DEPAUL UNIVERSITY

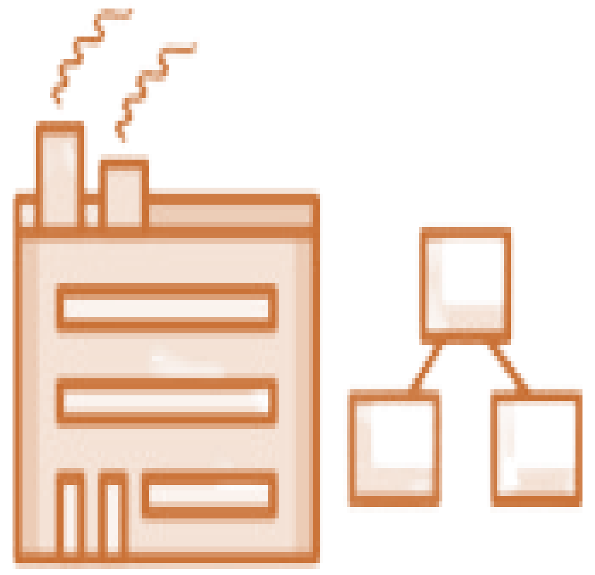
Design Patterns:

Factory Method, Abstract Factory

Object-oriented Software Development
SE 350– Spring 2021

Vahid Alizadeh

Week 8.1
May 18, 2021

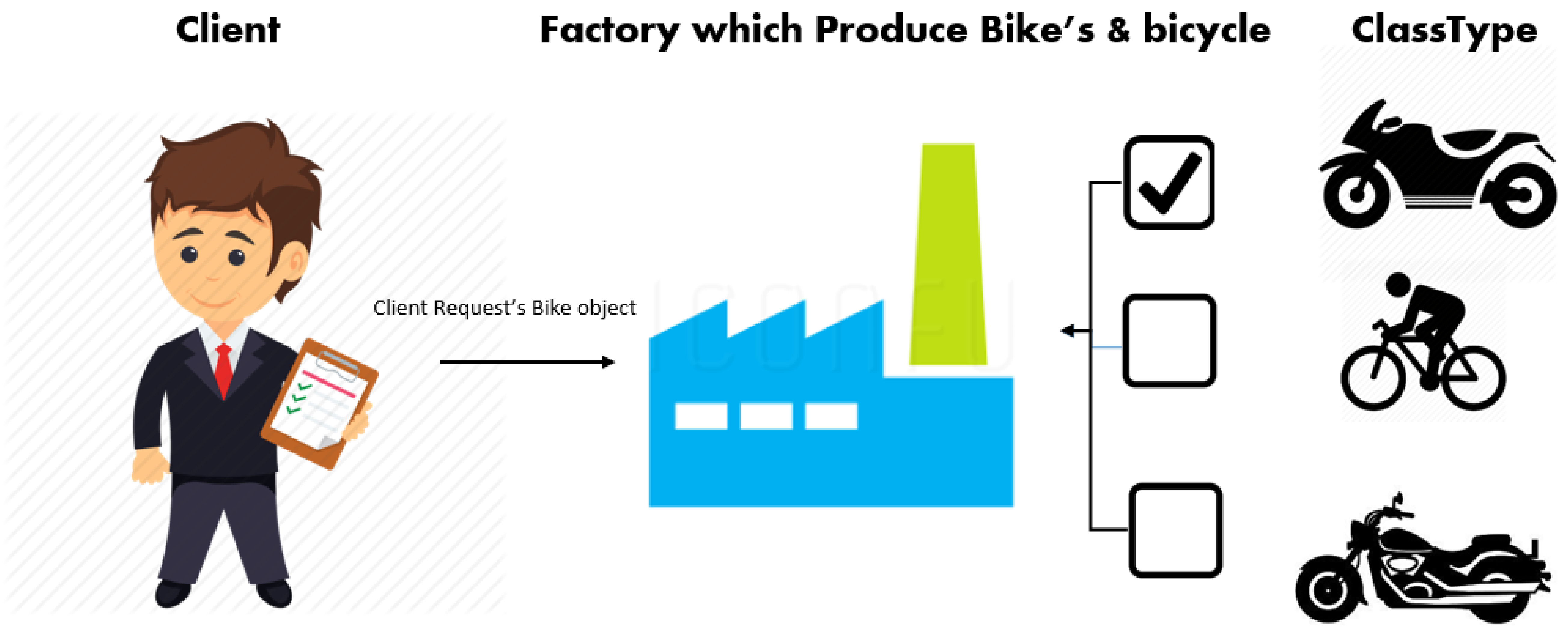


Factory Method

CREATIONAL

Factory Method Pattern Introduction

Factory Method is a creational design pattern that that provides an interface for creating objects and defer instantiation to subclasses.


Client
Factory which Produce Bike's & bicycle
ClassType
Client Request's Bike object

Factory Method Design Pattern

INTENT

- Define an interface for creating an object.
- Defining a "virtual" constructor.
- The new operator considered harmful.

PROBLEM

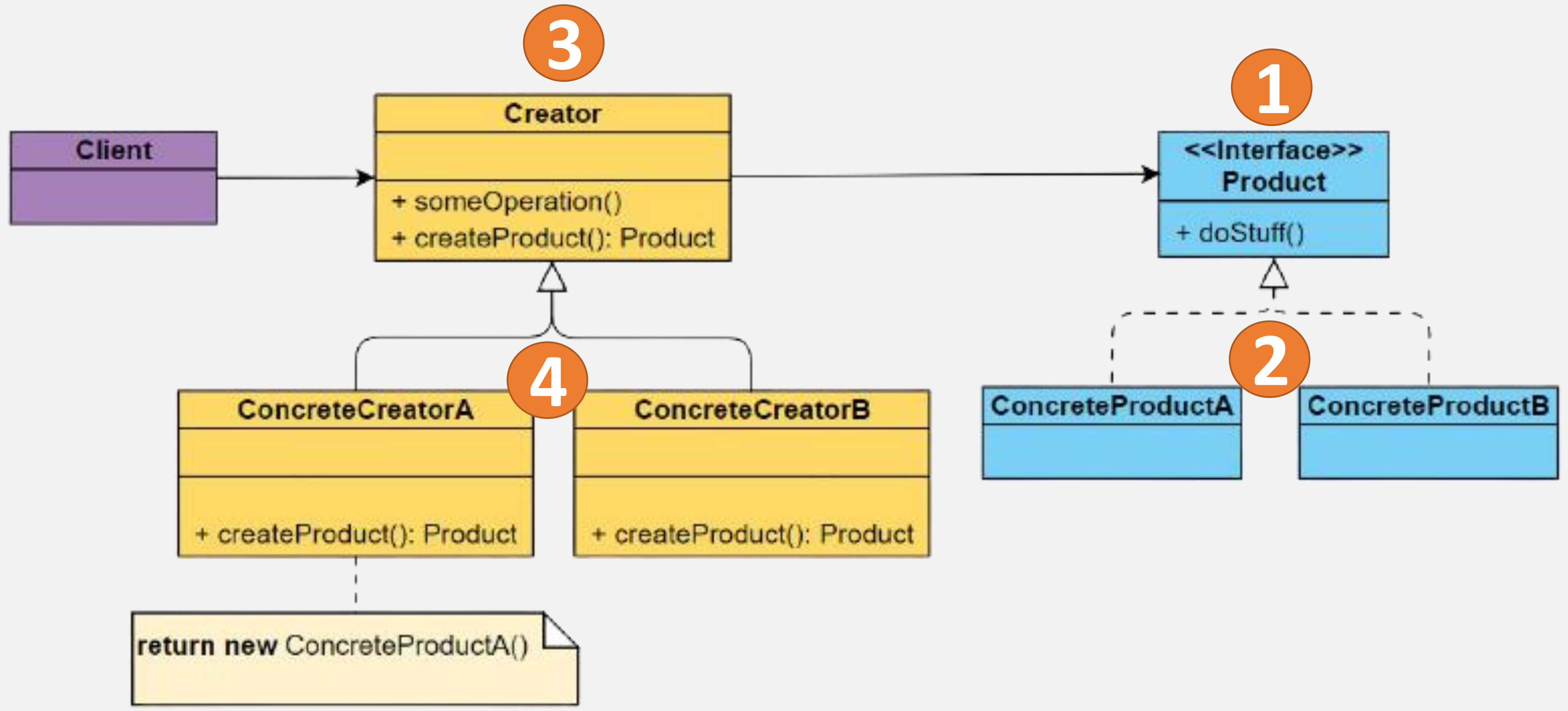
- A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects

STRUCTURE

- 1- Product interface
- 2- Concrete Products
- 3- Creator class with the factory method
- 4- Concrete creators

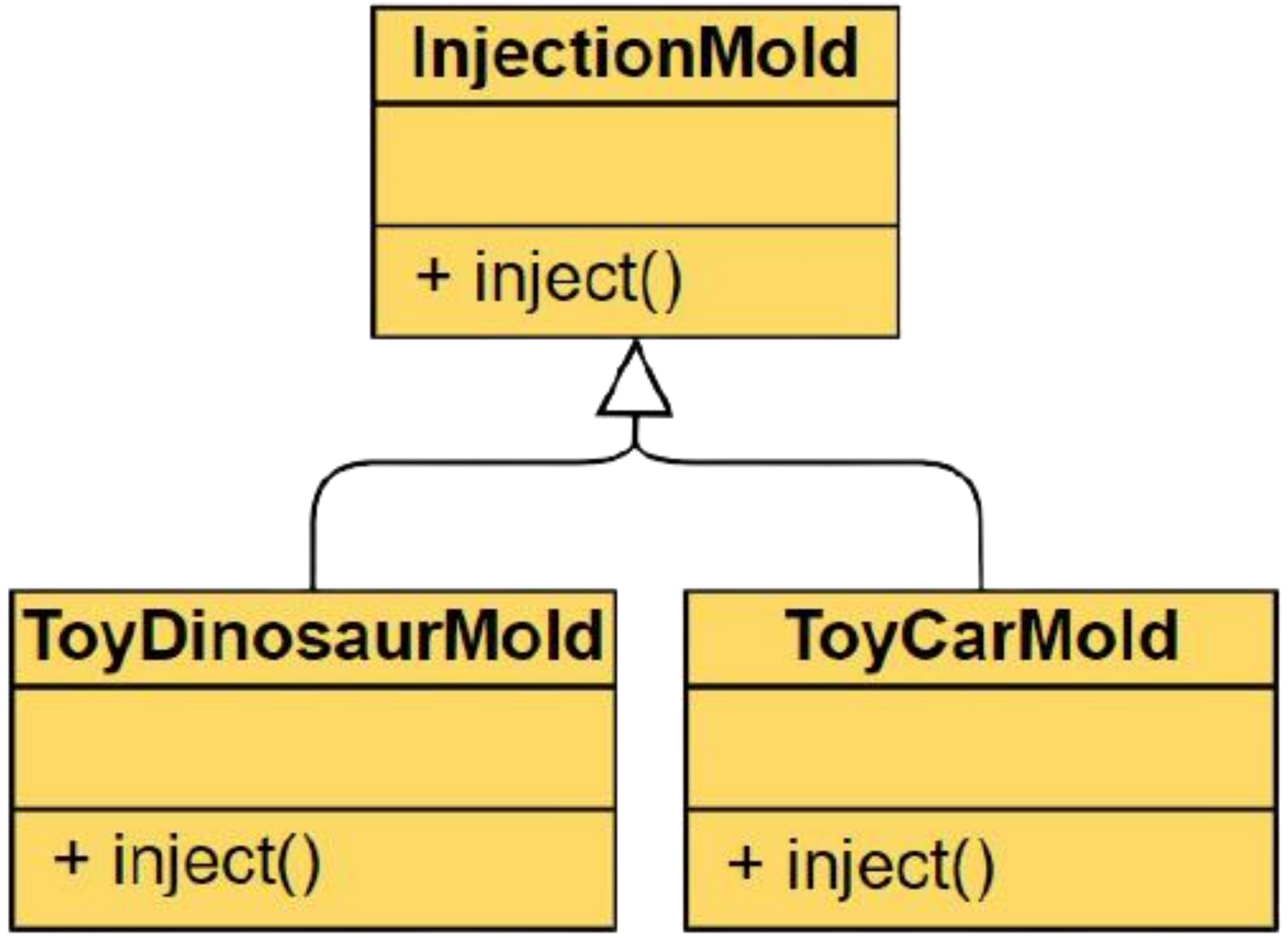
Factory method vs. Constructor:

- Instance of subclass / reusing existing object / different name

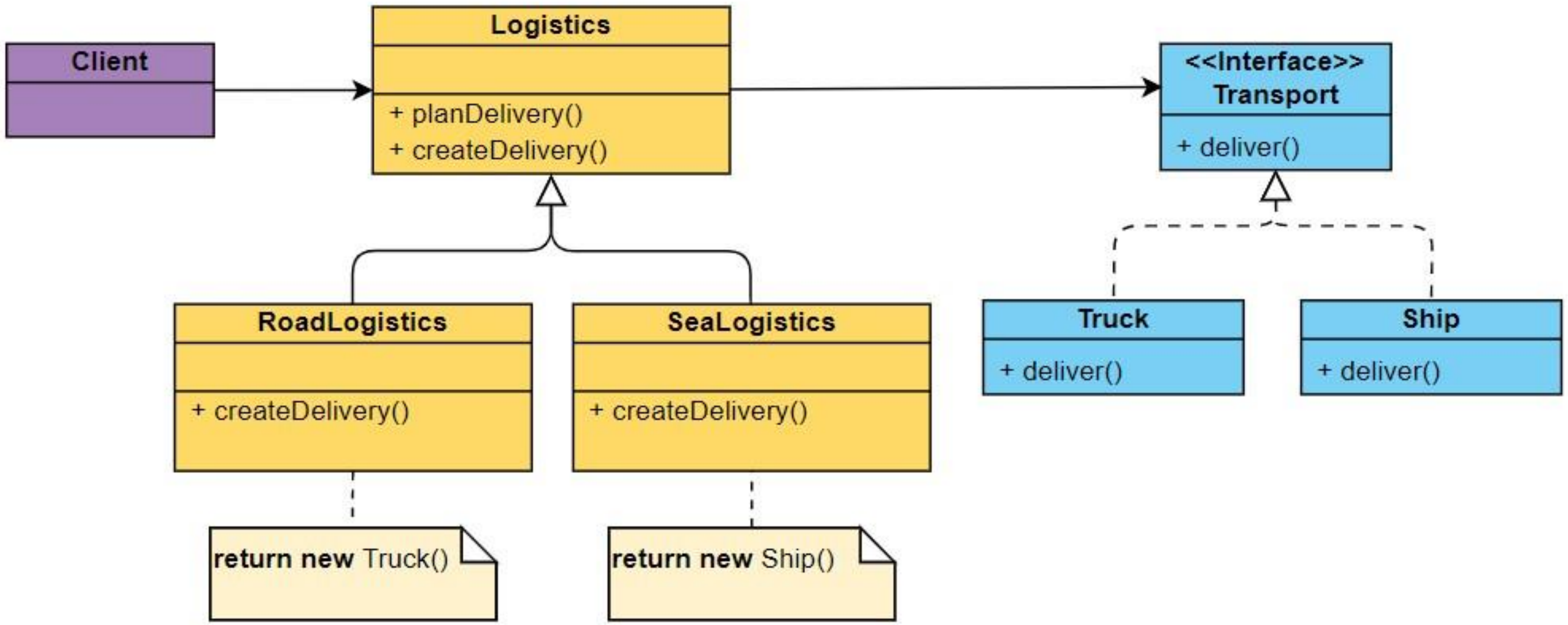


Factory Method Real-world Example

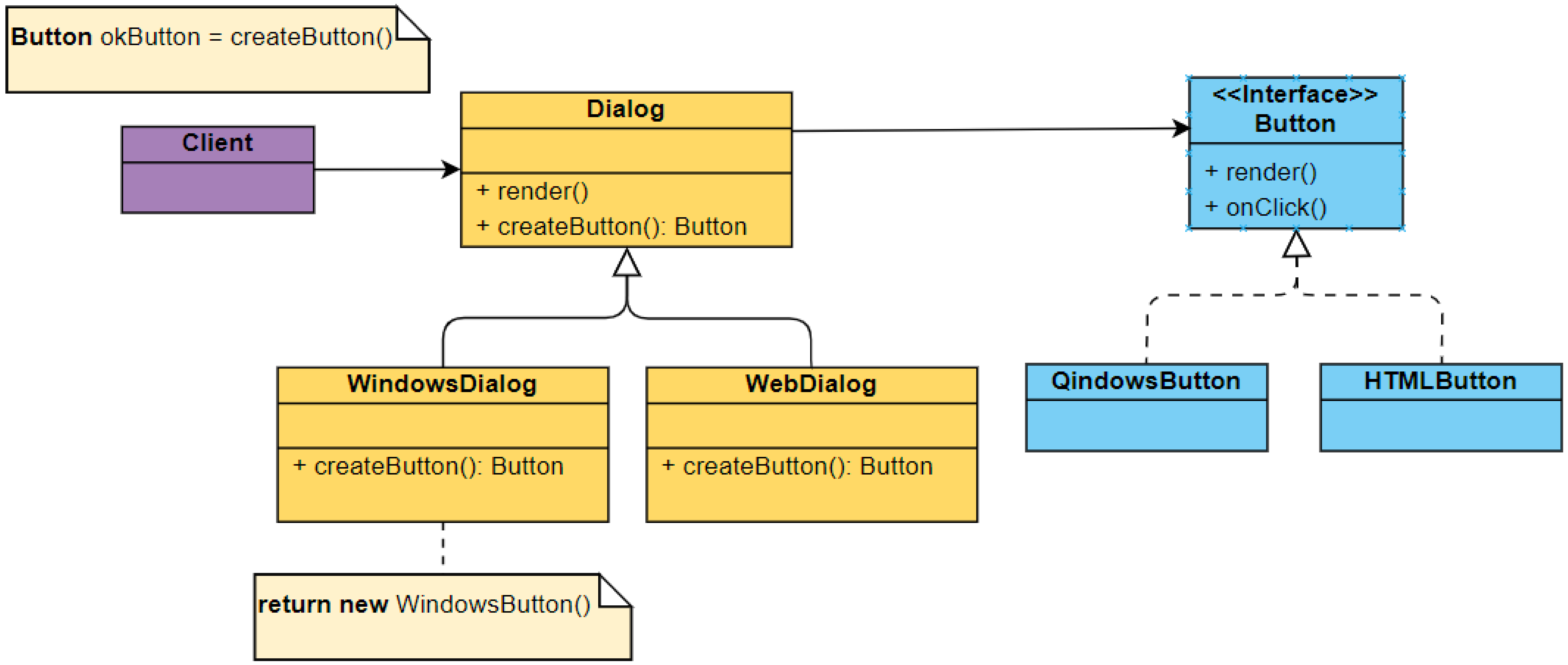
- Manufacturers of plastic toys.
- Inject the plastic into molds of the desired shapes.



Factory Method Use Case Example: Logistics Management App



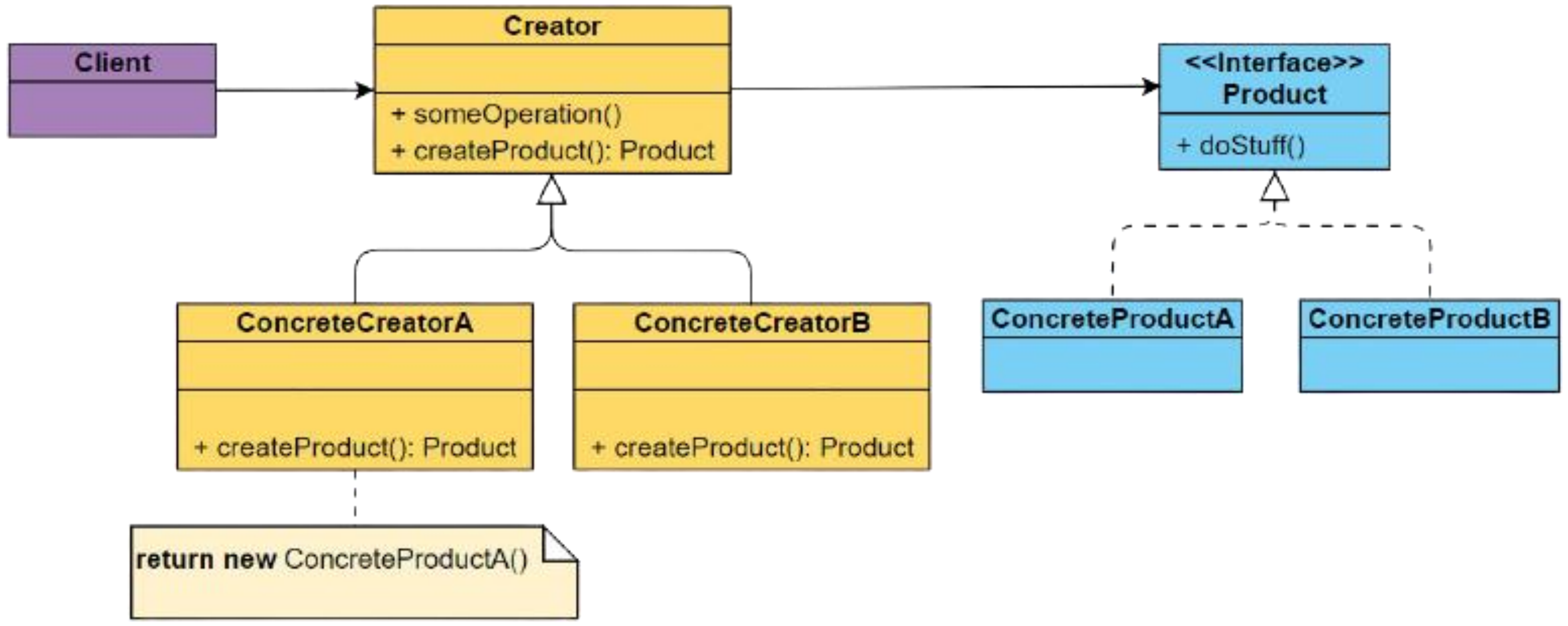
Factory Method Use Case Example: Cross-platform UI Elements



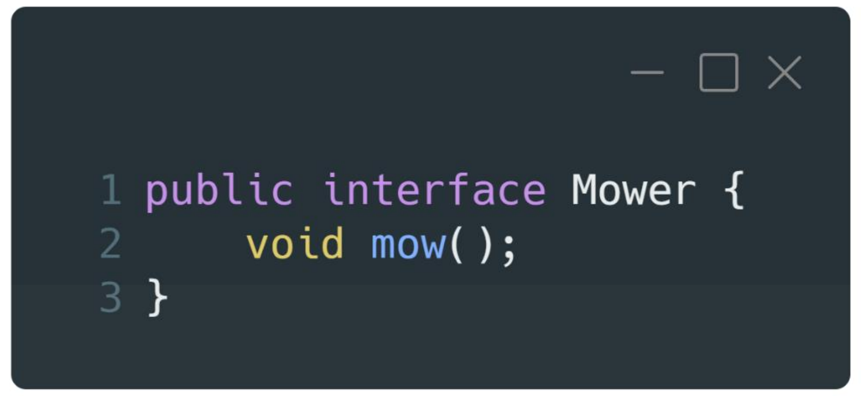
Factory Method Implementation

- **Common Implementation Steps:**
  - All products implement the same interface.
  - Factory method inside Creator class with product interface return type.
  - Replace references to product constructor with calls to the factory method.
  - Add Creator subclasses for each type of product and override their factory methods.
  - If there are many product types, reuse the control parameter from the base class.
  - Now, you can make the base factory method abstract since it is empty.

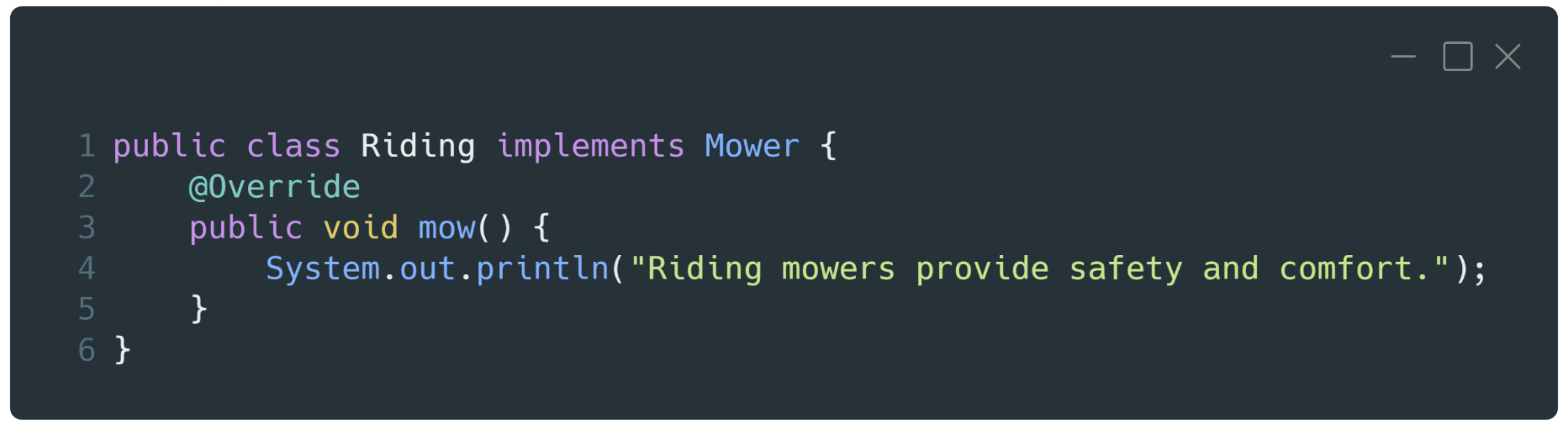
- **Check list:**
  - If you have inheritance hierarchy, you can add a polymorphic creation capability via static factory method.
  - Design factory methods' arguments.
  - Consider designing an internal "object pool" to allow reusing objects.
  - Consider making all constructors private.



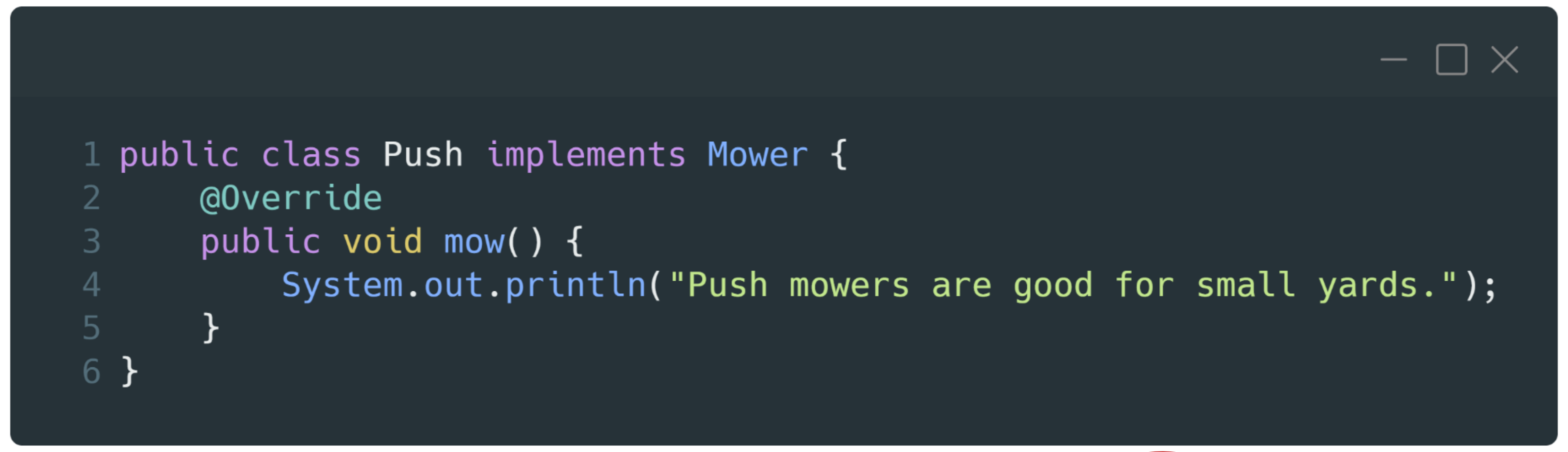
Factory Method Use Case Example: Mower Selection



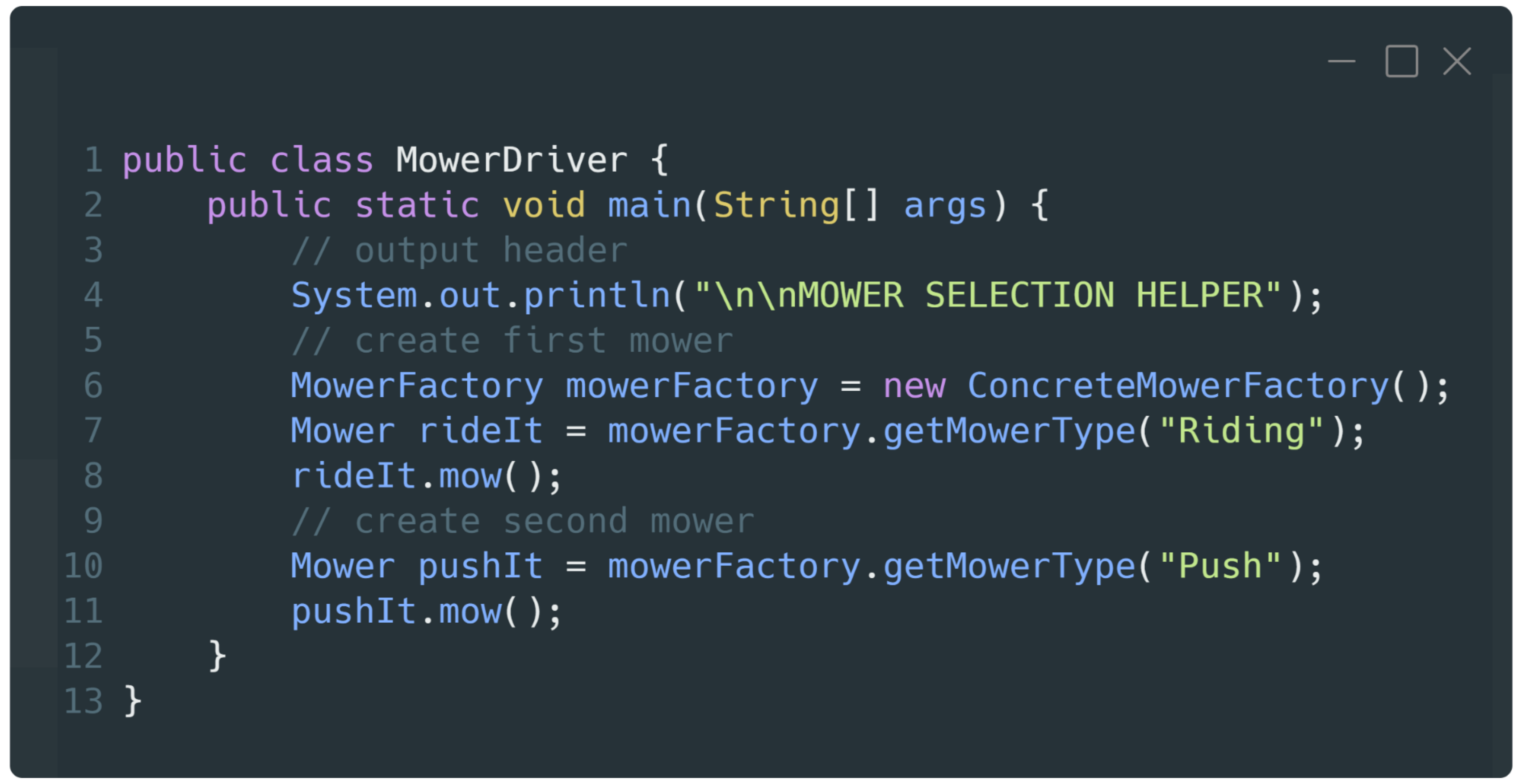
```java
public interface Mower {
    void mow();
}
```



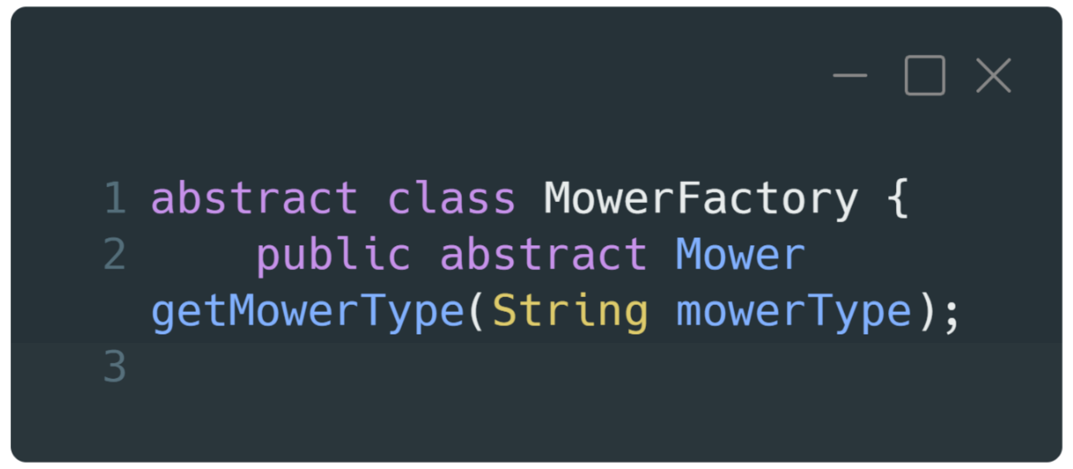
```java
public class Riding implements Mower {
    @Override
    public void mow() {
        System.out.println("Riding mowers provide safety and comfort.");
    }
}
```



```java
public class Push implements Mower {
    @Override
    public void mow() {
        System.out.println("Push mowers are good for small yards.");
    }
}
```

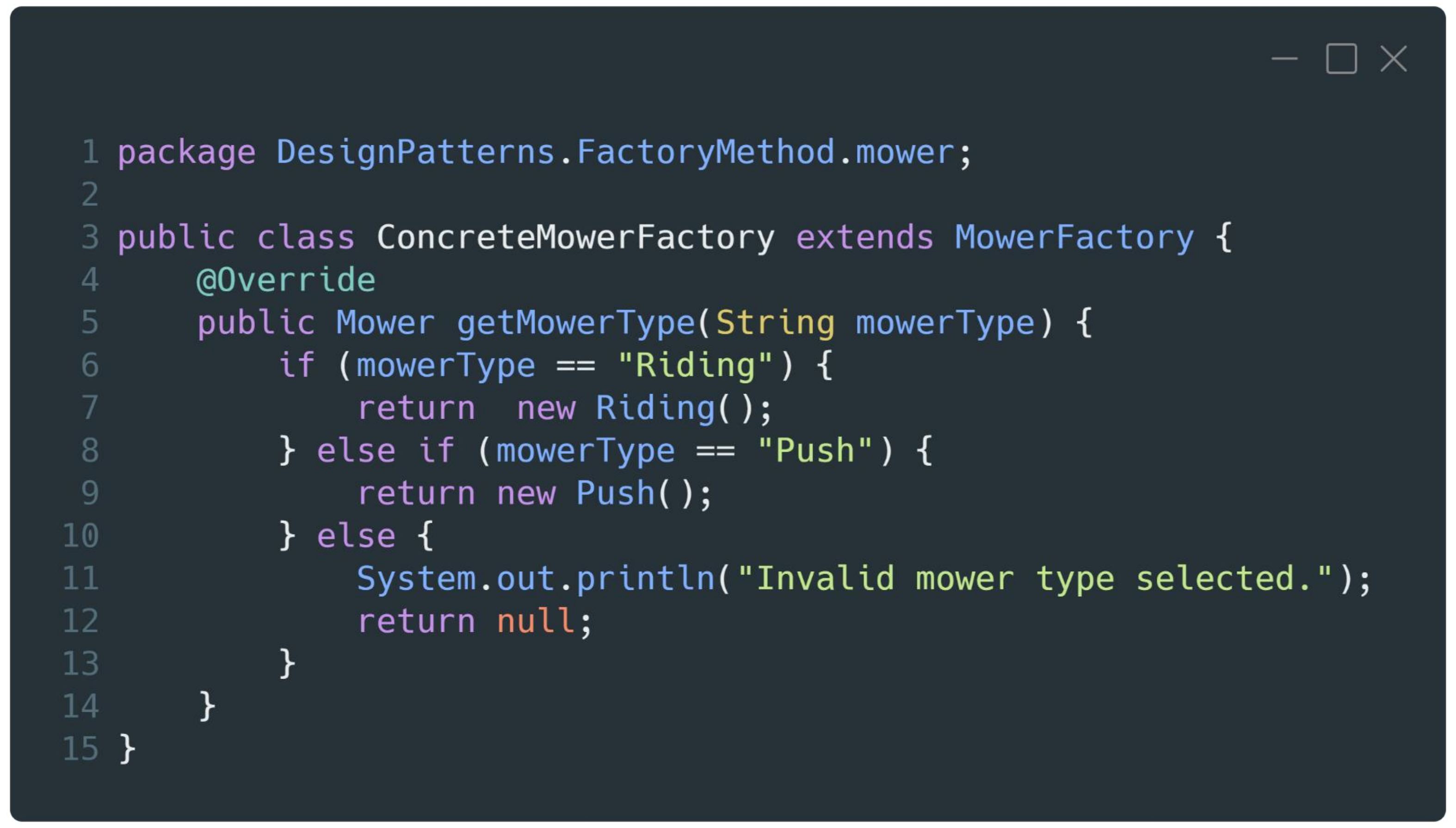


```java
public class MowerDriver {
    public static void main(String[] args) {
        // output header
        System.out.println("\n\nMOWER SELECTION HELPER");
        // create first mower
        MowerFactory mowerFactory = new ConcreteMowerFactory();
        Mower rideIt = mowerFactory.getMowerType("Riding");
        rideIt.mow();
        // create second mower
        Mower pushIt = mowerFactory.getMowerType("Push");
        pushIt.mow();
    }
}
```



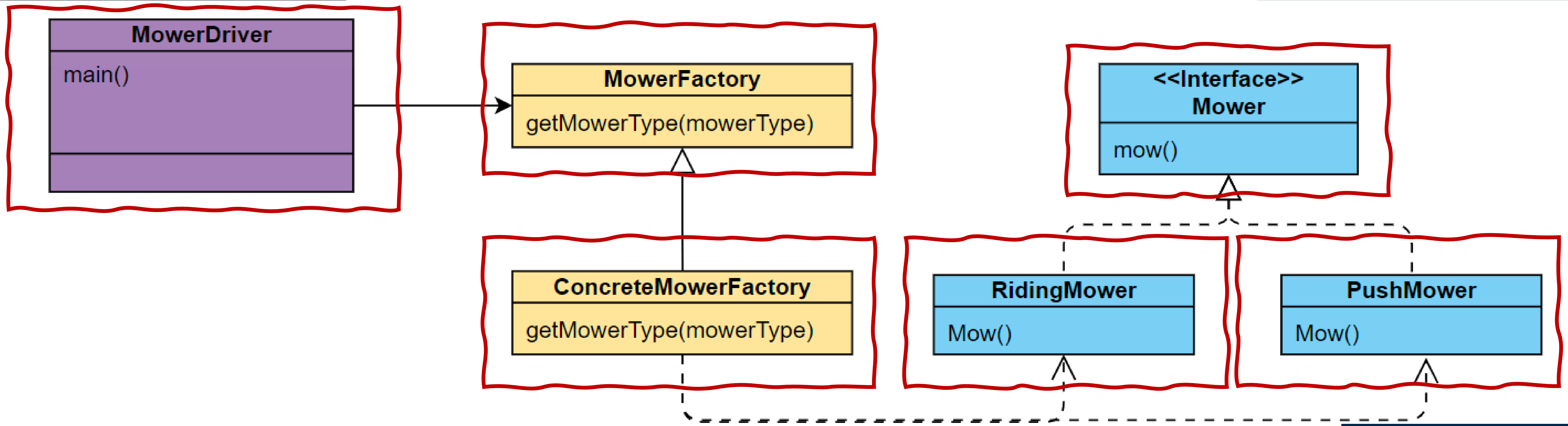
```java
abstract class MowerFactory {
    public abstract Mower
getMowerType(String mowerType);

```



```java
package DesignPatterns.FactoryMethod.mower;

public class ConcreteMowerFactory extends MowerFactory {
    @Override
    public Mower getMowerType(String mowerType) {
        if (mowerType == "Riding") {
            return  new Riding();
        } else if (mowerType == "Push") {
            return new Push();
        } else {
            System.out.println("Invalid mower type selected.");
            return null;
        }
    }
}
```



Find the source codes in the course GitHub repository.

Factory Method Pattern Pros & Cons

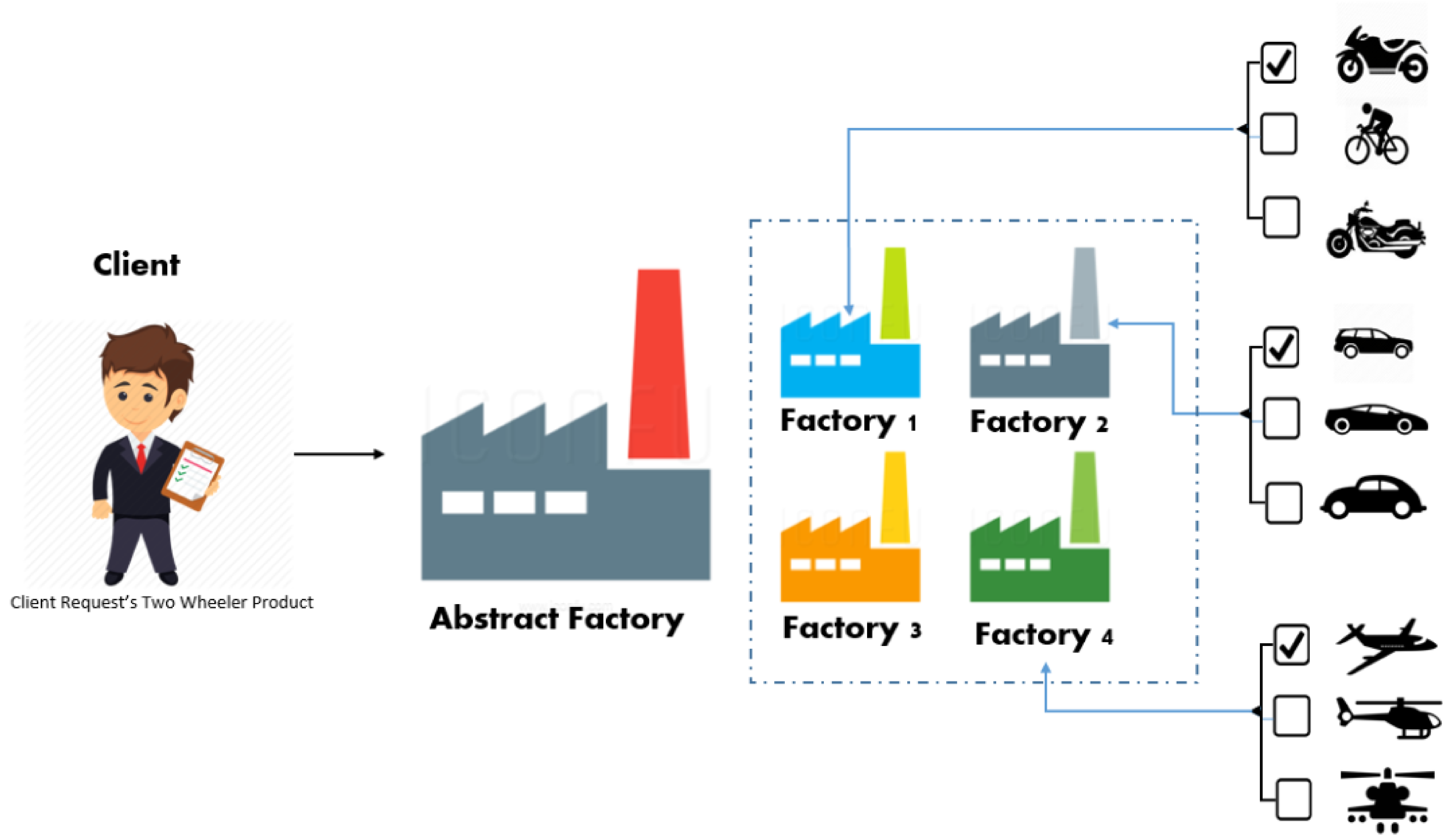
Pros	Cons
✓ Avoiding tight coupling.	✗ The code may become more complicated because of introducing many new subclasses.
✓ Single Responsibility Principle.	
✓ Open/Closed Principle.	

Abstract Factory Pattern

CREATIONAL

Abstract Factory Pattern Introduction

Abstract Factory is a creational design pattern that lets you produce families of related objects.



Abstract Factory Design Pattern

INTENT

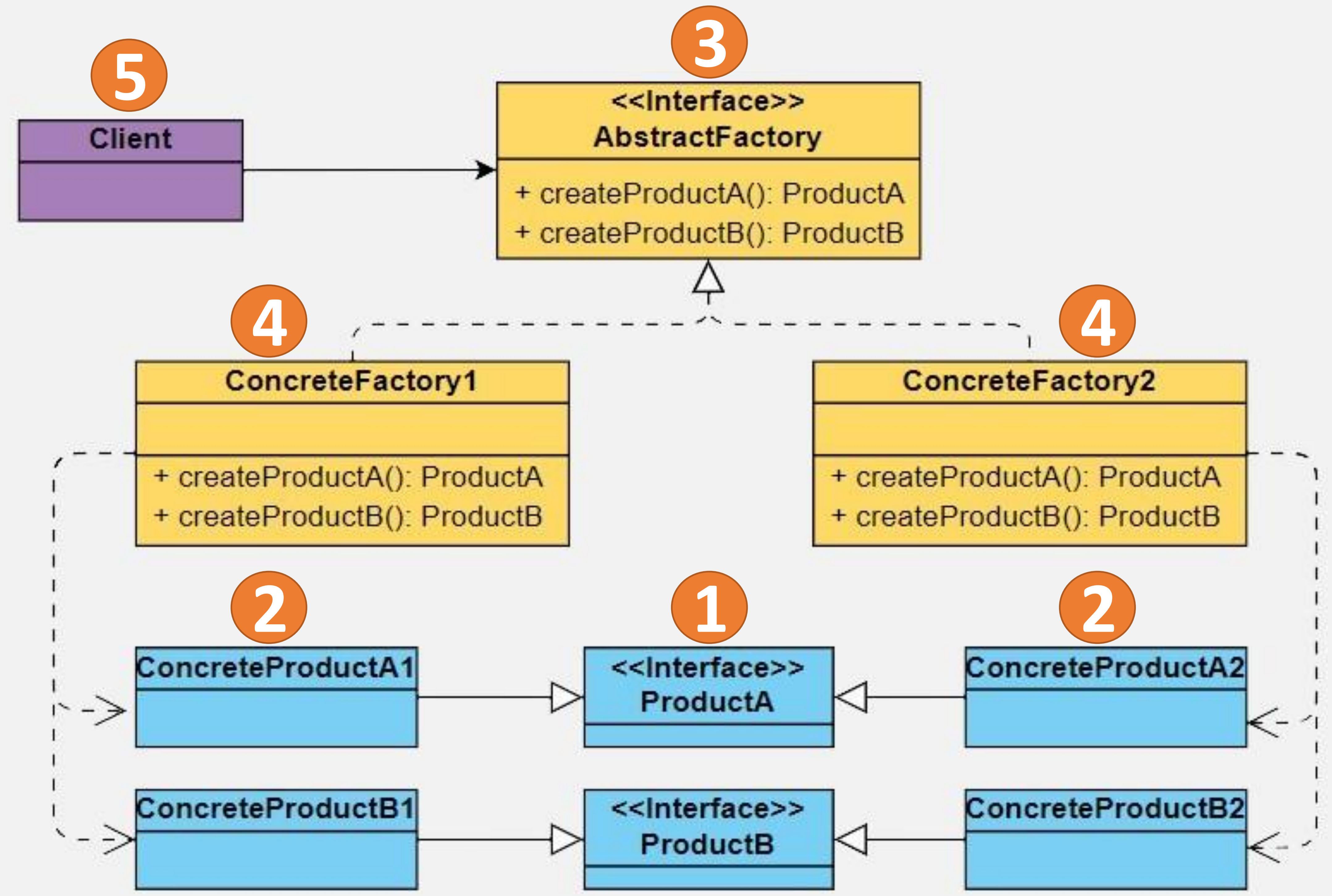
- Define an interface for creating families of related objects.
- A hierarchy that encapsulates many possible "platforms“.
- The new operator considered harmful.

PROBLEM

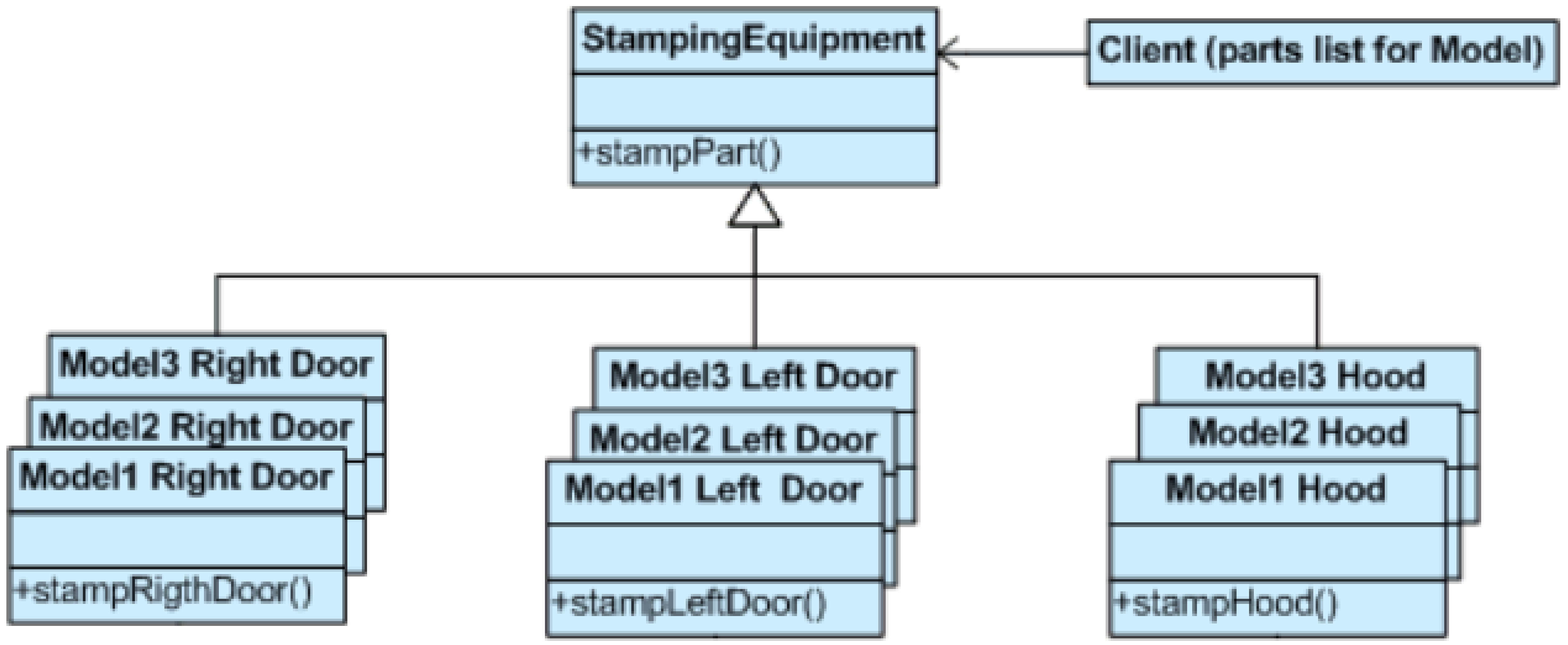
- If an application is to be portable, it needs to encapsulate platform dependencies. (ex. windowing system, operating system, database, etc.)
- This encapsulation is not engineered in advance.

STRUCTURE

- 1- Abstract products
- 2- Concrete Products
- 3- Abstract Factory interface
- 4- Concrete Factories implement creation methods
- 5- The Client



Abstract Factory: Real-world Example

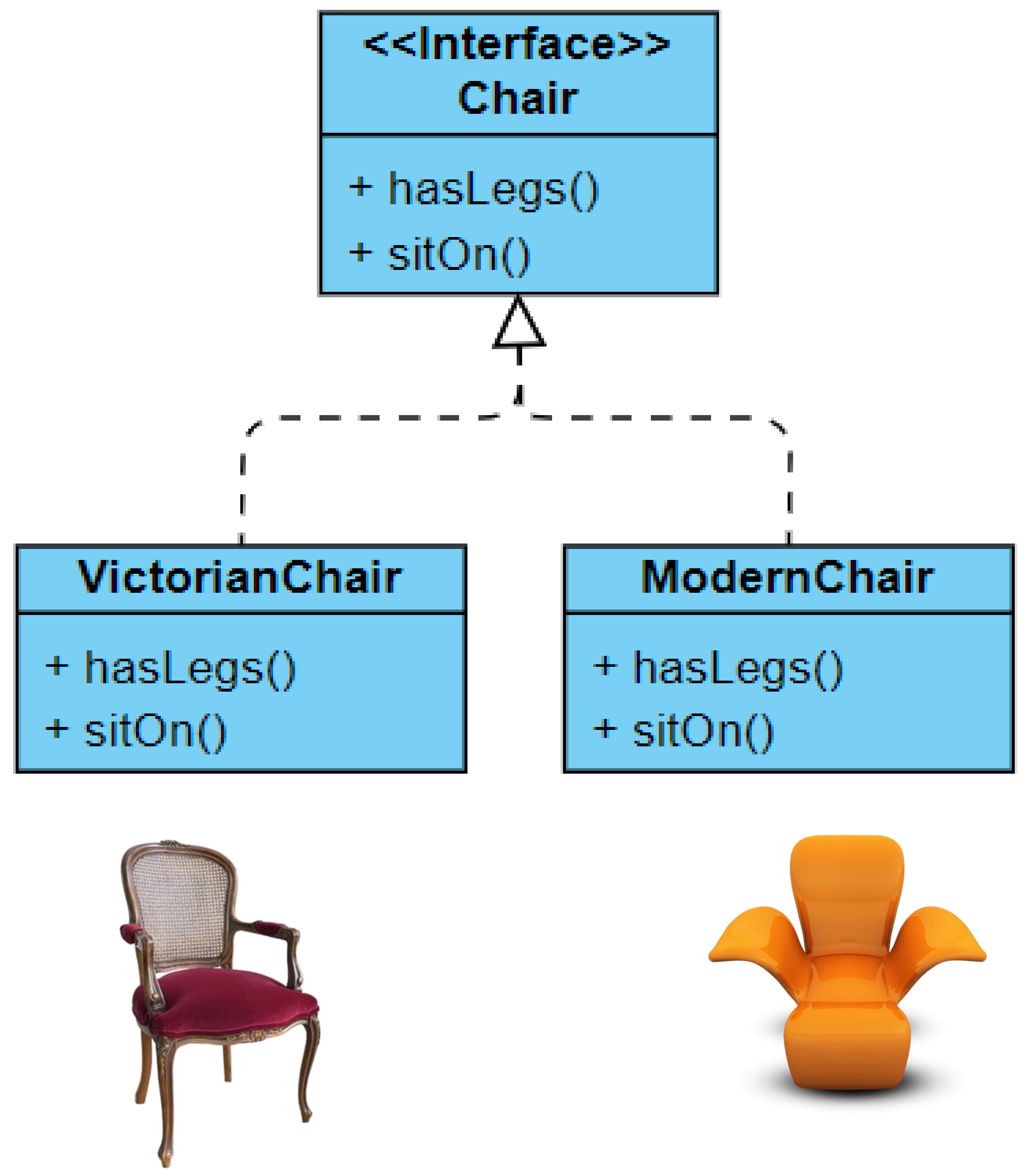


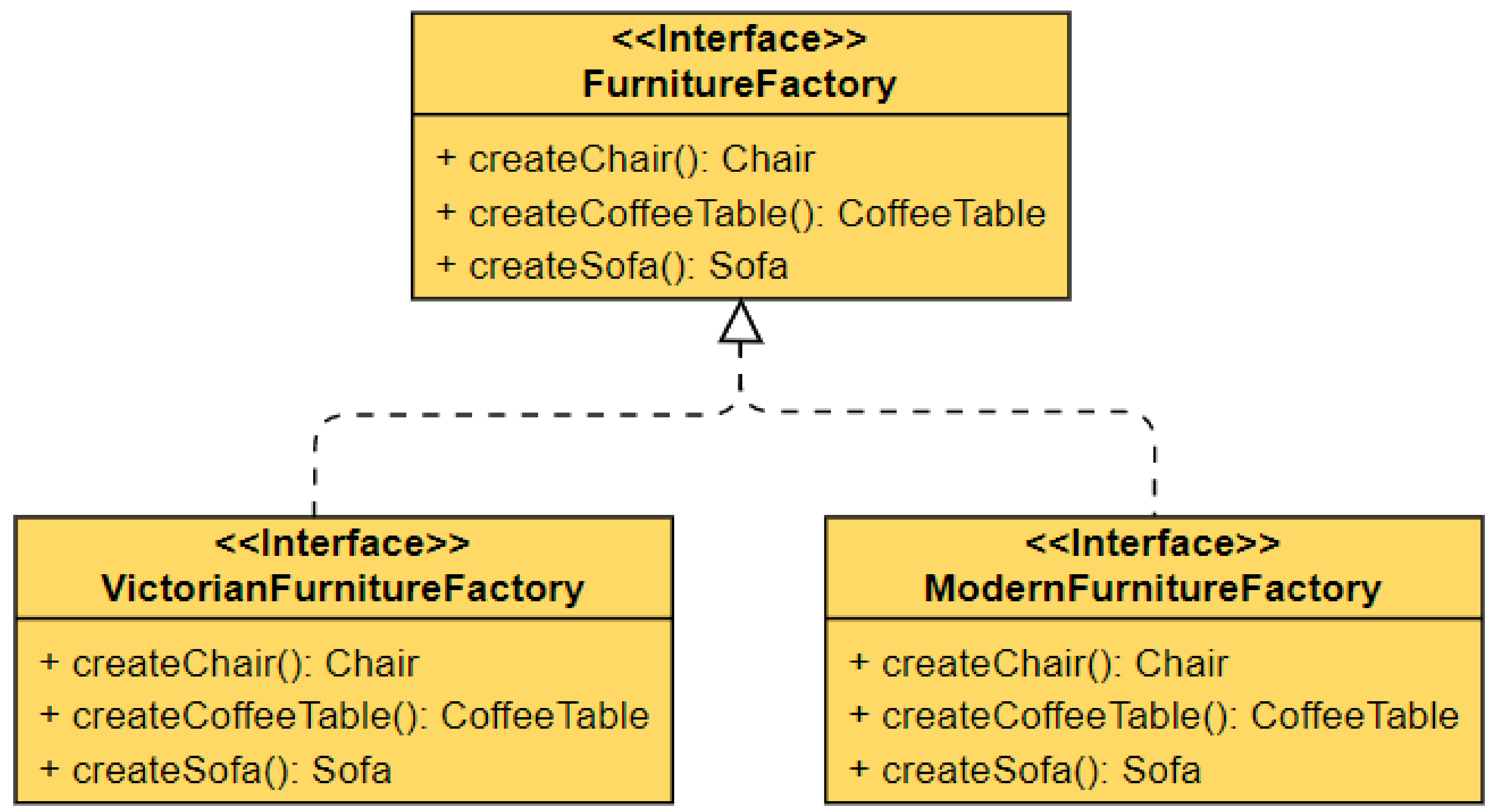
Model 1

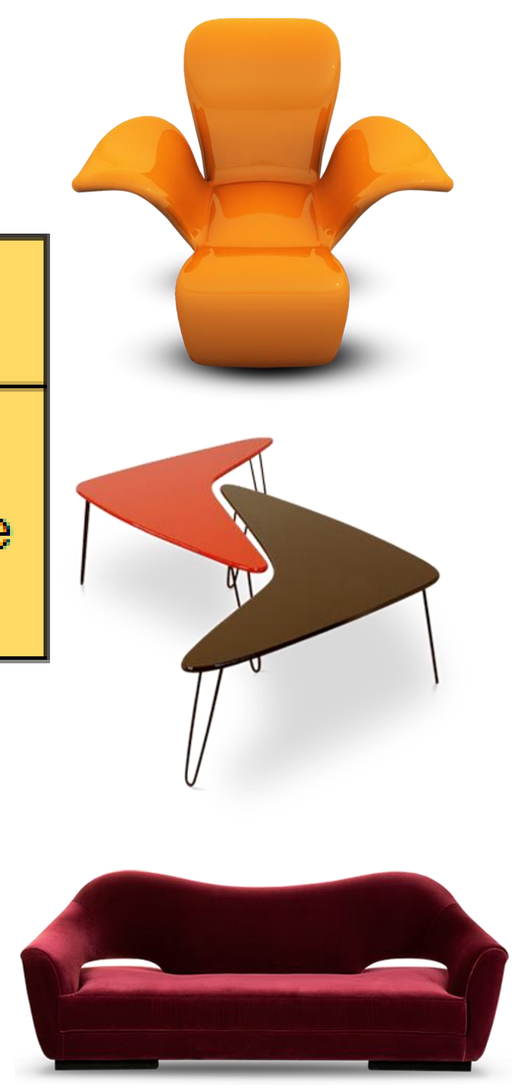
Model 2

Model 3

Abstract Factory Use Case Example: Furniture Shop App



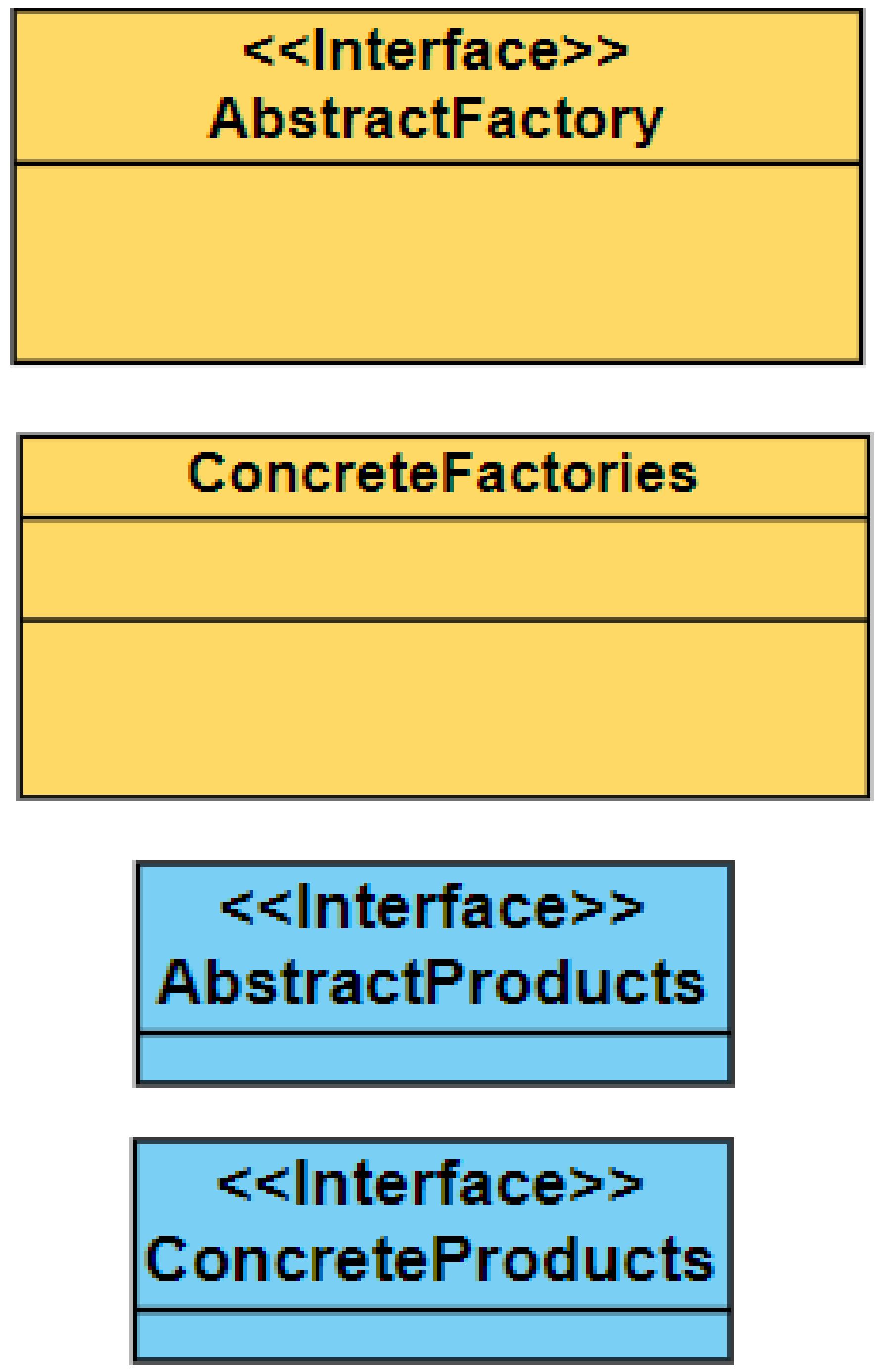




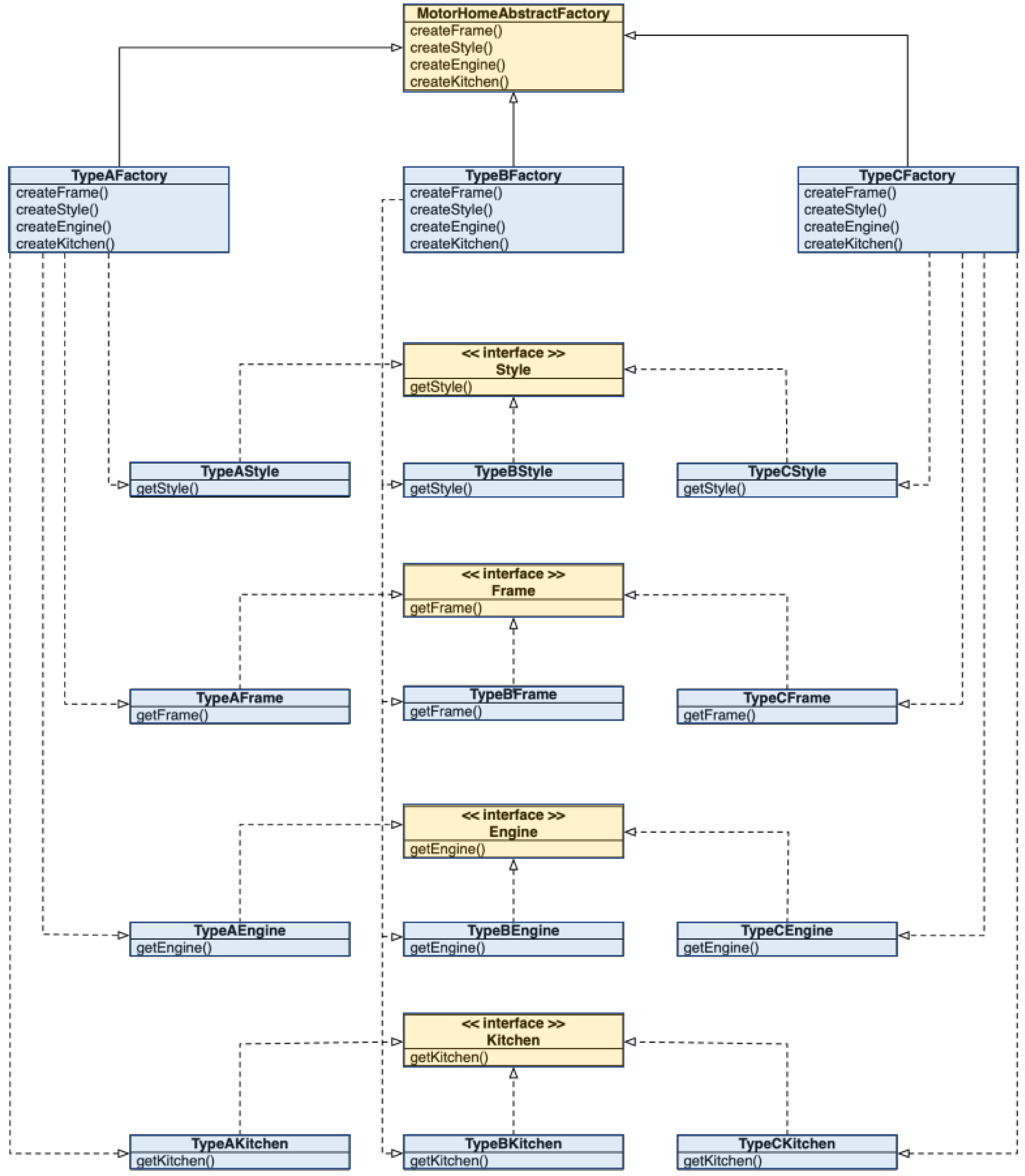
Abstract Factory Implementation

- **Common Implementation Steps:**
  - Create a matrix of product types and platforms (variants).
  - Create abstract product interfaces and concrete product classes.
  - Create the abstract factory interface with creation methods for abstract products.
  - Implement concrete factory classes for each platform.
  - Create factory initialization code.
  - Replace direct calls to product constructors with calls to creation methods.

- **Class Compositions:**



Abstract Factory Use Case Example: Motor-home Manufacturing



Interfaces	General Classes	TypeA Classes	TypeB Classes	TypeC Classes
Type	MotorHomeAbstractFactory	TypeAFactory	TypeBFactory	TypeCFactory
Style	MotorHomeDriver	TypeAStyle	TypeBStyle	TypeCStyle
Frame		TypeAFrame	TypeBFrame	TypleCFrame
Engine		TypeAEngine	TypeBEngine	TypleCEngine
Kitchen		TypeAKitchen	TypeBKitchen	TypeCKitchen

Find the source codes in the course GitHub repository.

Factory Method Pattern Pros & Cons

Pros	Cons
✓ Making sure to have compatible products.	✗ The code may become more complicated because of introducing many interfaces, classes, and subclasses.
✓ Avoiding tight coupling.	
✓ Single Responsibility Principle.	
✓ Open/Closed Principle.	

Any Question

????????????????

How do you feel about the course?

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Please Send Your Question or Feedback...

Top

New

Powered by Poll Everywhere