SE 350 - Object-Oriented Software Development

Software Design Patterns: Abstract Factory

Instructor: Stefan Mitsch

Learning Objectives

- Understand the problem solved by Abstract Factory
- Understand the pattern

GoF Design Pattern Space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	**Abstract Factory**  Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

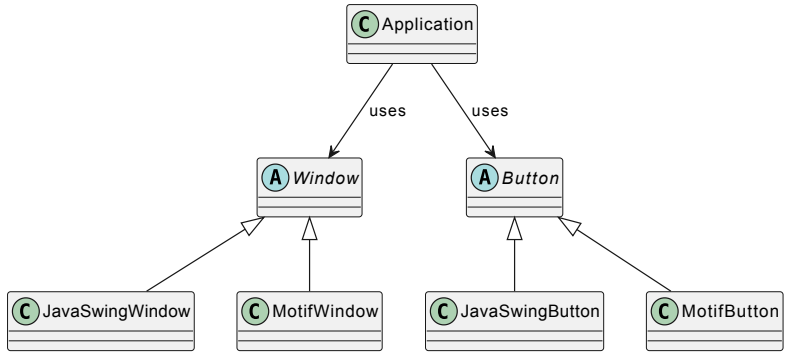
Introduction

Abstract Factory produces families of related object

- Creational design pattern
- For applications that need to **switch out entire families of objects** (rather than a single product)
  - Framework phrased in terms of interfaces and abstract classes
  - Concrete subclasses of the framework decide *what* to instantiate

Problem illustration: `Application`, `Window`, `Button`, and other elements are a framework for implementing user interfaces.

❓ What kind of specific `Window` and `Button` should application use?



Overview

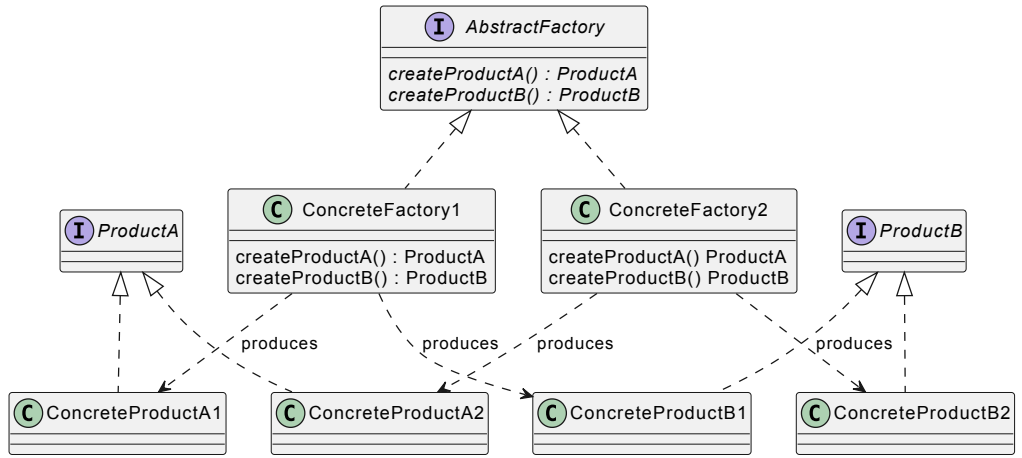
Problem

- Encapsulate platform dependencies
- Have families of domain objects to switch

Intent

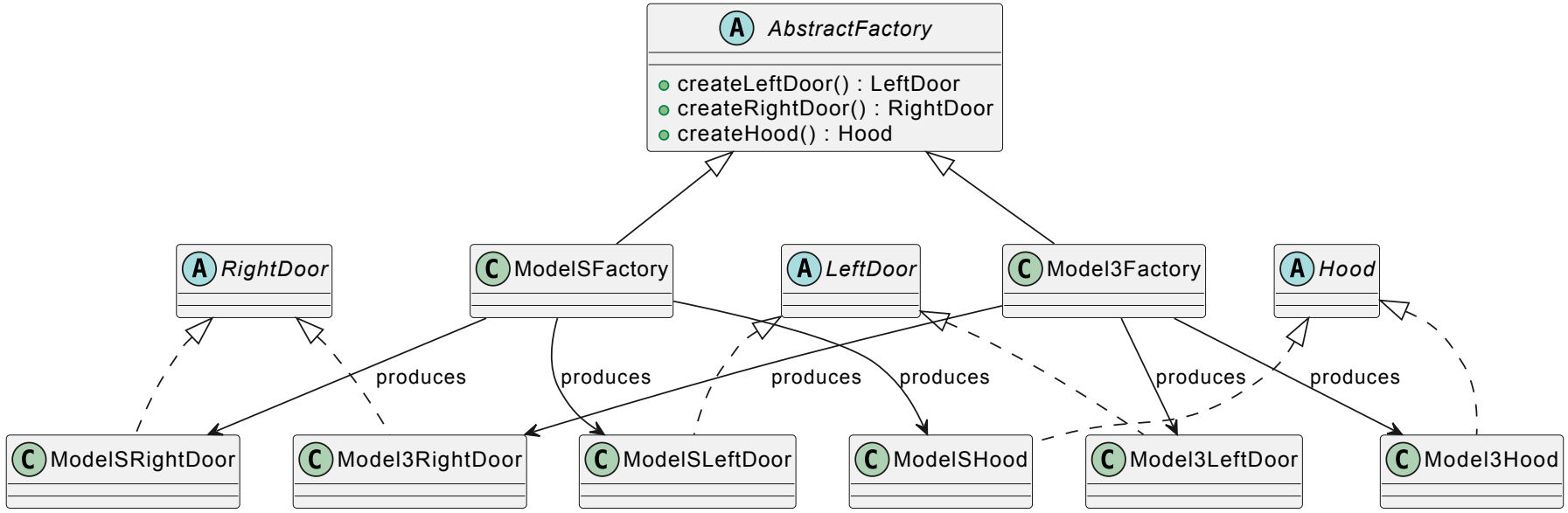
- Interface for creating families of objects
- Hierarchy that encapsulates several different "platforms"
- Avoid hardcoded `new` creation

Structure



Abstract Factory Real-World Example

- Manufacture parts of different car models



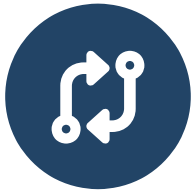
Implementation

- Create a matrix of product types and platforms
- Create abstract product interfaces and concrete product classes
- Create the abstract factory class with `create` methods for abstract products
- Implement concrete factories for each platform
- Replace direct constructor calls with factory calls

```
public abstract class AbstractFactory {
  public abstract Room createRoom();
  public abstract Door createDoor();
}
public abstract class Room {}
public abstract class Door {}

// Concrete creator and product: dungeon game
public class DungeonFactory extends AbstractFactory {
  @Override public Room createRoom() {
    return new Chamber();
  }
  @Override public Door createDoor() {
    return new WoodenDoor();
  }
}
public class Chamber extends Room {}
public class WoodenDoor extends Door {}

// Concrete creator and product: enchanted maze game
public class EnchantedMazeFactory extends AbstractFactory {
  @Override public Room createRoom() {
    return new EnchantedRoom();
  }
  @Override public Door createDoor() {
    return new EnchantedDoor();
  }
}
public class EnchantedRoom extends Room {}
public class EnchantedDoor extends Door {}
```



Abstract Factory Variations

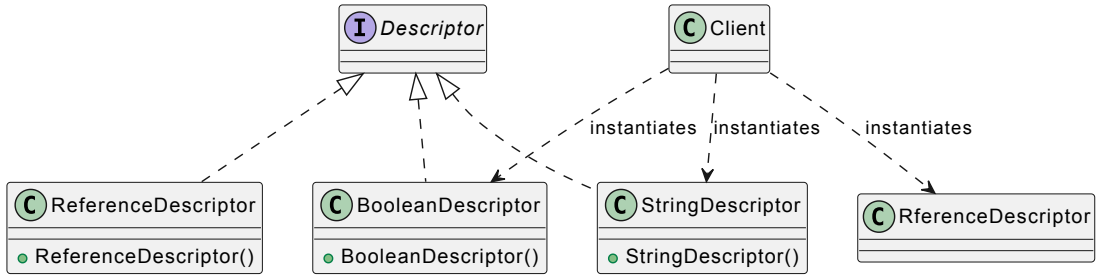
- Factories as singletons: when an application only needs one concrete factory
- Combine with the "Protype" design pattern: when families have large overlap
- Extensible factories
    - In a simple implementation, Abstract Factory violates the Open/Closed Principle (new products require changing the abstract factory and all concrete factories)
    - No easy solution (factory method parameters have the same problem)
- Collection of factory methods

Refactoring to Factory Method Collection

Code Smell: Clients directly instantiate classes that implement a common interface

Before: Clients directly instantiate classes

▶ After Refactoring



Discussion

Benefits

- Ensures compatible products
- Reduces coupling
- Promotes Single Responsibility Principle
- Promotes the Open/Closed Principle

Drawbacks

- May introduce many interfaces for `Product` and subclasses of `AbstractFactory`
- Difficult to make extensible