# Abstract Factory



Resources

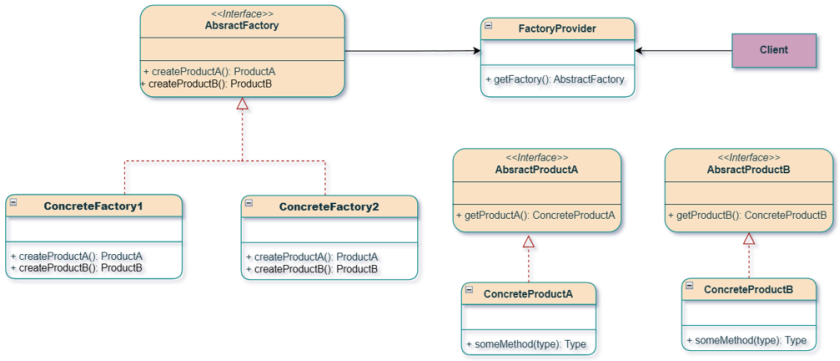
- Video → 27. All Creational Design Patterns | Prototype, Singleton, Factory, AbstractFactory, Builder Pattern
- Video → 5. Factory Pattern Vs Abstract Factory Pattern Explanation (Hindi) | Low Level System Design

## Definition

> *The Abstract Factory Pattern provides an **interface** for **creating families of related objects** without specifying their **concrete classes**. It's like having multiple factories, where each factory produces a complete family of products that **work together/are related/dependent.***

As a result, an Abstract Factory is known as the Factory of Factories or a Super Factory. It is more complex than the Factory pattern, as it is responsible for creating families of multiple related product types. Hence, we see multiple factory methods in the interface. This high level of abstraction enables the client to switch between different product families at runtime without knowing the concrete implementation of product instantiation.

## Class Diagram



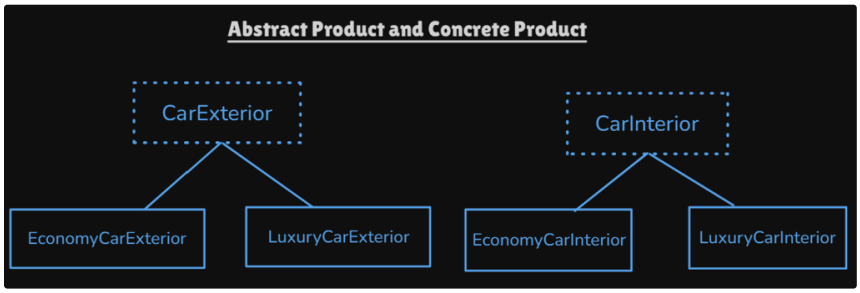

## Example: Car Manufacturing

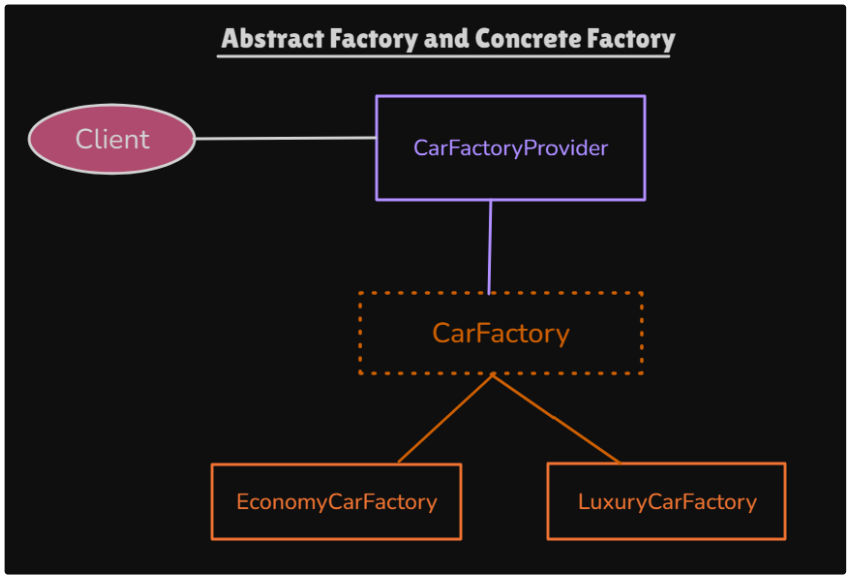
1. **Abstract Product Interfaces(** `CarInterior` , `CarExterior` **)**
   - Define interfaces for a family of related products.
   - Each product type has its interface.
   - Clients depend on this interface to create objects without needing to know their specific types.
2. **Concrete Products (** `EconomyCarInterior` , `LuxuryCarExterior` **, etc.)**

- Implement the abstract product interfaces.
- Products from the same family are designed to work together.

3. **Abstract Factory (`CarFactory`)**
   - Declares creation methods for each product type.
   - Defines the contract for creating product families.
4. **Concrete Factories (`EconomyCarFactory`, `LuxuryCarFactory`)**
   - Implement the abstract factory interface.
   - Each factory produces products from a single specific family.
5. **Factory Provider**
   - Returns a concrete factory based on client input in the request.
   - This concrete factory is used to create families of concrete products as per the client's request.
6. **Client (`Application`)**
   - Utilizes the **Abstract Factory** Interface to request concrete products.

## Implementation




1. **Abstract Product Interfaces**

```java
// Step 1: Abstract Product interfaces - Define product families
public interface CarExterior {
    void addExteriorComponents();
}
public interface CarInterior {
    void addInteriorComponents();
```

```
}
```

2. **Concrete Products**

```
// Step 2: Concrete Products for Economy Car Family
public class EconomyCarExterior implements CarExterior {
    @Override
    public void addExteriorComponents() {
        System.out.println("Adding basic exterior components for
Economy Car.");
    }
}
public class EconomyCarInterior implements CarInterior {
    @Override
    public void addInteriorComponents() {
        System.out.println("Adding basic interior components for
Economy Car.");
    }
}
// Step 3: Concrete Products for Luxury Car Family
public class LuxuryCarExterior implements CarExterior {
    @Override
    public void addExteriorComponents() {
        System.out.println("Adding luxurious exterior components for
Luxury Car.");
    }
}
public class LuxuryCarInterior implements CarInterior {
    @Override
    public void addInteriorComponents() {
        System.out.println("Adding luxurious interior components for
Luxury Car.");
    }
}
```

3. **Abstract Factory**

```
// Step 4: Abstract Factory interface
public interface CarFactory {

    // Factory methods
    CarInterior createInterior();

    CarExterior createExterior();

    // Template method that uses all factory methods
    default void produceCompleteVehicle() {

        System.out.println("Starting complete vehicle production...");

        // Create all components
        CarInterior interior = createInterior();
        CarExterior exterior = createExterior();

        // Assemble the vehicle
        interior.addInteriorComponents();
        exterior.addExteriorComponents();

        System.out.println("Vehicle production completed!");
    }
}
```

4. **Concrete Factories**

```
// Step 5: Concrete Factories
public class EconomyCarFactory implements CarFactory {
    private String brand;

    public EconomyCarFactory(String brand) {
        this.brand = brand;
```

```
    }
    @Override
    public CarInterior createInterior() {
        return new EconomyCarInterior();
    }

    @Override
    public CarExterior createExterior() {
        return new EconomyCarExterior();
    }
}
// Step 5: Concrete Factories
public class LuxuryCarFactory implements CarFactory {
    private final String brand;

    public LuxuryCarFactory(String brand) {
        this.brand = brand;
    }

    @Override
    public CarInterior createInterior() {
        return new LuxuryCarInterior();
    }

    @Override
    public CarExterior createExterior() {
        return new LuxuryCarExterior();
    }
}
```

5. **Factory Provider**

```
// Step 6: Factory Provider
public class CarFactoryProvider {
    public CarFactory getFactory(CarType type, String brand) {
        switch (type) {
            case ECONOMY:
                return new EconomyCarFactory(brand);
            case PREMIUM:
            case LUXURY:
                return new LuxuryCarFactory(brand);
            default:
                throw new IllegalArgumentException("Unknown car type: " + type);
        }
    }
}
```

6. Client

```
// Step 7: Client Application
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        System.out.println("=====Abstract Factory Design Pattern=====");
        // Get Factory Provider
        CarFactoryProvider carFactoryProvider = new CarFactoryProvider();

        // Get Economy Car Factory
        CarFactory economyCar = carFactoryProvider.getFactory(CarType.ECONOMY, "Honda");
        economyCar.produceCompleteVehicle();

        // Get Luxury Car Factory
        CarFactory luxuryCar = carFactoryProvider.getFactory(CarType.LUXURY, "Mercedes");
        luxuryCar.produceCompleteVehicle();

        // Get Premium Car Factory
```

```
        CarFactory premiumCar =
carFactoryProvider.getFactory(CarType.PREMIUM, "Rolls Royce");
        premiumCar.produceCompleteVehicle();
    }
}
```

**Output**

```
=====Abstract Factory Design Pattern=====
Starting complete vehicle production...
Adding basic interior components for Economy Car.
Adding basic exterior components for Economy Car.
Vehicle production completed!
Starting complete vehicle production...
Adding luxurious interior components for Luxury Car.
Adding luxurious exterior components for Luxury Car.
Vehicle production completed!
Starting complete vehicle production...
Adding luxurious interior components for Luxury Car.
Adding luxurious exterior components for Luxury Car.
Vehicle production completed!

Process finished with exit code 0
```

The Abstract Factory Pattern is highly effective when you need to ensure that related products work well together, making it especially suitable for complex systems with various product variants. This highlights the pattern's strong extensibility and its compliance with the Open/Closed Principle.

## Factory Method vs Abstract Factory

Refer → 5. Factory Pattern Vs Abstract Factory Pattern Explanation (Hindi) | Low Level System Design

**Summary:**

- One product, many variants → Use **Factory Method**
- Many products, grouped by family (theme, platform, brand, etc.) → Use **Abstract Factory**