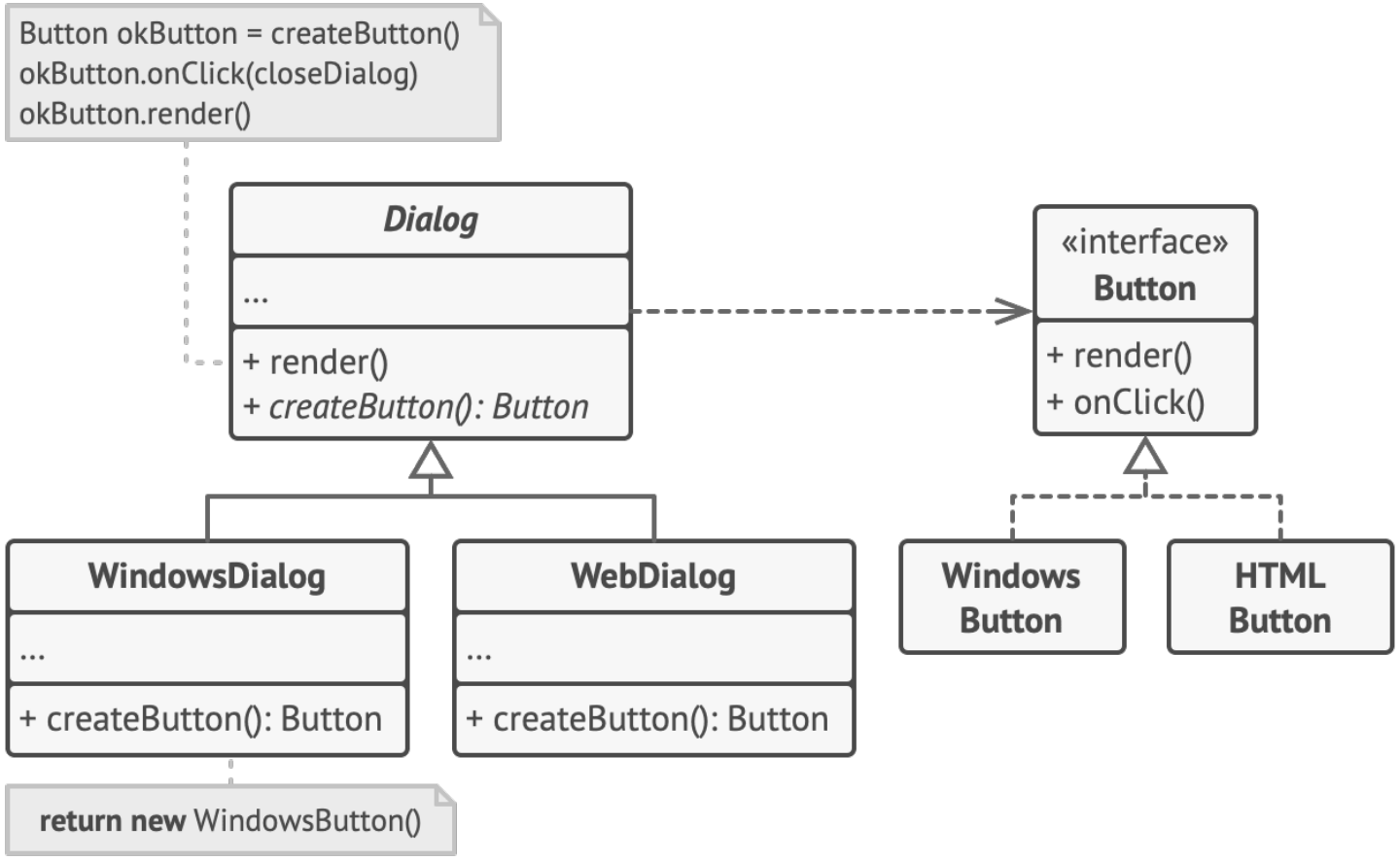


- **Note**
  - Product creation is **not** the primary responsibility of the creator
  - The factory method does not have to **create** new instances all the time

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/factory-method]

Factory Method: Example

- Creating cross-platform UI elements without coupling the client code to concrete UI classes



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/factory-method]

Factory Method: Applicability

- When you do not 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
  - To add a new product type, just create a new creator subclass
- When you want to provide users of your library or framework with a way to extend its internal components
  - Reduce the code that constructs components across the framework
  - Example: an app using an open source UI framework which provides square buttons
- When you want to save system resources by reusing existing objects instead of rebuilding them each time
  - Dealing with large, resource-intensive objects
  - A factory method that creates new objects and also reusing existing ones

Factory Method: Implementation

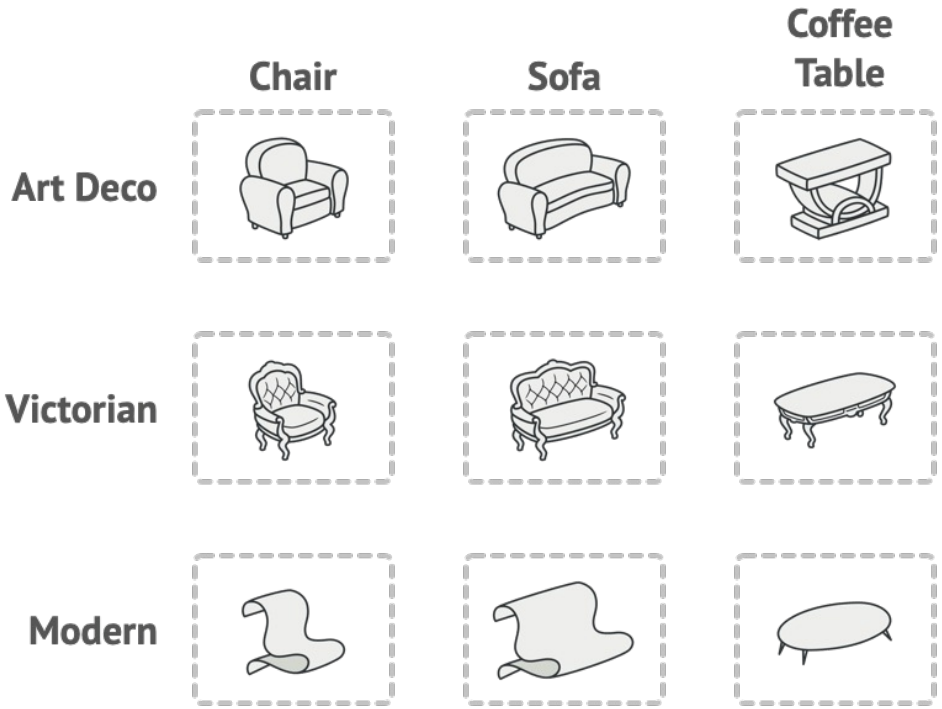
1. Make all products follow the same interface, which declares methods that make sense in every product
2. Add an empty factory method inside the creator class, with the return type matching the interface
3. Find all references to product constructors, and replace them with calls to the factory method, while extracting the product creation code into the factory method
   - You might need to add a temporary parameter to the factory method to control the type of returned product
4. Create a set of creator subclasses for each type of product, and override the factory method in the subclasses
5. If there are too many product types and it does not make sense to create subclasses for all of them, you can reuse the control parameter from the base class in subclasses
6. If the base factory method has become empty, make it abstract; if there is something left, make it a default behavior of the method

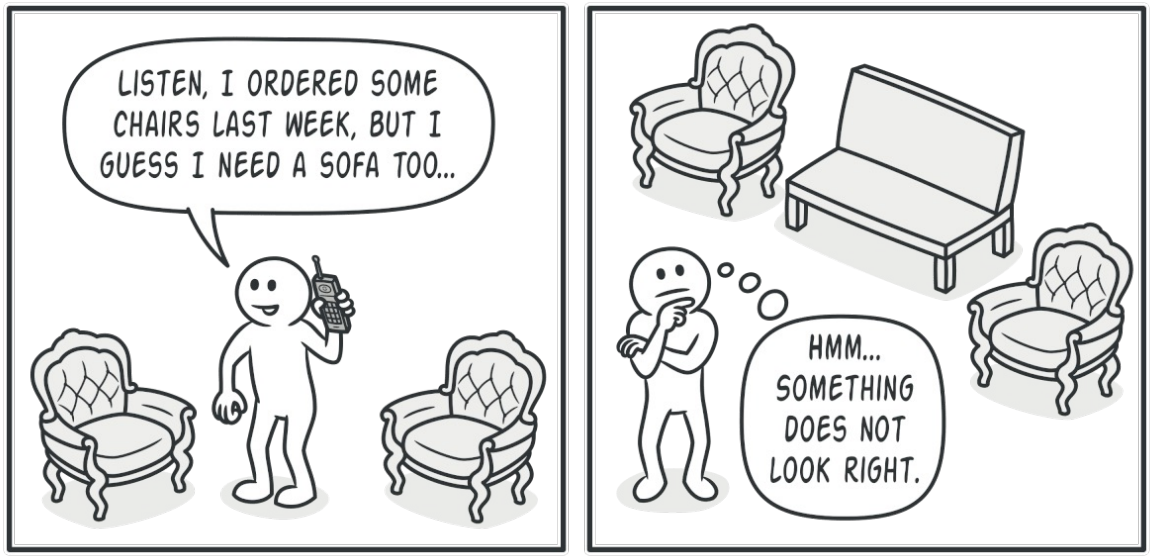
Factory Method: Pros and Cons

- **Pros**
  - Avoid tight coupling between the creator and the concrete products
  - Comply with the Single Responsibility Principle
  - Comply with the Open/Closed Principle
- **Cons**
  - Code may become complicated, since a lot of new subclasses are introduced

Abstract Factory: Problem

- **Example: a furniture shop simulator**
  - A family of related products: Chair + Sofa + CoffeeTable
  - Variants of this family: ArtDeco, Victorian, Modern

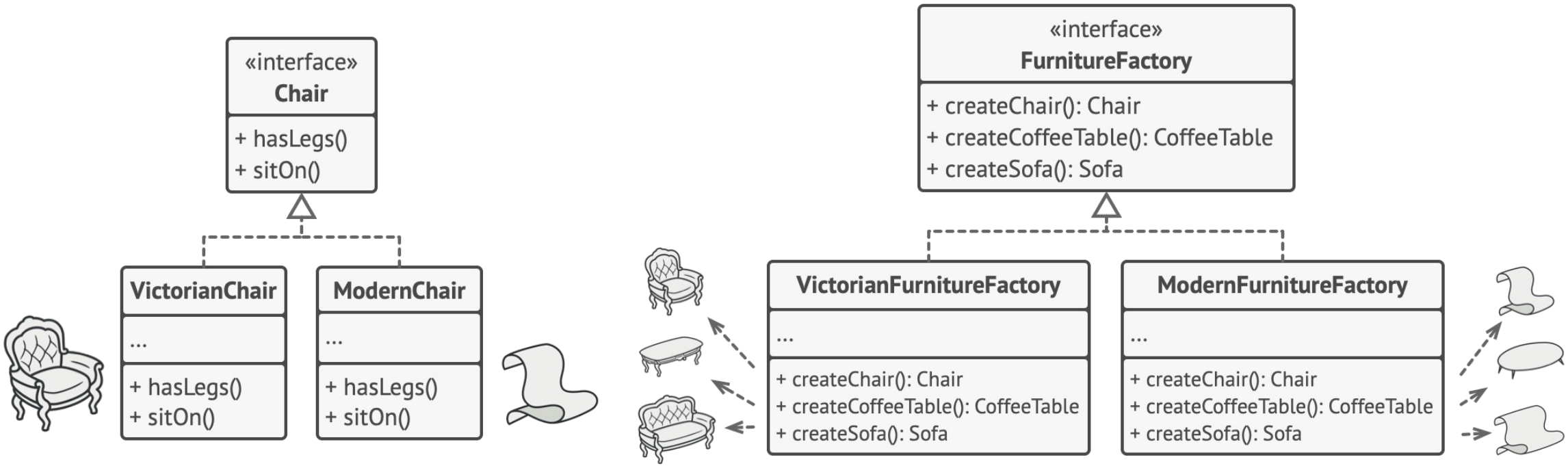




[Figures in this slide are extracted from https://refactoring.guru/design-patterns/abstract-factory]

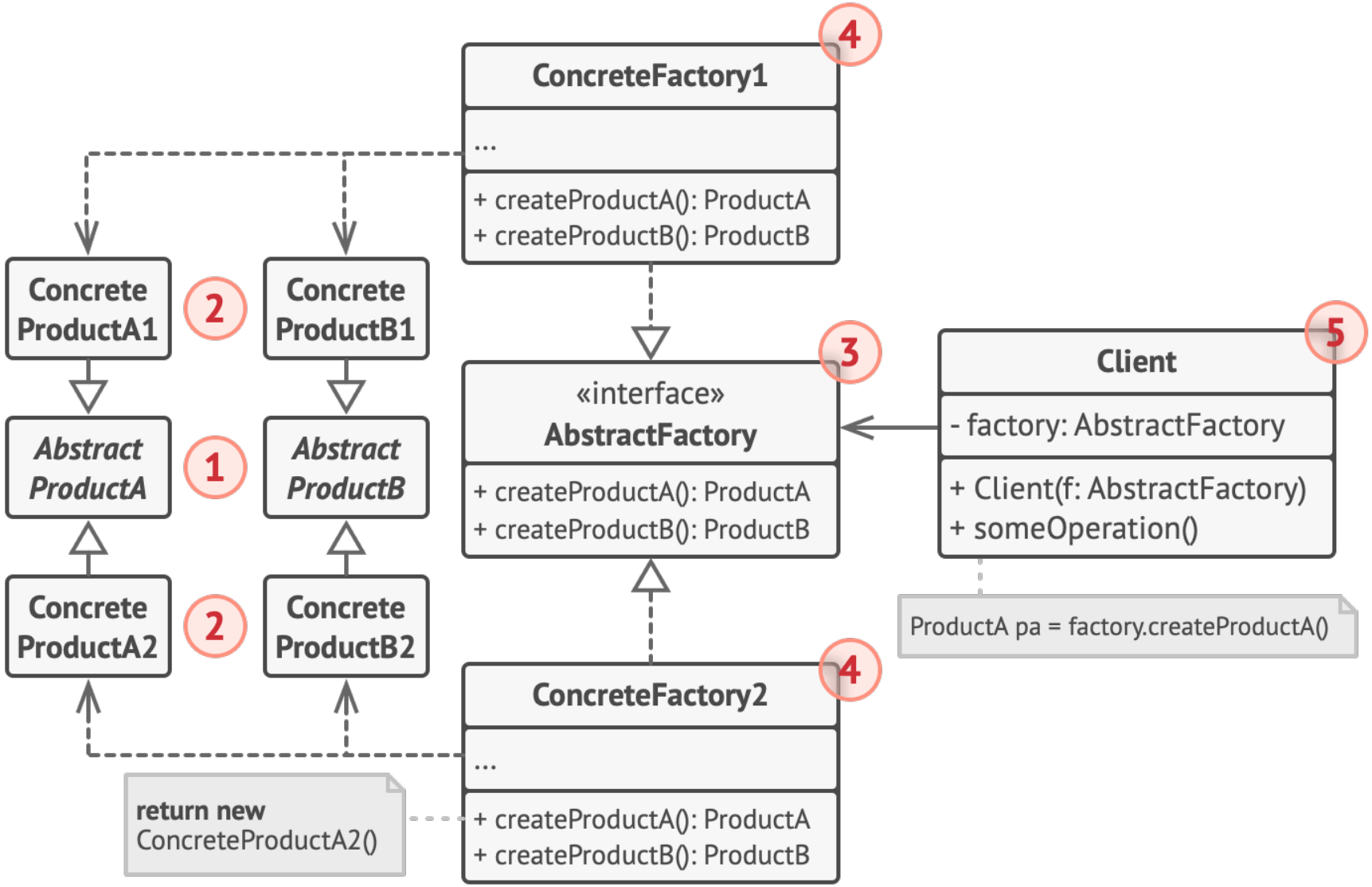
Abstract Factory: Solution

- **Firstly**, explicitly declare **interfaces for each distinct product** of the product family, and then make **all variants follow the interfaces**
- **Secondly**, declare the **Abstract Factory**: an interface with **a list of creation methods** for all products, and the methods **return abstract product types** represented by the interfaces
- **Thirdly**, create a **separate factory class for each variant**
- **Finally**, the client creates a concrete factory object at the **initialization stage**



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/abstract-factory]

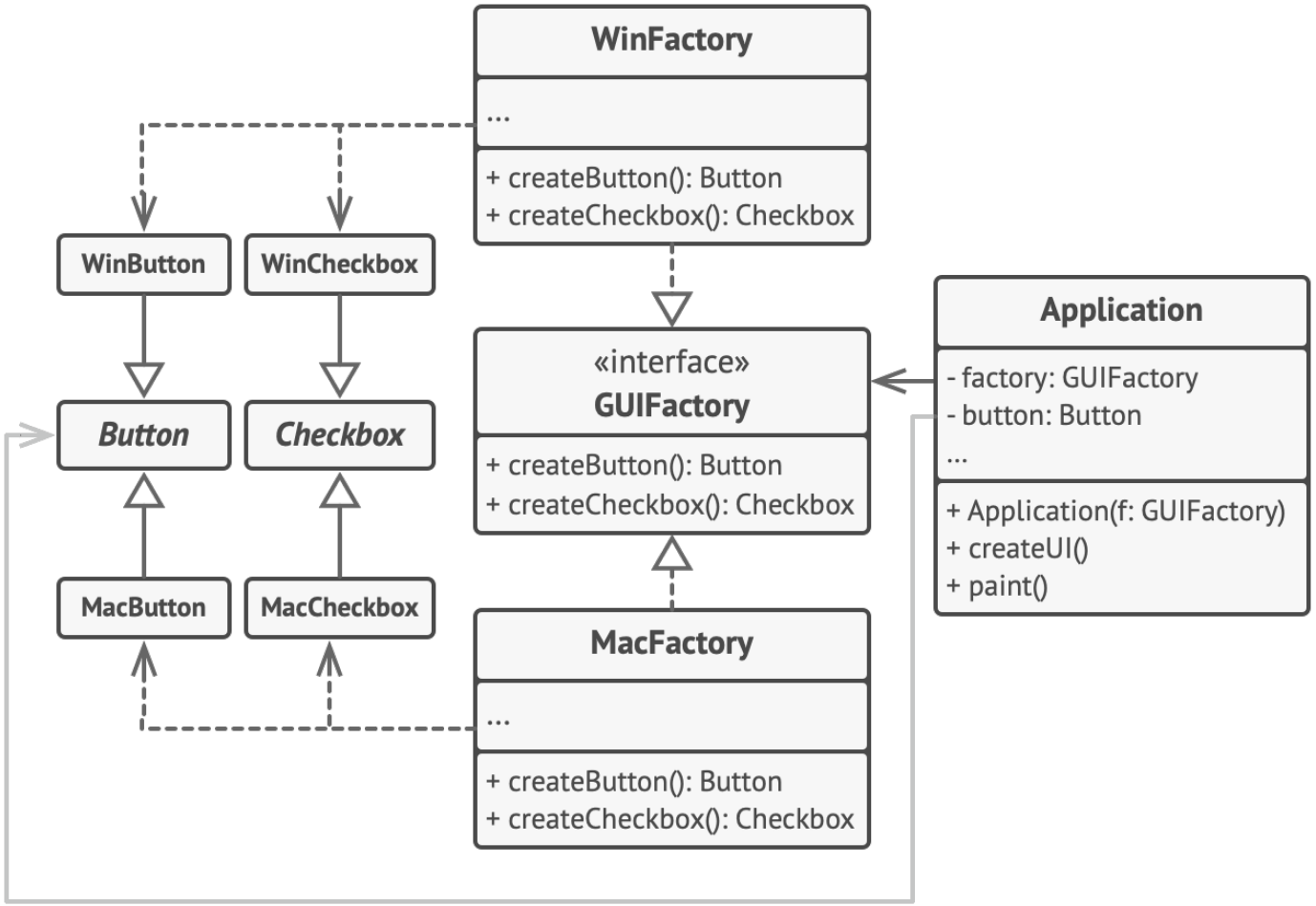
Abstract Factory: Structure



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/abstract-factory]

Abstract Factory: Example

- Creating cross-platform UI elements without coupling the client code to concrete UI classes, while keeping all created elements consistent with a selected operating system



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/abstract-factory]

Abstract Factory: Applicability

- When your code needs to work with various families of related products, but you do not want it to depend on the concrete classes of those products
  - They might be unknown beforehand
  - You simply want to allow for future extensibility
- When you have a class with a set of Factory Methods that blur its primary responsibility
  - Each class should be responsible only for one thing

Abstract Factory: Implementation

1. Map out **a matrix** of distinct **product types** versus **variants** of these products
2. Declare **abstract product interfaces** for all product types, and make all concrete product classes implement these interfaces
3. Declare the **abstract factory interface** with a set of creation methods for all abstract products
4. Implement a set of **concrete factory classes**, one for each product variant
5. Create **factory initialization** code somewhere in the app, instantiating one of the concrete factory classes
6. Scan through the code and find all direct calls to product constructors, and replace them with calls to creation methods on the factory object

Abstract Factory: Pros and Cons

- **Pros**
  - The products from one factory are compatible with each other
  - Avoid tight coupling between concrete products and client code
  - Comply with the Single Responsibility Principle
  - Comply with the Open/Closed Principle
- **Cons**
  - The code may become more complicated, since a lot of new interfaces and classes are introduced

Comparison on Factory-Related Terms

- **Factory**
  - An ambiguous term that stands for a function, method or class that produces objects, files, records, etc.
- **Creation method**, defined in *Refactoring To Patterns*
  - A method that creates objects: a wrapper around a constructor call
- **Static creation method**
  - Called on a class, which can no longer be extended
- **Simple factory pattern**, defined in *Head First Design Patterns*
  - A class that has one creation method, based on method parameters choosing which product class to instantiate and return
  - An intermediate step of introducing Factory Method or Abstract Factory patterns
- **Factory Method pattern** (introduced in this lecture)
- **Abstract Factory pattern** (introduced in this lecture)