Abstract Factory Design Pattern

The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It helps to ensure that a group of related objects is used together, which can be useful when working with complex systems involving multiple types of objects.

Use Case:

Imagine you're developing a user interface system that needs to work with different themes (e.g., light theme and dark theme). Each theme has its own set of buttons, checkboxes, and text fields. The Abstract Factory Pattern can be used to create these components without tightly coupling the client code to specific classes for each theme.

Components:

1. Abstract Factory (UIFactory): Declares an interface for operations that create abstract product objects.
2. Concrete Factories (LightThemeFactory, DarkThemeFactory): Implement the operations to create concrete product objects.
3. Abstract Products (Button, Checkbox): Declare interfaces for different types of products.
4. Concrete Products (LightButton, DarkButton, LightCheckbox, DarkCheckbox): Implement the abstract product interfaces to create specific products.
5. Client: Uses the abstract factory to create objects without being concerned about their concrete classes.

Example: UI Theme System

1. Abstract Products (Button, Checkbox):

```java
public interface Button {
    void paint();
}

public interface Checkbox {
```

```
    void paint();
}
```

2. Concrete Products for Light Theme (LightButton, LightCheckbox):

```java
public class LightButton implements Button {
    @Override
    public void paint() {
        System.out.println("Rendering a light-themed button.");
    }
}

public class LightCheckbox implements Checkbox {
    @Override
    public void paint() {
        System.out.println("Rendering a light-themed checkbox.");
    }
}
```

3. Concrete Products for Dark Theme (DarkButton, DarkCheckbox):

```java
public class DarkButton implements Button {
    @Override
    public void paint() {
        System.out.println("Rendering a dark-themed button.");
```

```
    }
}

public class DarkCheckbox implements Checkbox {
    @Override
    public void paint() {
        System.out.println("Rendering a dark-themed checkbox.");
    }
}
```

4. Abstract Factory (UIFactory):

```java
public interface UIFactory {
    Button createButton();
    Checkbox createCheckbox();
}
```

5. Concrete Factories for Light and Dark Themes:

```java
public class LightThemeFactory implements UIFactory {
    @Override
    public Button createButton() {
        return new LightButton();
    }
```

```
    @Override
    public Checkbox createCheckbox() {
        return new LightCheckbox();
    }
}

public class DarkThemeFactory implements UIFactory {
    @Override
    public Button createButton() {
        return new DarkButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new DarkCheckbox();
    }
}
```

6. Client Code:

```java
public class AbstractFactoryDemo {
    private Button button;
    private Checkbox checkbox;

    public AbstractFactoryDemo(UIFactory factory) {
        button = factory.createButton();
```

```
        checkbox = factory.createCheckbox();
    }

    public void renderUI() {
        button.paint();
        checkbox.paint();
    }

    public static void main(String[] args) {
        UIFactory lightFactory = new LightThemeFactory();
        AbstractFactoryDemo lightUI = new AbstractFactoryDemo(lightFactory);
        lightUI.renderUI();

        UIFactory darkFactory = new DarkThemeFactory();
        AbstractFactoryDemo darkUI = new AbstractFactoryDemo(darkFactory);
        darkUI.renderUI();
    }
}
```

Key Points:

- Abstract Factory: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- Decoupling: It decouples the client code from the concrete classes of the products.
- Consistency: Ensures that products from the same family (e.g., buttons and checkboxes from the same theme) are used together, providing a consistent look and feel.