Factory Design Pattern - Complete Guide

📋 Table of Contents

📖 Pattern Overview

The Factory Design Pattern is a **creational design pattern** that provides an interface for creating objects without specifying their exact classes. It encapsulates object creation logic and promotes loose coupling between client code and concrete classes.

Problem Solved

- Eliminates direct object instantiation using `new` keyword
- Centralizes object creation logic
- Makes code more flexible and maintainable
- Supports Open/Closed Principle

Key Benefits

- **Encapsulation**: Object creation logic is hidden
- **Flexibility**: Easy to add new product types
- **Loose Coupling**: Client doesn't depend on concrete classes
- **Single Responsibility**: Creation logic separated from business logic

---

🏭 Simple Factory Pattern

Based on the first whiteboard diagram, let's understand the burger creation system:

Structure

Client → BurgerFactory → Creates → {BasicBurger, StandardBurger, PremiumBurger}

Components

- **Product Interface**: `Burger` (abstract)
- **Concrete Products**: `BasicBurger`, `StandardBurger`, `PremiumBurger`
- **Factory Class**: `BurgerFactory`
- **Client**: Uses factory to get burgers

Code Implementation

java

```
// Product Interface
abstract class Burger {
    protected String name;
    protected double price;

    public abstract void prepare();
    public abstract void cook();
    public abstract void pack();

    public String getName() { return name; }
    public double getPrice() { return price; }
}

// Concrete Products
class BasicBurger extends Burger {
    public BasicBurger() {
        name = "Basic Burger";
        price = 5.99;
    }

    @Override
    public void prepare() {
        System.out.println("Preparing " + name + " with basic ingredients");
    }

    @Override
    public void cook() {
        System.out.println("Cooking basic burger for 3 minutes");
    }

    @Override
    public void pack() {
        System.out.println("Packing in standard wrapper");
    }
}

class StandardBurger extends Burger {
    public StandardBurger() {
        name = "Standard Burger";
        price = 8.99;
    }

    @Override
    public void prepare() {
        System.out.println("Preparing " + name + " with cheese and lettuce");
    }
```

```
    @Override
    public void cook() {
        System.out.println("Cooking standard burger for 5 minutes");
    }

    @Override
    public void pack() {
        System.out.println("Packing in branded box");
    }
}

class PremiumBurger extends Burger {
    public PremiumBurger() {
        name = "Premium Burger";
        price = 12.99;
    }

    @Override
    public void prepare() {
        System.out.println("Preparing " + name + " with premium ingredients");
    }

    @Override
    public void cook() {
        System.out.println("Cooking premium burger for 7 minutes");
    }

    @Override
    public void pack() {
        System.out.println("Packing in luxury packaging");
    }
}

// Simple Factory
class BurgerFactory {
    public static Burger createBurger(String type) {
        switch (type.toLowerCase()) {
            case "basic":
                return new BasicBurger();
            case "standard":
                return new StandardBurger();
            case "premium":
                return new PremiumBurger();
            default:
                throw new IllegalArgumentException("Unknown burger type: " + type);
        }
```

```
    }
}

// Client Usage
public class SimpleFactoryDemo {
    public static void main(String[] args) {
        // Client doesn't need to know about concrete classes
        Burger burger1 = BurgerFactory.createBurger("basic");
        Burger burger2 = BurgerFactory.createBurger("premium");

        // Process orders
        processBurgerOrder(burger1);
        processBurgerOrder(burger2);
    }

    private static void processBurgerOrder(Burger burger) {
        System.out.println("\n--- Processing Order for " + burger.getName() + " ---");
        burger.prepare();
        burger.cook();
        burger.pack();
        System.out.println("Order complete! Price: $" + burger.getPrice());
    }
}
```

Simple Factory Limitations

- Violates Open/Closed Principle (need to modify factory for new products)
- Factory becomes complex with many product types
- Not a true GoF pattern (more of a programming idiom)

🏭 Factory Method Pattern

Based on the second diagram, this shows the evolution to Factory Method pattern:

Structure

Abstract Factory → Concrete Factories → Create → Specific Products
BurgerFactory → {StyleBurgerFactory, KingBurgerFactory} → Create → Different Burger Types

Components

- **Abstract Creator**: `BurgerFactory`
- **Concrete Creators**: `StyleBurgerFactory`, `KingBurgerFactory`
- **Abstract Product**: `Burger`

- **Concrete Products**: Different burger implementations for each factory

Code Implementation

java

```
// Abstract Creator (Factory Method Pattern)
abstract class BurgerFactory {
    // Factory Method - subclasses will implement this
    public abstract Burger createBurger(String type);

    // Template method that uses factory method
    public Burger orderBurger(String type) {
        Burger burger = createBurger(type);  // Factory method call

        // Common ordering process
        System.out.println("\n--- Processing " + burger.getName() + " Order ---");
        burger.prepare();
        burger.cook();
        burger.pack();
        return burger;
    }
}

// Concrete Creator 1 - Style Burger Factory
class StyleBurgerFactory extends BurgerFactory {
    @Override
    public Burger createBurger(String type) {
        switch (type.toLowerCase()) {
            case "basic":
                return new StyleBasicBurger();
            case "garlic":
                return new StyleGarlicBurger();
            case "wheat":
                return new StyleWheatBurger();
            default:
                throw new IllegalArgumentException("Style Burger type not available: " + type);
        }
    }
}

// Concrete Creator 2 - King Burger Factory
class KingBurgerFactory extends BurgerFactory {
    @Override
    public Burger createBurger(String type) {
        switch (type.toLowerCase()) {
            case "basic":
                return new KingBasicBurger();
            case "garlic":
                return new KingGarlicBurger();
            case "wheat":
                return new KingWheatBurger();
```

```
            default:
                throw new IllegalArgumentException("King Burger type not available: " + type);
        }
    }
}

// Style Burger Products
class StyleBasicBurger extends Burger {
    public StyleBasicBurger() {
        name = "Style Basic Burger";
        price = 6.99;
    }

    @Override
    public void prepare() {
        System.out.println("Style kitchen preparing basic burger with signature sauce");
    }

    @Override
    public void cook() {
        System.out.println("Grilling on Style's special grill for 4 minutes");
    }

    @Override
    public void pack() {
        System.out.println("Packing in Style's eco-friendly wrapper");
    }
}

class StyleGarlicBurger extends Burger {
    public StyleGarlicBurger() {
        name = "Style Garlic Burger";
        price = 8.99;
    }

    @Override
    public void prepare() {
        System.out.println("Style kitchen preparing garlic burger with roasted garlic");
    }

    @Override
    public void cook() {
        System.out.println("Grilling garlic burger with Style's garlic seasoning");
    }

    @Override
    public void pack() {
```

```
        System.out.println("Packing in Style's garlic-themed packaging");
    }
}

// King Burger Products
class KingBasicBurger extends Burger {
    public KingBasicBurger() {
        name = "King Basic Burger";
        price = 7.99;
    }

    @Override
    public void prepare() {
        System.out.println("King kitchen preparing basic burger with royal spices");
    }

    @Override
    public void cook() {
        System.out.println("Flame-grilling on King's royal grill for 5 minutes");
    }

    @Override
    public void pack() {
        System.out.println("Packing in King's golden wrapper");
    }
}

class KingGarlicBurger extends Burger {
    public KingGarlicBurger() {
        name = "King Garlic Burger";
        price = 9.99;
    }

    @Override
    public void prepare() {
        System.out.println("King kitchen preparing garlic burger with royal garlic blend");
    }

    @Override
    public void cook() {
        System.out.println("Flame-grilling garlic burger with King's secret technique");
    }

    @Override
    public void pack() {
        System.out.println("Packing in King's royal garlic box");
    }
```

```java
}

// Factory Method Demo
public class FactoryMethodDemo {
    public static void main(String[] args) {
        // Create different factory instances
        BurgerFactory styleFactory = new StyleBurgerFactory();
        BurgerFactory kingFactory = new KingBurgerFactory();

        // Order from Style factory
        System.out.println("=== Style Burger Orders ===");
        Burger styleBurger1 = styleFactory.orderBurger("basic");
        Burger styleBurger2 = styleFactory.orderBurger("garlic");

        // Order from King factory
        System.out.println("\n=== King Burger Orders ===");
        Burger kingBurger1 = kingFactory.orderBurger("basic");
        Burger kingBurger2 = kingFactory.orderBurger("garlic");

        // Display results
        System.out.println("\n=== Order Summary ===");
        System.out.println(styleBurger1.getName() + " - $" + styleBurger1.getPrice());
        System.out.println(styleBurger2.getName() + " - $" + styleBurger2.getPrice());
        System.out.println(kingBurger1.getName() + " - $" + kingBurger1.getPrice());
        System.out.println(kingBurger2.getName() + " - $" + kingBurger2.getPrice());
    }
}
```

🏭 Abstract Factory Pattern

For creating families of related products:

Code Implementation

```java
```

```
// Abstract Factory for creating families of burger components
interface BurgerComponentFactory {
    Bun createBun();
    Patty createPatty();
    Sauce createSauce();
}

// Product families
interface Bun {
    String getType();
}

interface Patty {
    String getType();
}

interface Sauce {
    String getType();
}

// Style family products
class StyleBun implements Bun {
    public String getType() { return "Style Artisan Bun"; }
}

class StylePatty implements Patty {
    public String getType() { return "Style Organic Beef Patty"; }
}

class StyleSauce implements Sauce {
    public String getType() { return "Style Signature Sauce"; }
}

// King family products
class KingBun implements Bun {
    public String getType() { return "King Royal Sesame Bun"; }
}

class KingPatty implements Patty {
    public String getType() { return "King Flame-Grilled Patty"; }
}

class KingSauce implements Sauce {
    public String getType() { return "King Special Sauce"; }
}
```

```java
// Concrete Factories
class StyleComponentFactory implements BurgerComponentFactory {
    public Bun createBun() { return new StyleBun(); }
    public Patty createPatty() { return new StylePatty(); }
    public Sauce createSauce() { return new StyleSauce(); }
}

class KingComponentFactory implements BurgerComponentFactory {
    public Bun createBun() { return new KingBun(); }
    public Patty createPatty() { return new KingPatty(); }
    public Sauce createSauce() { return new KingSauce(); }
}

// Burger that uses components
class ComponentBurger {
    private Bun bun;
    private Patty patty;
    private Sauce sauce;

    public ComponentBurger(BurgerComponentFactory factory) {
        bun = factory.createBun();
        patty = factory.createPatty();
        sauce = factory.createSauce();
    }

    public void display() {
        System.out.println("Burger made with:");
        System.out.println("- " + bun.getType());
        System.out.println("- " + patty.getType());
        System.out.println("- " + sauce.getType());
    }
}

// Abstract Factory Demo
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        // Create Style burger
        BurgerComponentFactory styleFactory = new StyleComponentFactory();
        ComponentBurger styleBurger = new ComponentBurger(styleFactory);

        System.out.println("=== Style Burger ===");
        styleBurger.display();

        // Create King burger
        BurgerComponentFactory kingFactory = new KingComponentFactory();
        ComponentBurger kingBurger = new ComponentBurger(kingFactory);
```

```
        System.out.println("\n=== King Burger ===");
        kingBurger.display();
    }
}
```

🔍 Comparison of Factory Patterns

Aspect	Simple Factory	Factory Method	Abstract Factory
Complexity	Simple	Medium	Complex
Flexibility	Low	High	Very High
Open/Closed	Violates	Follows	Follows
Use Case	Single product family	Multiple implementations	Related product families
Inheritance	No inheritance	Uses inheritance	Uses composition
Products	Single type	Single type variants	Multiple related types

---

🌟 Real-World Applications

1. GUI Framework

java

```
// Different UI factories for different operating systems
interface UIFactory {
    Button createButton();
    Menu createMenu();
}

class WindowsUIFactory implements UIFactory { /* ... */ }
class MacUIFactory implements UIFactory { /* ... */ }
class LinuxUIFactory implements UIFactory { /* ... */ }
```

2. Database Connection

java

```
// Factory for different database connections
class DatabaseFactory {
    public static Connection createConnection(String type) {
        switch(type) {
            case "mysql": return new MySQLConnection();
            case "postgresql": return new PostgreSQLConnection();
            case "oracle": return new OracleConnection();
        }
    }
}
```

3. Payment Processing

```
java

// Factory for different payment processors
interface PaymentProcessorFactory {
    PaymentProcessor createProcessor();
}

class StripeFactory implements PaymentProcessorFactory { /* ... */ }
class PayPalFactory implements PaymentProcessorFactory { /* ... */ }
```

💡 Best Practices

✅ When to Use Factory Patterns

1. **Simple Factory**: When you have a simple object creation logic
2. **Factory Method**: When you need different implementations of the same interface
3. **Abstract Factory**: When you need to create families of related objects

🎯 Implementation Guidelines

1. **Keep factories focused**: Single responsibility for object creation
2. **Use dependency injection**: Pass factories as dependencies
3. **Consider caching**: Cache expensive objects when appropriate
4. **Handle errors gracefully**: Provide meaningful error messages
5. **Document factory contracts**: Clear documentation for factory methods

⚠️ Common Pitfalls

1. **Over-engineering**: Don't use factory for simple object creation
2. **God factories**: Avoid factories that create too many different types

3. **Tight coupling**: Ensure factory doesn't depend on concrete products
4. **Missing validation**: Always validate input parameters

📚 Key Takeaways

1. **Factory patterns encapsulate object creation logic**
2. **They promote loose coupling between client and products**
3. **Each factory type serves different complexity levels**
4. **Choose the right factory pattern based on your needs**
5. **They support SOLID principles, especially Open/Closed**

Remember the Factory Hierarchy

> Simple Factory → Factory Method → Abstract Factory
> (Increasing complexity and flexibility)

The Factory Design Pattern is fundamental to creating flexible, maintainable code that can adapt to changing requirements without major refactoring!