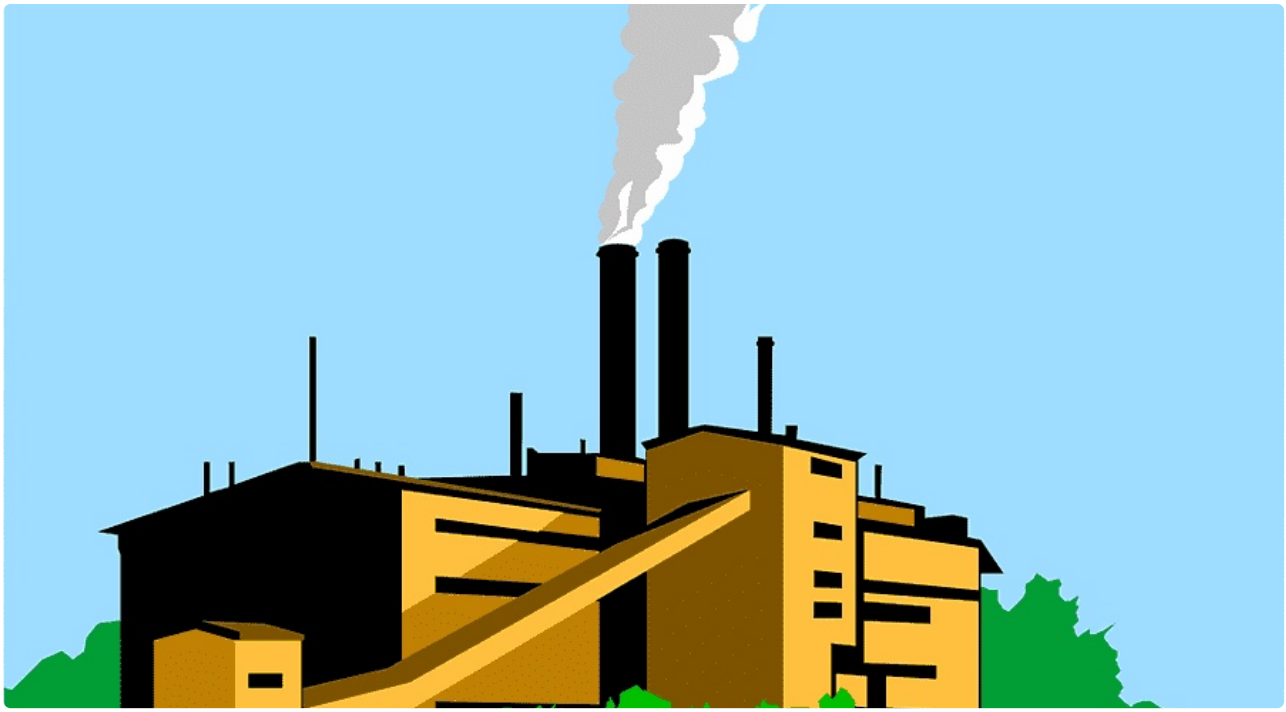
팩토리 메소드 패턴 (Factory Method Pattern)



팩토리 메소드 패턴의 개념과 역할

팩토리 메소드 패턴(Factory Method Pattern)은 객체를 생성하기 위한 인터페이스를 정의하고, 객체의 생성을 서브 클래스에서 결정하도록 하는 패턴입니다. 객체 생성 코드를 클라이언트로부터 분리하여 객체를 생성하는 방법을 제공합니다.

팩토리 메소드 패턴에서는 추상 팩토리(Abstract Factory)와 구체적인 팩토리(Concrete Factory)를 사용합니다. 추상 팩토리는 객체 생성을 위한 인터페이스를 정의하고, 구체적인 팩토리는 추상 팩토리를 상속받아 구현합니다. 구체적인 팩토리는 객체를 생성하기 위한 메소드를 구현하는데, 이를 팩토리 메소드라고 합니다.

팩토리 메소드 패턴은 객체의 생성과정에서 클라이언트와 객체 생성 코드를 분리하여 유연성을 제공합니다. 클라이언트는 구체적인 객체 생성 방법을 알 필요가 없으며, 객체 생성 과정이 변경되더라도 클라이언트 코드에 영향을 미치지 않습니다. 또한, 다형성을 이용하여 객체를 생성하는 코드를 추상화하므로 코드의 재사용성이 높아집니다.

팩토리 메소드 패턴의 구성 요소

Creator(생성자) 추상 클래스 또는 인터페이스

- 제품 객체를 생성하는 팩토리 메소드를 선언하는 추상 클래스 또는 인터페이스입니다. 이를 통해 제품 객체를 생성하는 책임을 서브클래스에게 위임합니다.
- Creator는 제품 객체의 생성 메소드를 선언하며, 이 메소드를 통해 객체를 생성합니다. 이 메소드는 추상 메소드로 선언됩니다.

Concrete Creator(구체적인 생성자) 클래스

- Creator 인터페이스를 구현하여 제품 객체를 생성하는 구체적인 클래스입니다. 구체적인 제품을 생성하는 책임을 갖습니다.

Product(제품) 추상 클래스 또는 인터페이스

- 팩토리 메소드 패턴에서 생성되는 객체의 추상 클래스 또는 인터페이스입니다. 이 클래스나 인터페이스는 구체적인 클래스의 공통 기능을 정의합니다.

Concrete Product(구체적인 제품) 클래스

- Product 인터페이스를 구현하여 구체적인 제품을 생성하는 클래스입니다. 제품의 구체적인 속성과 동작을 구현합니다.

팩토리 메소드 패턴의 동작 방식

팩토리 메소드 패턴은 객체를 생성하는 인터페이스를 정의하고, 객체의 생성을 서브클래스에서 결정하도록 하는 디자인 패턴입니다.

일반적으로 팩토리 메소드 패턴은 다음과 같은 구성 요소로 이루어집니다.

1. Creator: 객체 생성을 담당하는 클래스로, 팩토리 메소드를 포함합니다. 실제 객체 생성을 담당하지 않고, 팩토리 메소드를 이용해 객체 생성을 요청합니다. Creator 클래스는 추상 클래스로 구현될 수 있습니다.
2. ConcreteCreator: 객체 생성을 구현하는 클래스로, Creator 추상 클래스를 구현합니다. 구체적으로 어떤 객체를 생성할지 결정합니다.
3. Product: Creator 클래스가 생성하는 객체의 인터페이스입니다.
4. ConcreteProduct: Product 인터페이스를 구현하는 클래스로, 실제 객체입니다.

팩토리 메소드 패턴은 다음과 같은 방식으로 동작합니다.

1. Creator 추상 클래스에서 팩토리 메소드를 정의합니다.
2. ConcreteCreator 클래스에서 팩토리 메소드를 구현합니다. 이때 ConcreteCreator 클래스는 Product 인터페이스를 구현하는 ConcreteProduct 객체를 생성합니다.
3. Client는 Creator 추상 클래스를 사용해 객체를 생성합니다. Creator 클래스는 팩토리 메소드를 이용해 적절한 ConcreteProduct 객체를 반환합니다.
4. ConcreteProduct 객체는 Product 인터페이스를 구현하므로, Client는 ConcreteProduct 객체를 Product 인터페이스로 다룰 수 있습니다.

팩토리 메소드 패턴을 사용하면, 객체 생성 코드를 다른 코드와 분리할 수 있습니다. 객체 생성 방식을 변경하고자 할 때, 새로운 ConcreteCreator 클래스를 추가하면 됩니다. 이를 통해 코드 유연성이 향상됩니다. 또한, 팩토리 메소드 패턴을 사용하면 생성할 객체를 선택하는 코드를 간결하게 유지할 수 있습니다.

팩토리 메소드 패턴과 전략 패턴의 차이점

팩토리 메소드 패턴과 전략 패턴은 모두 객체 생성을 추상화하는 디자인 패턴이지만, 목적과 사용 방법에서 차이가 있습니다.

팩토리 메소드 패턴은 객체 생성에 대한 책임을 서브 클래스로 분리하여, 어떤 클래스의 인스턴스를 생성할지에 대한 결정을 서브 클래스에서 하도록 합니다. 이렇게 하면 객체 생성 코드를 사용하는 클라이언트 코드와 분리하여, 유연성을 높일 수 있습니다. 이는 특히 어떤 클래스의 인스턴스를 생성할지에 대한 결정이 런타임 시점에 이루어져야 할 경우 유용합니다.

반면에 전략 패턴은 알고리즘의 인터페이스와 구현을 분리하여, 알고리즘을 동적으로 변경할 수 있도록 합니다. 이는 동일한 문제를 다양한 방법으로 해결할 필요가 있는 경우 유용합니다. 전략 패턴은 객체 생성과는 관련이 없습니다.

즉, 팩토리 메소드 패턴은 객체 생성에 대한 책임을 분리하여 유연성을 높이는 디자인 패턴이고, 전략 패턴은 알고리즘을 동적으로 변경할 수 있도록 하는 디자인 패턴입니다.

팩토리 메소드 패턴의 장단점

장점

- 객체 생성과 관련된 코드를 별도의 클래스로 분리하여 캡슐화하고, 객체 생성에 필요한 인터페이스를 정의함으로써 유연성과 확장성을 높일 수 있습니다.
- 객체 생성에 대한 책임과 결정을 서브클래스에 위임함으로써, 클라이언트 코드가 객체의 구체적인 클래스에 의존하지 않아도 되므로 유지보수성을 높일 수 있습니다.
- 객체 생성 과정에서 중복된 코드를 제거하고, 코드 재사용성을 높일 수 있습니다.

단점

- 팩토리 클래스의 추가로 인해 코드의 복잡도가 증가할 수 있습니다.
- 객체를 생성하는데 필요한 팩토리 클래스가 많아질 경우, 코드의 양이 많아질 수 있습니다.

팩토리 메소드 패턴의 구현 방법과 주의사항

1. 팩토리 메소드 패턴은 객체 생성을 위해 인터페이스를 사용합니다. 이 때 인터페이스의 변경은 해당 팩토리와 그 팩토리를 사용하는 모든 클래스에 영향을 미칩니다. 따라서 인터페이스 설계에 주의해야 합니다.
2. 팩토리 메소드 패턴에서는 서브 클래스에서 팩토리 메소드를 구현합니다. 이 때 서브 클래스에서 팩토리 메소드를 재정의하지 않고 기존의 팩토리 메소드를 사용하는 경우도 있습니다. 이런 경우 상위 클래스에서 정의한 알고리즘을 변경하기 어렵습니다.
3. 팩토리 메소드 패턴은 객체 생성을 위한 추상 클래스와 팩토리 메소드를 제공합니다. 이 때 추상 클래스의 구현은 템플릿 메소드 패턴과 유사한 형태를 가집니다. 따라서 팩토리 메소드 패턴과 템플릿 메소드 패턴을 함께 사용하는 경우가 있습니다. 이 경우 클래스 구조가 복잡해질 수 있습니다.