



# Abstract Factory Pattern (GoF Creational)

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”

---



## Real-Life Analogy (Advanced Logistics System)

Let's say you now have a logistics system that doesn't just create a `Transport`, but also creates a `Packaging` service depending on **transport type**.

- Road Logistics:
  - `Transport` → `Truck`
  - `Packaging` → `CardboardBox`
- Sea Logistics:
  - `Transport` → `Ship`
  - `Packaging` → `ContainerBox`

You now need a system that **creates both transport and packaging together**, depending on the selected logistics mode (road or sea).

---



## Abstract Factory Pattern Implementation

---

### ➤ Step 1: Product Families

```
// Product A: Transport
interface Transport {
    void deliver();
}

// Product B: Packaging
interface Packaging {
    void pack();
}
```

---

### ➤ Step 2: Concrete Products

```
// Road Products
class Truck implements Transport {
    public void deliver() {
        System.out.println("Delivering by truck.");
    }
}
```

```

class CardboardBox implements Packaging {
    public void pack() {
        System.out.println("Packing with cardboard box.");
    }
}

// Sea Products
class Ship implements Transport {
    public void deliver() {
        System.out.println("Delivering by ship.");
    }
}

class ContainerBox implements Packaging {
    public void pack() {
        System.out.println("Packing with container box.");
    }
}

```

---

### ➤ Step 3: Abstract Factory

```

interface LogisticsFactory {
    Transport createTransport();
    Packaging createPackaging();
}

```

---

### ➤ Step 4: Concrete Factories

```

class RoadLogisticsFactory implements LogisticsFactory {
    public Transport createTransport() {
        return new Truck();
    }

    public Packaging createPackaging() {
        return new CardboardBox();
    }
}

class SeaLogisticsFactory implements LogisticsFactory {
    public Transport createTransport() {
        return new Ship();
    }

    public Packaging createPackaging() {
        return new ContainerBox();
    }
}

```

---

### ➤ Step 5: Client Code

```

public class AbstractFactoryDemo {
    private Transport transport;
    private Packaging packaging;

    public AbstractFactoryDemo(LogisticsFactory factory) {
        transport = factory.createTransport();
        packaging = factory.createPackaging();
    }
}

```

```

    }

    public void planShipment() {
        packaging.pack();
        transport.deliver();
    }

    public static void main(String[] args) {
        // Road shipment
        AbstractFactoryDemo roadShipment = new AbstractFactoryDemo(new
RoadLogisticsFactory());
        roadShipment.planShipment();

        // Sea shipment
        AbstractFactoryDemo seaShipment = new AbstractFactoryDemo(new
SeaLogisticsFactory());
        seaShipment.planShipment();
    }
}

```

## What Abstract Factory Solves

- You need **multiple related objects** (Transport + Packaging).
- You want to ensure **they're compatible** (Ship ↔ ContainerBox, Truck ↔ CardboardBox).
- You avoid hardcoding types; clients only work with **factory interface**.

## Factory Method vs Abstract Factory – In-Depth Comparison

Feature	Factory Method	Abstract Factory
<b>Purpose</b>	Create one product type	Create <b>families</b> of related products
<b>How many products created?</b>	One	Multiple (usually two or more)
<b>Pattern Type</b>	Creational	Creational
<b>Extensibility</b>	Easy to add new product via subclass	Easy to add new product families via new factories
<b>Client depends on</b>	Abstract Creator class	Abstract Factory interface
<b>Complexity</b>	Moderate	Higher – more moving parts
<b>Example</b>	Only creates Transport	Creates both Transport and Packaging

## Summary

When to Use...	Factory Method	Abstract Factory
You need <b>one object</b> , and may vary it via subclassing	You need <b>families of related objects</b>	
Example: Make a Vehicle depending	Example: Make both a Vehicle and its	

When to Use...	Factory Method	Abstract Factory
on the route Easy to maintain and test	Packaging Excellent for plug-and-play architectures	

---



## Design Tip

- Use **Factory Method** when you care about varying **one product** via subclassing.
  - Use **Abstract Factory** when you want to switch entire **families of products** with a single configuration change.
-