# Design Pattern

BCS1430

Dr. Ashish Sai

📅 **Week 4 Lecture 2**

💻 **BCS1430.ashish.nl**

📍 **EPD150 MSM Conference Hall**


DESIGN PATTERNS
LIGHTYEAR
DESIGN PATTERNS EVERYWHERE

# Design Patterns

# Design Patterns

Design patterns are typical solutions to recurring design problems in software engineering.

- Essence of Design Patterns
  - **Blueprints** : Pre-made blueprints for solving recurring design issues.
  - **Customizable** : Adaptable to solve specific problems in your code.

# Design Patterns

- Patterns 🎨: Not Just Code
  - **Conceptual** : More concept than the actual code.
  - **Problem-Solving Tools** : Time-tested solutions to frequent software design issues.

# Differentiating Patterns from Algorithms

- Algorithms 🧮: Step-by-step procedures for solving problems, like cooking recipes.
- Design Patterns 🎨: Abstract, flexible schemes for software structure, similar to architectural blueprints.
  - The same pattern can result in different code across different applications.

# The Purpose Behind Design Patterns

- **Creational Patterns:** Simplify object creation, making a system independent of how its objects are created, composed, and represented.

# The Purpose Behind Design Patterns

- **Structural Patterns:** Help to form larger structures while keeping the system flexible and efficient. They ensure that changing one part of the system does not affect other parts.

# The Purpose Behind Design Patterns

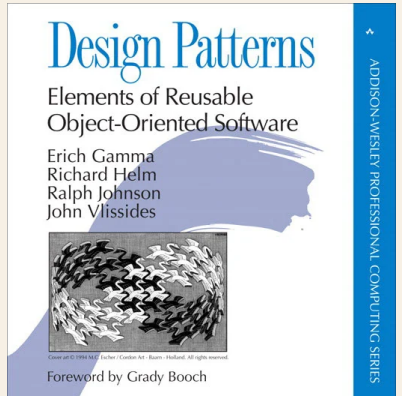
- **Behavioral Patterns:** Enhance communication between objects and help in the assignment of responsibilities between objects, making the interaction more flexible and efficient.

# The Origins and Evolution of Design Patterns

- Initially described by Christopher Alexander in the context of town and building planning  .

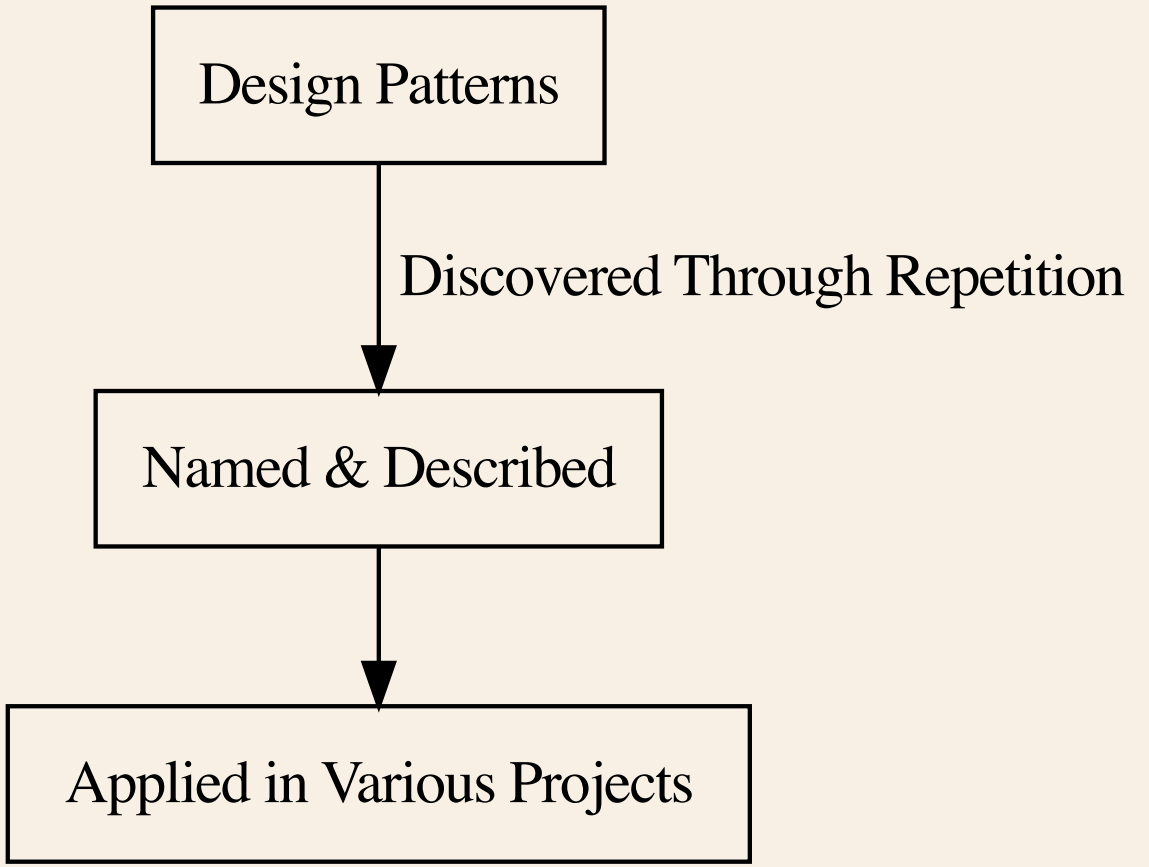
# The Origins and Evolution of Design Patterns

- **Adaptation to Software :**
- **Gang of Four (GoF) :**
  - Design Patterns: Elements of Reusable Object-Oriented Software.
  - Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm
  - /> 500,000 copies!



# Discovery and Documentation of Patterns

Patterns emerge when a solution is repeated across various projects, eventually gaining a name and detailed description.



# The Significance of Learning Design Patterns

- **Problem-Solving Toolkit** : Offers proven solutions for common problems, improving problem-solving skills.
- **Design Vocabulary** : Creates a common language for developers, enhancing communication and collaboration.
- **Best Practices** : Promotes best practices in software design for maintainable and scalable code.

# Criticism of Design Patterns

> If all you have is a hammer 🔨 , everything looks like a nail.

Understanding the limitations and appropriate use of design patterns is crucial.

# Criticism of Design Patterns

- **Language Limitations** : Patterns may arise from a language's shortcomings.
- **Efficiency Concerns** : Misapplying patterns can lead to complex, inefficient solutions. Context matters.
- **Overuse by Novices** : Inexperienced use can lead to overcomplicated code when simpler options exist.

# Classifying Design Patterns

- **By Complexity and Detail:**
  - **Idioms:** Specific to a programming language, addressing low-level issues and specifics.
  - **Architectural Patterns:** High-level patterns that guide the overall structure and organization of software systems.

# Classifying Design Patterns

- **By Purpose:**
  - **Creational Patterns:** Concerned with object creation mechanisms.
  - **Structural Patterns:** Deal with object composition and the formation of larger structures.
  - **Behavioral Patterns:** Focus on communication between objects and responsibilities.

# Creational Design Patterns

# Creational Design Patterns

Creational patterns are fundamental for object creation in software design. They provide mechanisms that increase flexibility and reuse of existing code.

# Five Main Creational Patterns

| Pattern | Description | Covered |
| --- | --- | --- |
| Factory Method | Delegates the creation of objects to subclasses, promoting flexibility and integration. | ✅ |
| Abstract Factory | Creates families of related or dependent objects without specifying their concrete classes. | ✅ |
| Builder | Constructs complex objects step by step, allowing the creation process to create different types and representations of an object. | ❌ |
| Prototype | Creates new objects by copying an existing object, known as the prototype. | ❌ |
| Singleton | Ensures a class has only one instance while providing a global point of access to it. | ✅ |

# Factory Method Pattern

# Introduction to Factory Pattern

- Essential design pattern in object-oriented programming
- Focuses on object creation mechanisms
- Aims to create objects in a manner suitable to the situation

## Types of Factory Pattern

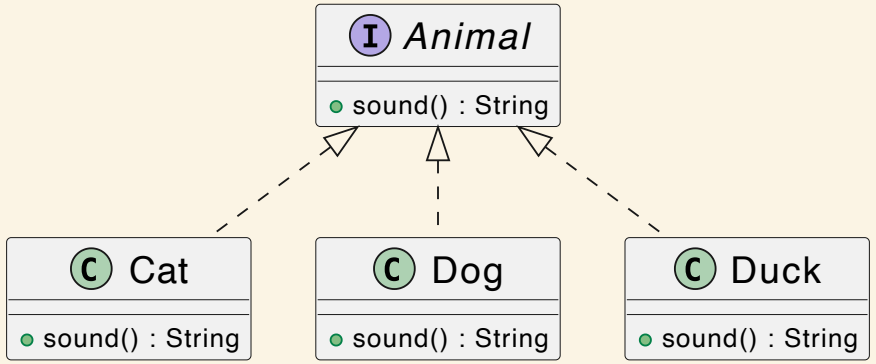
1. Simple Factory
2. Factory Method
3. Abstract Factory

Note: Simple Factory is not a true design pattern

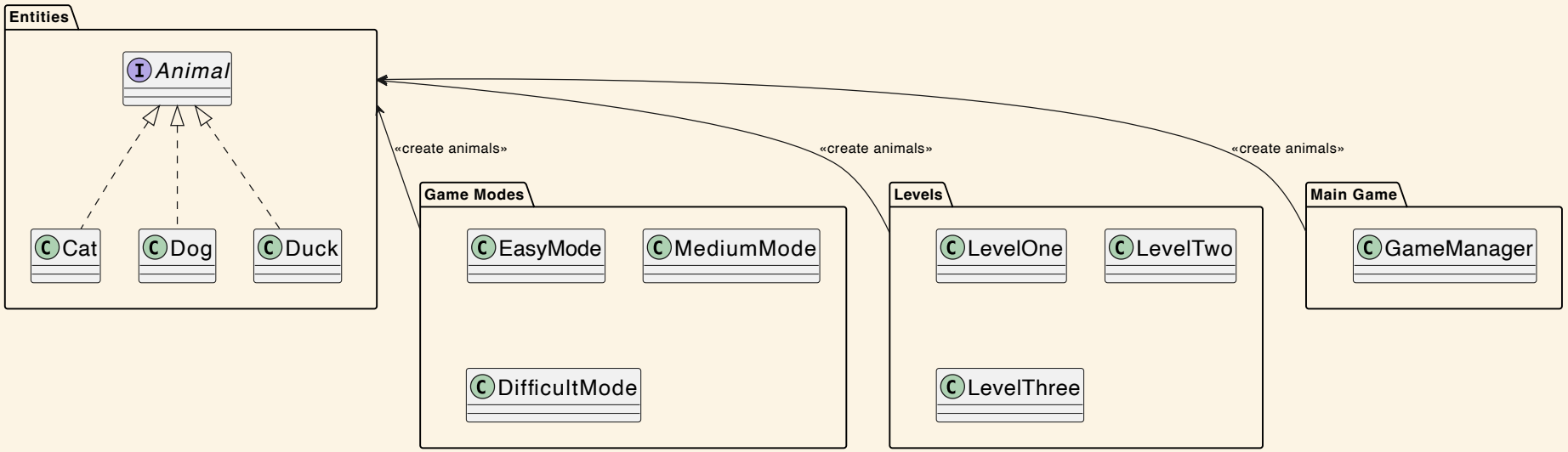
# Creating Animals

- Create an interface animal with some subclasses?
- Subclasses:
  - Cat
  - Dog
  - Duck

# Creating Animals

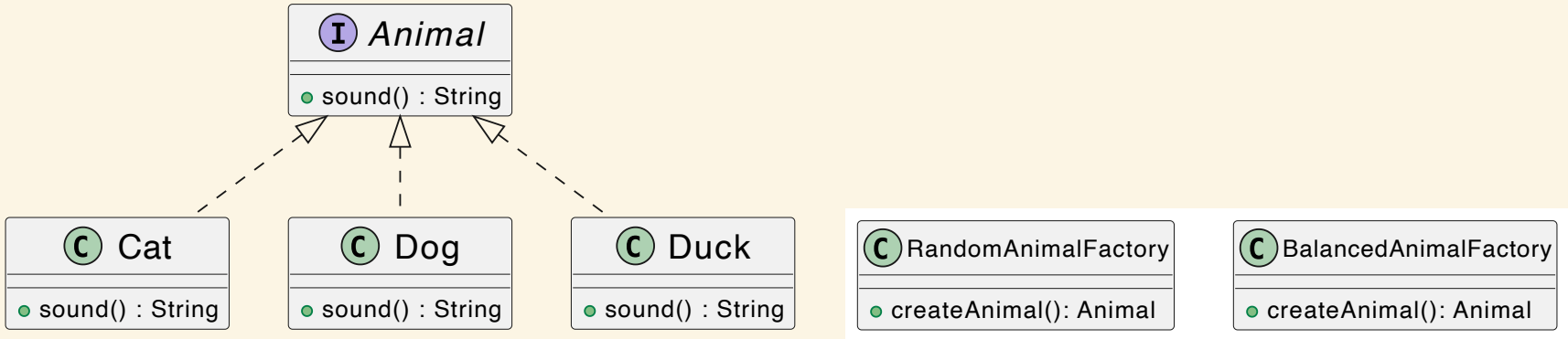


# Creating Animals in the Game

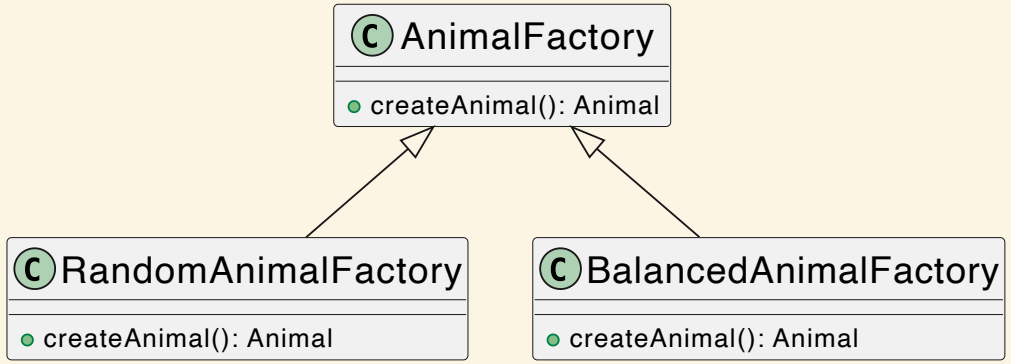


# Creating Animals

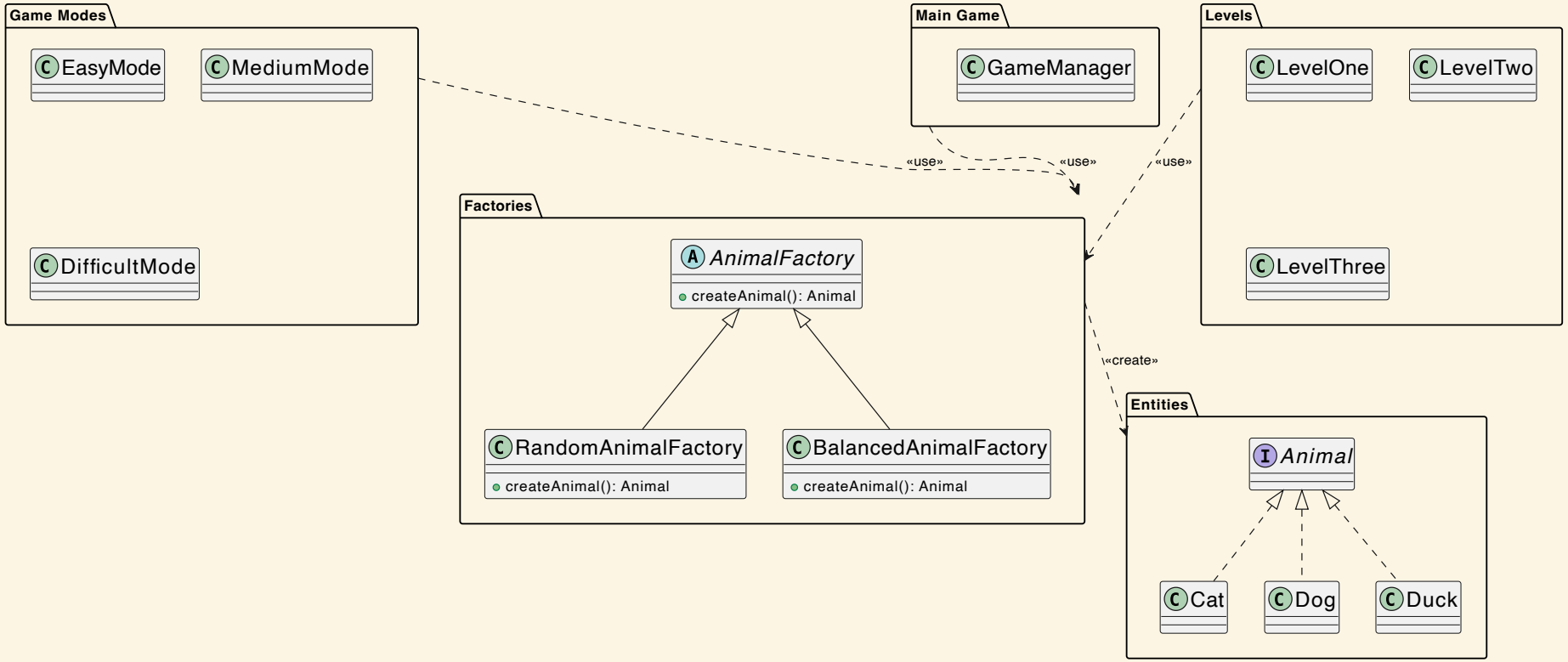
- We have two types of animal creation mechanisms:
  - Random (create Animals at random)
  - Balanced (create equal number of Animals of all types)



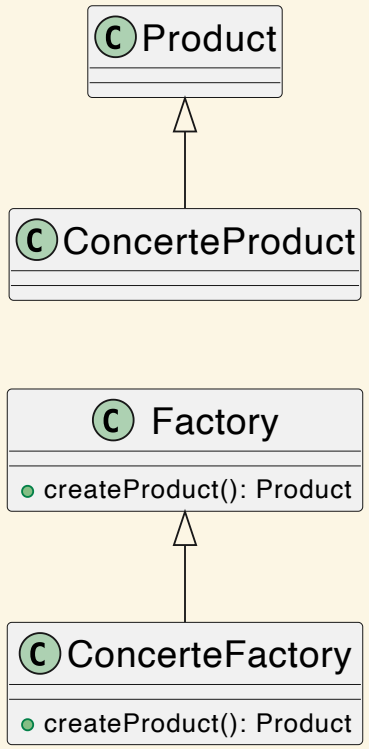
# A Good Animal Factory



# Code with Good Animal Factory

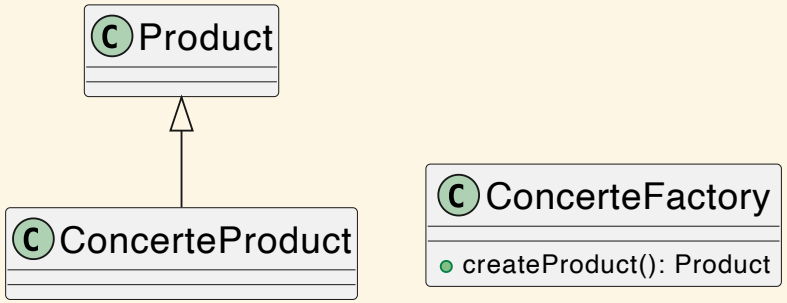


# Stop talking about animal factories




- `Product` is `Animal`
  - `ConcerteProduct` is `Cat`, `Dog` and `Duck`
- `Factory` is `AnimalFactory`
  - `ConcerteFactory` is `BalancedAnimalFactory` and `RandomAnimalFactory`

# Simple Factory



# Introduction to Factory Method

Factory Method is a creational design pattern that provides an interface/contract for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. It’s also known as Virtual Constructor.

## Factory Method Pattern

- Defines an interface/contract for creating objects
- Delegates instantiation to subclasses

# Factory Method: Intent

The Factory Method pattern is designed to:

- Provide an interface for object creation in a superclass.
- Allow subclasses to change the type of objects created.
- Promote loose coupling and scalability in the codebase.

# Factory Method: Problem Scenario

Imagine a logistics management application:

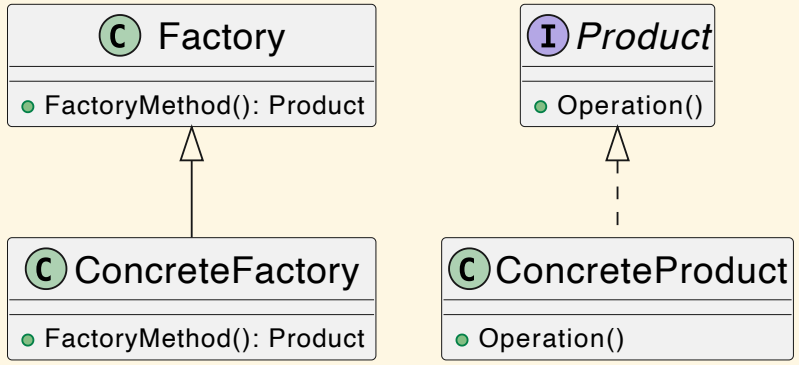
- Initially handles transportation by trucks (`Truck` class).
- Needs to incorporate sea transportation (`Ship` class) due to popular demand.
- Directly adding new classes leads to code tightly coupled with the `Truck` class, making the system inflexible and hard to maintain.

## Factory Method: Solution Overview

The Factory Method pattern suggests:

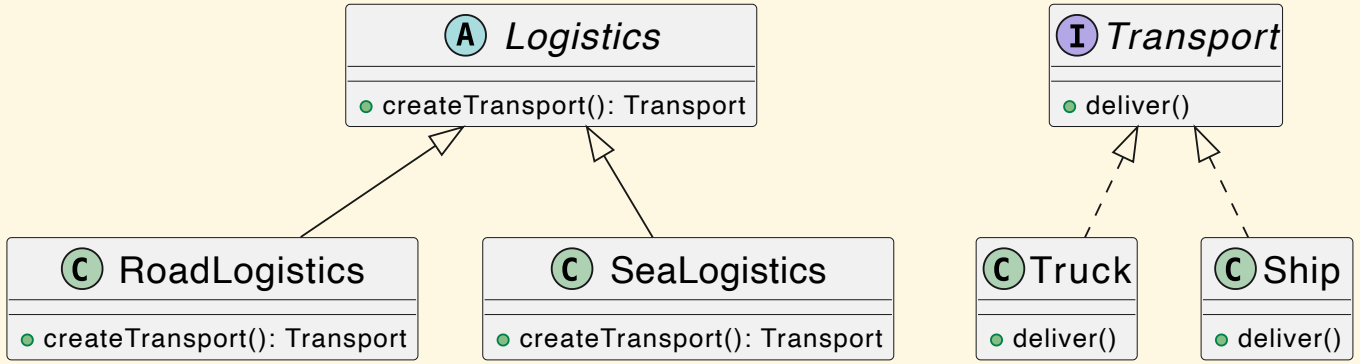
- Replacing direct object construction calls (using `new`) with calls to a special factory method.

# Factory Method: Implementation - Class Structure




This diagram represents the basic structure of the Factory Method pattern.

# Factory Method: Concrete Implementation



# Factory Method: Code Example - Base Classes

```java
public interface Transport {
    void deliver();
}

public abstract class Logistics {
    public abstract Transport createTransport();
    public void planDelivery() {
        Transport transport = createTransport();
        transport.deliver();
    }
}
```

# Factory Method: Code Example - Concrete Classes

```java
public class Truck implements Transport {
    public void deliver() {
        System.out.println("Delivery by land in a box.");
    }
}

public class Ship implements Transport {
    public void deliver() {
        System.out.println("Delivery by sea in a container.");
    }
}

public class RoadLogistics extends Logistics {
    public Transport createTransport() {
        return new Truck();
    }
}

public class SeaLogistics extends Logistics {
    public Transport createTransport() {
```

# Factory Method: Applicability

Use the Factory Method when:

- The exact types and dependencies of objects are not known beforehand.
- You want to provide users with a way to extend internal components of a library or framework.
- You want to save system resources by reusing existing objects instead of rebuilding them.

# Factory Method: Pros and Cons

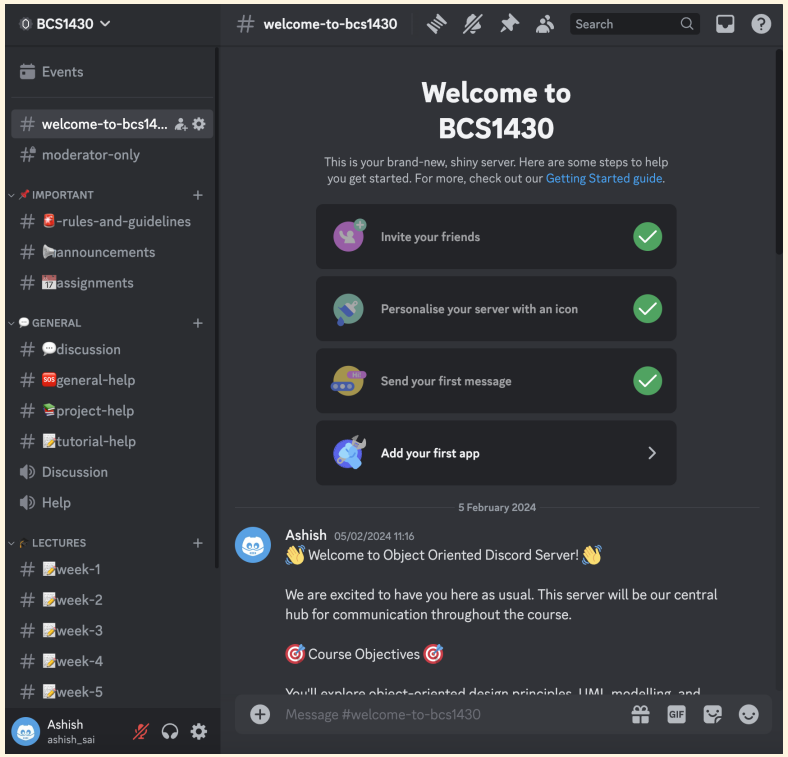
Pros:

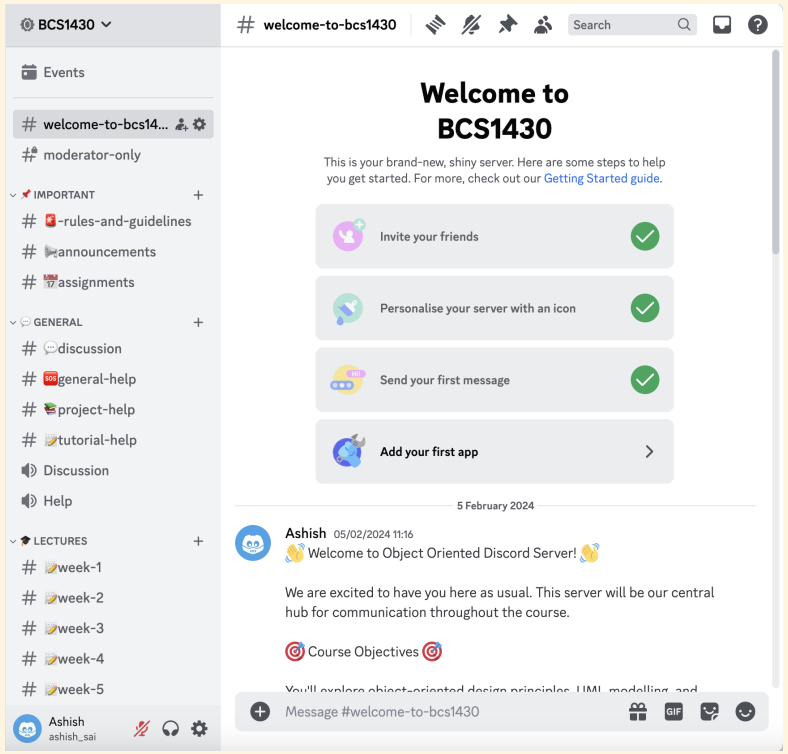
- Avoids tight coupling between creator and concrete products.
- Adheres to the Single Responsibility and Open/Closed Principles.
- Simplifies code maintenance and extension.

Cons:

- Can lead to an increase in the number of classes due to the need for numerous subclasses.

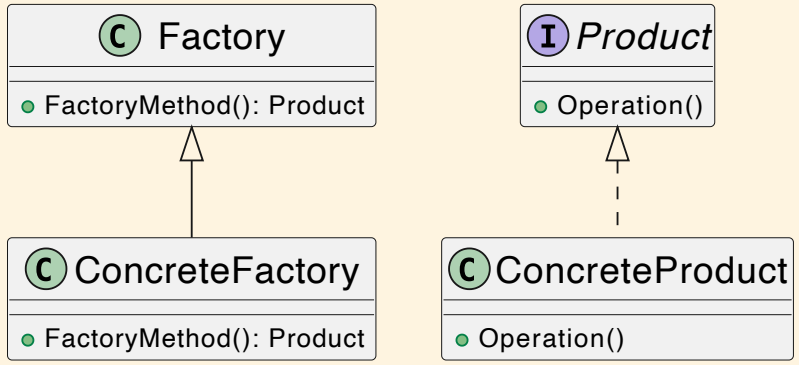
# Abstract Factory Pattern


BCS1430
welcome-to-bcs1430
Search
Events
welcome-to-bcs14...
moderator-only
IMPORTANT
-rules-and-guidelines
announcements
assignments
GENERAL
discussion
general-help
project-help
tutorial-help
Discussion
Help
LECTURES
week-1
week-2
week-3
week-4
week-5
Ashish
ashish_sai
Welcome to BCS1430
This is your brand-new, shiny server. Here are some steps to help you get started. For more, check out our Getting Started guide.
Invite your friends
Personalise your server with an icon
Send your first message
Add your first app
5 February 2024
Ashish 05/02/2024 11:16
Welcome to Object Oriented Discord Server!
We are excited to have you here as usual. This server will be our central hub for communication throughout the course.
Course Objectives
Message #welcome-to-bcs1430

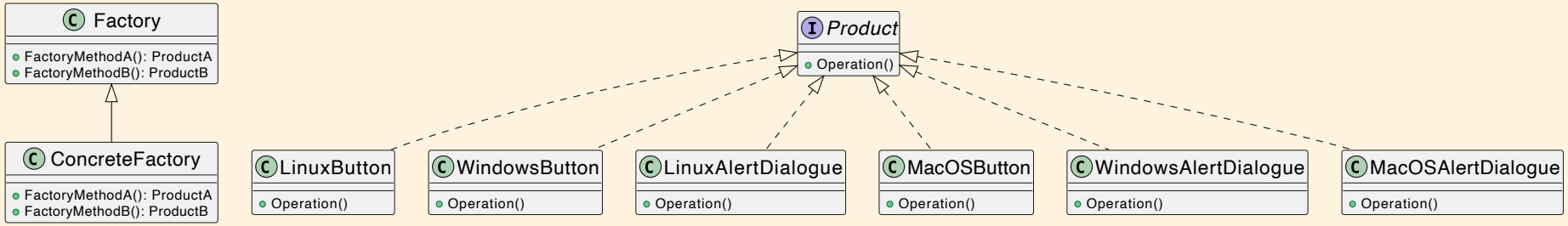

BCS1430
welcome-to-bcs1430
Search
Events
welcome-to-bcs14...
moderator-only
IMPORTANT
-rules-and-guidelines
announcements
assignments
GENERAL
discussion
general-help
project-help
tutorial-help
Discussion
Help
LECTURES
week-1
week-2
week-3
week-4
week-5
Ashish
ashish_sai
Welcome to BCS1430
This is your brand-new, shiny server. Here are some steps to help you get started. For more, check out our Getting Started guide.
Invite your friends
Personalise your server with an icon
Send your first message
Add your first app
5 February 2024
Ashish 05/02/2024 11:16
Welcome to Object Oriented Discord Server!
We are excited to have you here as usual. This server will be our central hub for communication throughout the course.
Course Objectives
Message #welcome-to-bcs1430

# Groups of Objects

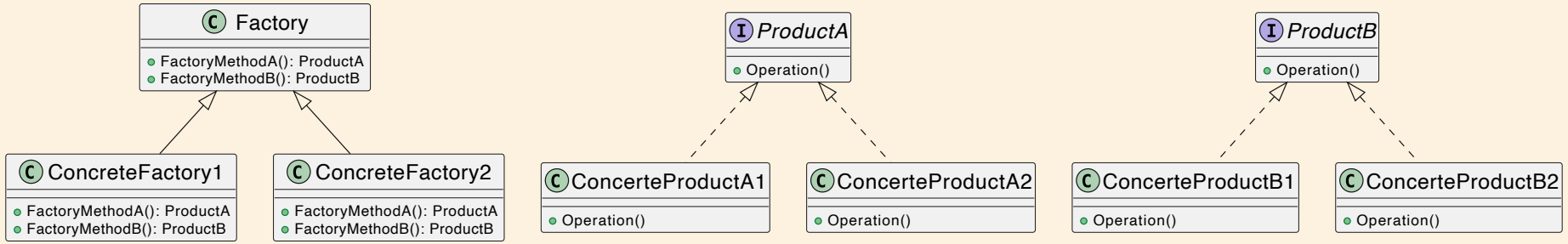
- You may need to create products that are related or dependent.
  - Ensuring that MacOS Alert Dialogues use MacOS Buttons, not Windows Buttons.



# Groups of Objects



# More generally



# Abstract Factory Pattern Overview

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

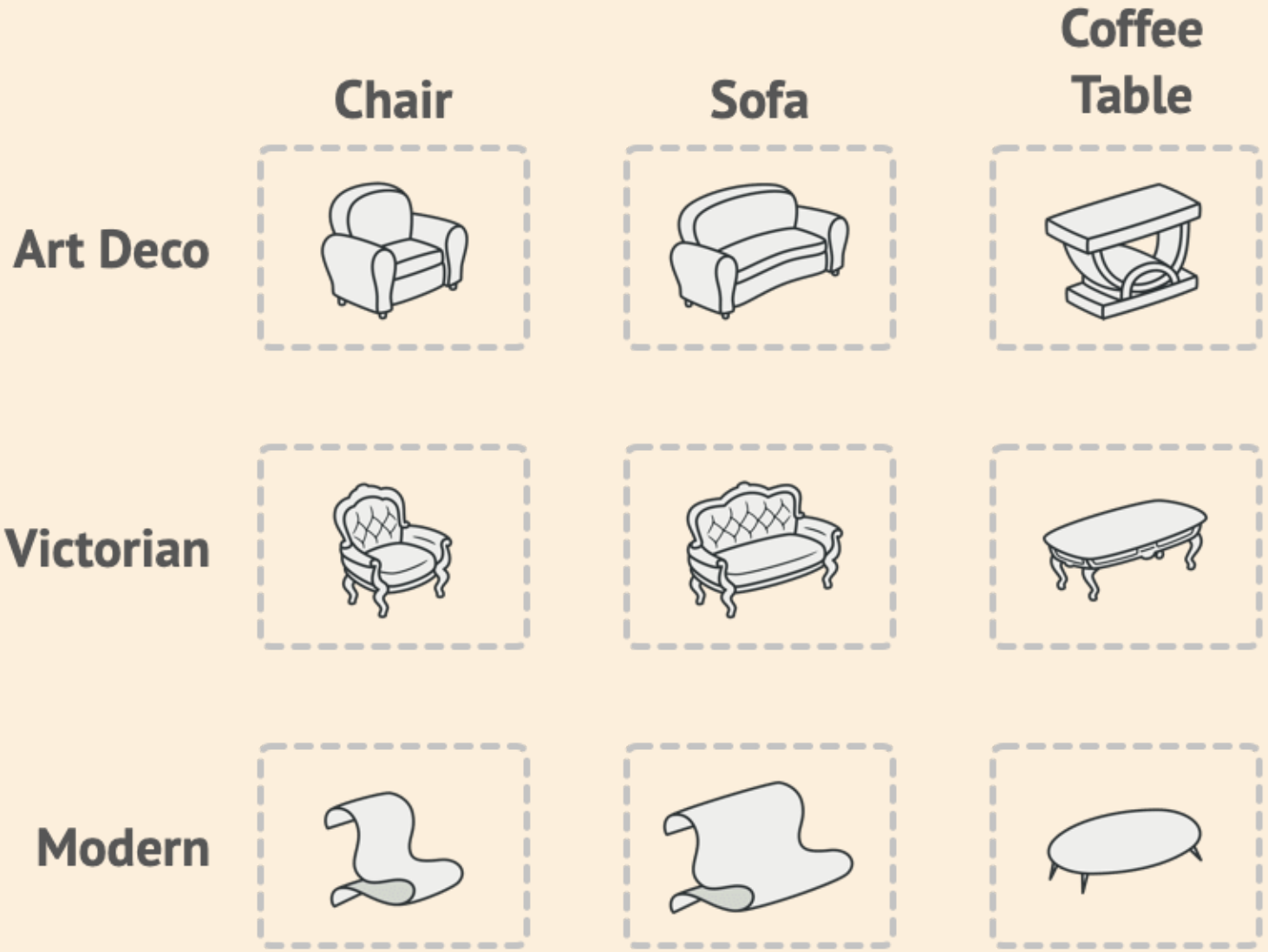
## When to Use

- When the system needs to be independent of how its objects are created
- When the family of related objects is designed to be used together

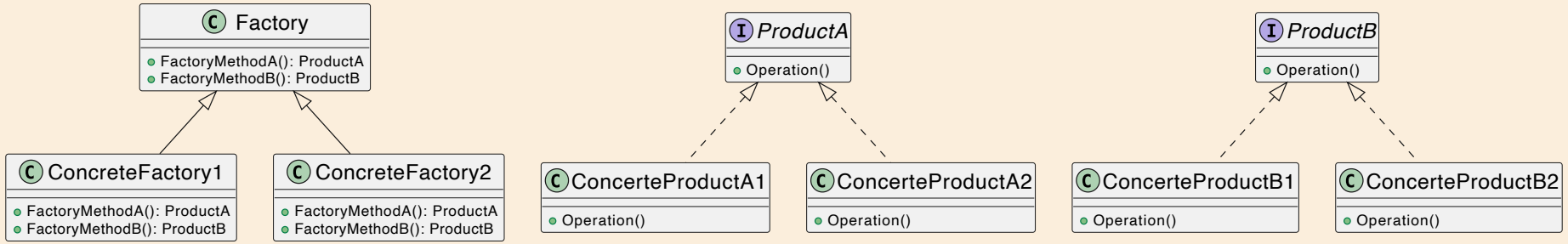
# Abstract Factory: The Problem

- Imagine creating a furniture shop simulator with families of products like `Chair`, `Sofa`, and `CoffeeTable`, available in `Modern`, `Victorian`, and `ArtDeco` styles.
- The challenge is ensuring that furniture pieces match in style without changing the code for new product additions.

# Abstract Factory: The Problem

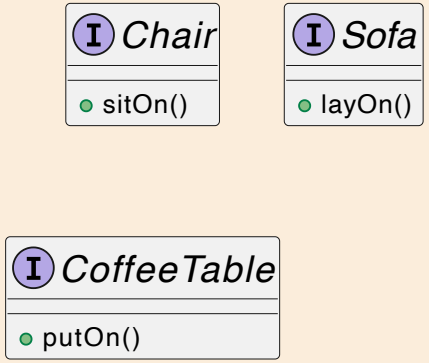


# Solution


Factory
FactoryMethodA(): ProductA
FactoryMethodB(): ProductB
ConcreteFactory1
FactoryMethodA(): ProductA
FactoryMethodB(): ProductB
ConcreteFactory2
FactoryMethodA(): ProductA
FactoryMethodB(): ProductB
ProductA
Operation()
ConcerteProductA1
Operation()
ConcerteProductA2
Operation()
ProductB
Operation()
ConcerteProductB1
Operation()
ConcerteProductB2
Operation()

# Abstract Factory: Abstract Products

Abstract Products declare interfaces for a set of distinct but related products forming a product family.



# Abstract Factory: Concrete Products

Concrete Products are implementations of abstract products, grouped by variants.

```java
// Java Example
interface Chair {
    void sitOn();
}

class ModernChair implements Chair {
    public void sitOn() {
        // Modern chair-specific code
    }
}

class VictorianChair implements Chair {
    public void sitOn() {
        // Victorian chair-specific code
    }
}
```

# Abstract Factory: Abstract Factory Interface

The Abstract Factory interface declares a set of creation methods for all abstract products.

```java
// Java Example
interface FurnitureFactory {
    Chair createChair();
    Sofa createSofa();
    CoffeeTable createCoffeeTable();
}
```

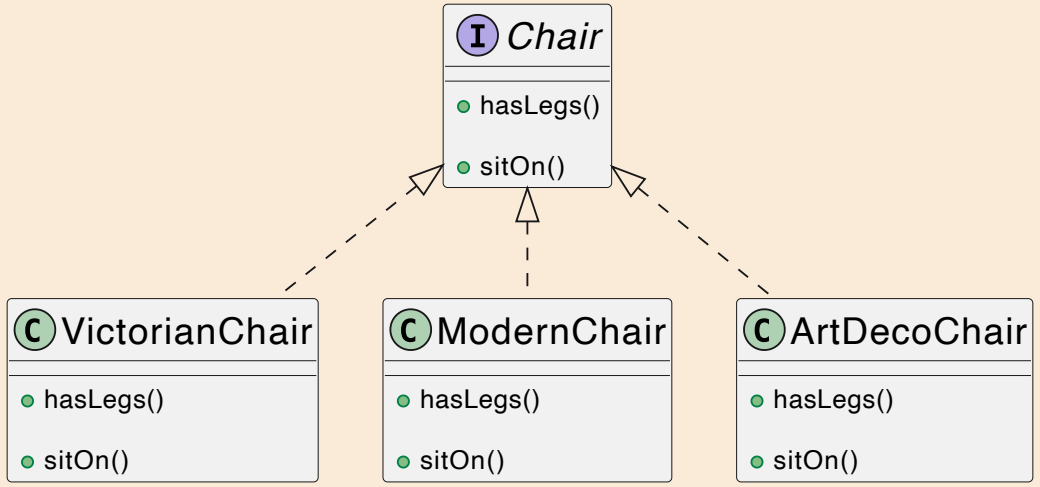
# Abstract Factory: Concrete Factories

Each Concrete Factory corresponds to a specific variant of products and creates only those product variants.

```java
// Java Example
class ModernFurnitureFactory implements FurnitureFactory {
    public Chair createChair() {
        return new ModernChair();
    }
    public Sofa createSofa() {
        return new ModernSofa();
    }
    public CoffeeTable createCoffeeTable() {
        return new ModernCoffeeTable();
    }
}
```

# Abstract Factory: The Solution

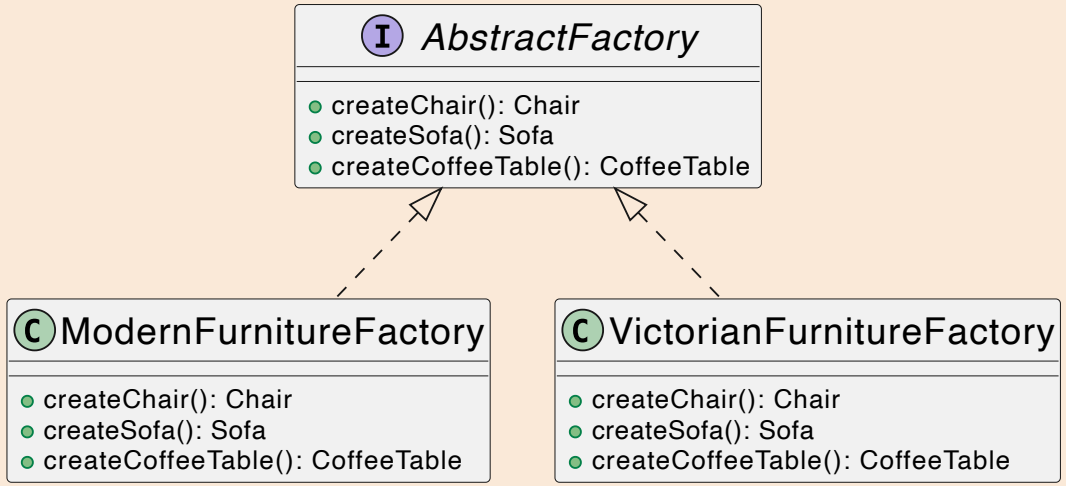
- Step 1. Interfaces for each product (e.g., `Chair`, `Sofa`) and making all variants follow these interfaces.



# Abstract Factory: The Solution

- Step 2. Factories for each variant then produce these objects without exposing the concrete classes.

::: {style="text-align: center;"}



# Abstract Factory: Client Code Interaction

Clients interact with factories and products through abstract types, allowing the flexibility to change factory and product types dynamically.

```java
// Java Example
class Application {
    private FurnitureFactory factory;
    private Chair chair;

    public Application(FurnitureFactory factory) {
        this.factory = factory;
    }

    void createUI() {
        this.chair = factory.createChair();
    }

    void paint() {
        chair.sitOn();
    }
}
```

# Abstract Factory: Pros and Cons

- **Pros:**
  - Ensures products are compatible with each other.
  - Decouples client code from concrete product classes.
  - Adheres to Single Responsibility and Open/Closed Principles.
- **Cons:**
  - Can become complex due to many new interfaces and classes.

# Simple Factory Pattern

## Creation Method

A creation method within a class is responsible for creating instances. This term generally refers to any method whose primary purpose is to create and return a new object.

# Creation Method

```
 1  public class Number {
 2      private int value;
 3
 4      public Number(int value) {
 5          this.value = value;
 6      }
 7
 8      public Number next() {
 9          return new Number(this.value + 1);
10      }
11  }
```

Here, `next` is a creation method, producing a new `Number` instance.

## Static Creation Method

Static creation methods allow calling a method on the class itself to create an instance, often providing clarity and control over the instantiation process.

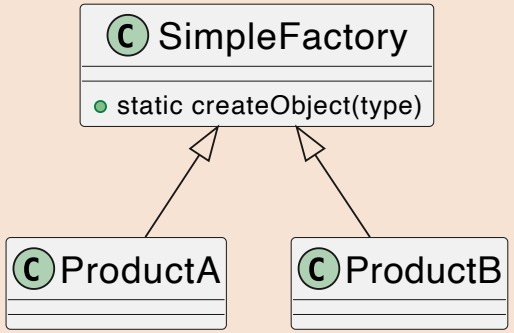
# Static Creation Method

```java
public class User {
    private int id;
    private String name, email, phone;

    private User(int id, String name, String email, String phone) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.phone = phone;
    }

    public static User fromId(int id) {
        // Fetch and construct user from id
        return new User(id, "Name", "Email", "Phone");
    }
}
```

`fromId` is a static creation method, simplifying user creation from an ID.

# Simple Factory Pattern

The Simple Factory encapsulates object creation for a specific type, often based on given parameters.



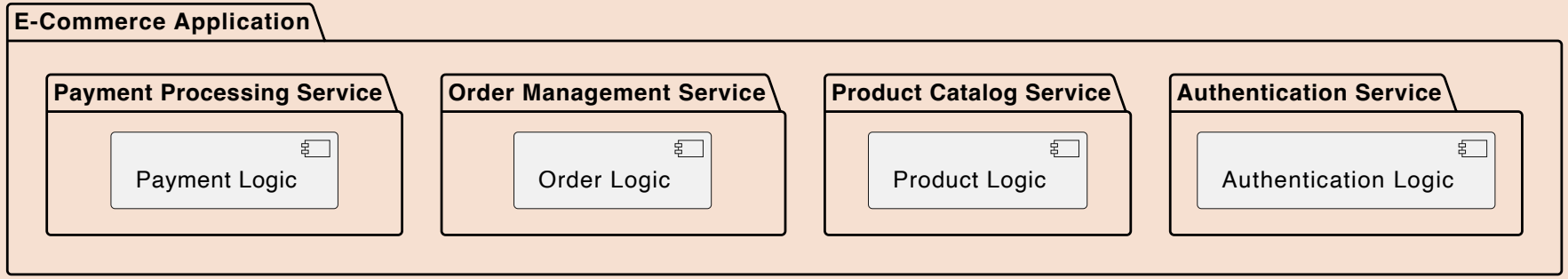
# Simple Factory Pattern

```java
public class SimpleFactory {
    public static Object createObject(String type) {
        if ("A".equals(type)) {
            return new ProductA();
        } else if ("B".equals(type)) {
            return new ProductB();
        }
        throw new IllegalArgumentException("Unknown type");
    }
}
```

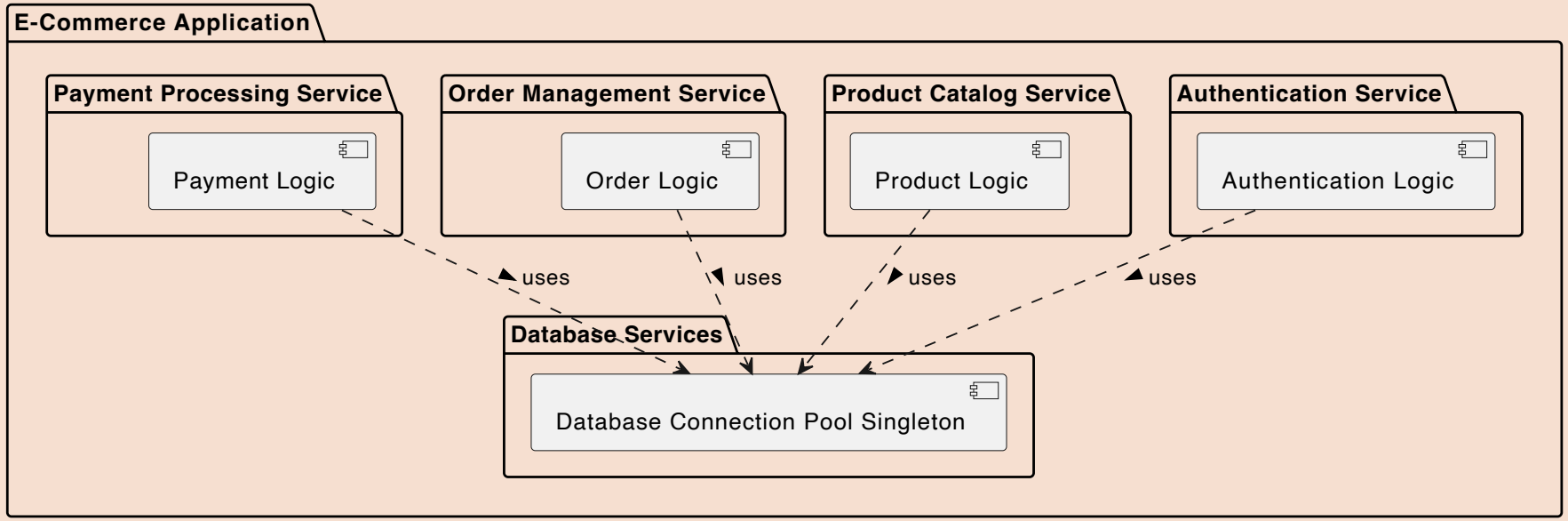
This pattern centralizes and simplifies object creation.

# Singleton Pattern

# You need a Database Object



# How to make it easy?



# What is Singleton Pattern?

The Singleton Pattern is a design pattern that:

- Ensures a class has **only one instance**
- Provides a **global point of access** to that instance

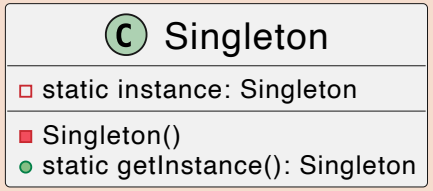
This pattern is useful for coordinating actions across a system.

# Singleton Pattern: The Problem

The Singleton pattern addresses two primary issues: - ensuring a single instance and providing a global access point.

These capabilities are vital for controlling access to shared resources and preventing inconsistencies in the application state.

# Singleton Pattern: Singleton Solution




Singleton implementation involves making the constructor private and providing a static method that returns the same instance.

# Singleton Pattern: Singleton Solution

Java Code:

```java
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

# Singleton Pattern: Implementation Steps

1. **Private Instance:** Add a private static field to store the singleton instance.
2. **Public Creation Method:** Implement a public static method for retrieving the singleton instance.
3. **Lazy Initialization:** Initialize the singleton lazily to ensure it’s created only when needed.

# Singleton Pattern: Lazy Initialization

Implementing lazy initialization ensures the Singleton instance is created only when it's first requested, optimizing resource usage and application performance.

Java Code:

```java
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

# Singleton Pattern: Pros and Cons

**Pros:** - Guaranteed single instance.

- Global access point.
- Lazy initialization.

**Cons:** - Single Responsibility Principle violation.

- Can mask bad design.
- Requires special handling in multithreaded scenarios.

# Singleton Pattern: Usage in Java

Java Code:

```java
public class Database {
    private static Database instance;
    private Database() {}
    public static synchronized Database getInstance() {
        if (instance == null) {
            instance = new Database();
        }
        return instance;
    }
    public void query(String sql) {
        // Implementation here
    }
}
```

This example shows a Singleton used for database connections, a common use case in many applications.

# See you in Lab!