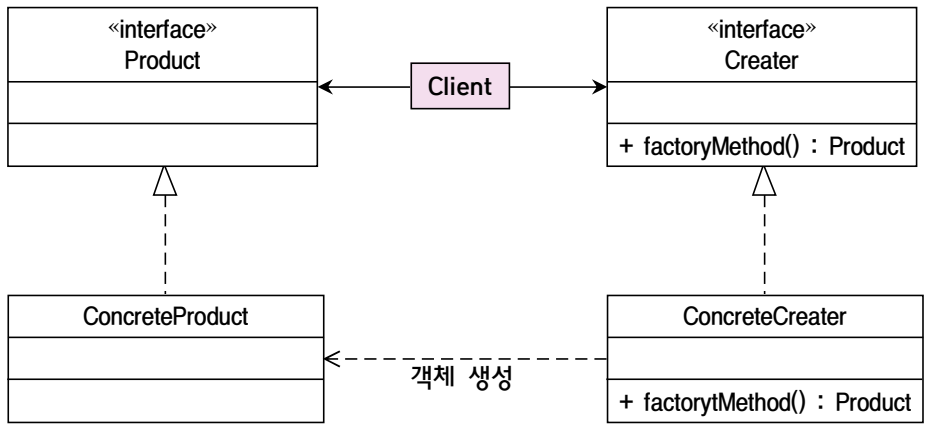
4. 팩토리 메서드 패턴

팩토리 메서드 패턴(factory method pattern) : 객체 생성을 다른 클래스에 **위임**하는 패턴

〈팩토리 매서드 패턴 클래스 다이어그램〉



- 객체를 사용할 Client가 직접 객체를 생성하지 않고 Creater(ConcreteCreater)에게 위임
- ConcreteProduct는 객체가 생성되는 대상이다.(객체 생성 주체: ConcreteCreater)

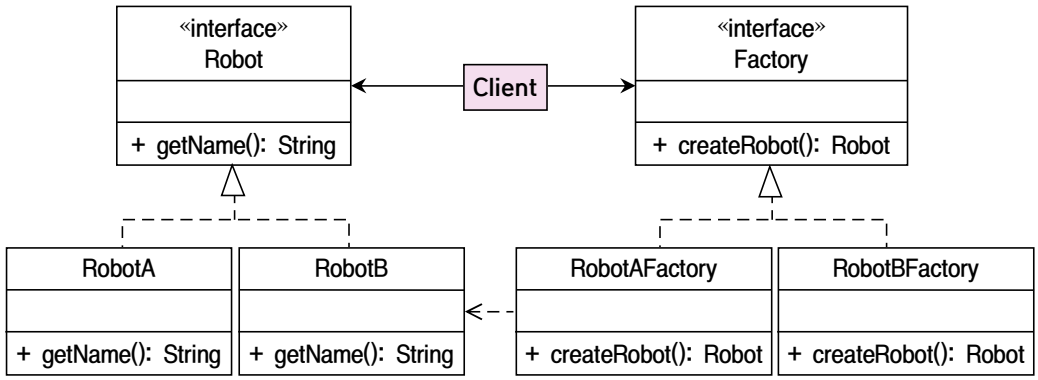
- **부모클래스**: 객체를 생성하는 팩토리 메서드의 인터페이스를 정의한다. - factoryMethod()
- **자식클래스**: 팩토리 메서드를 오버라이딩하여 실제 생성할 구체적인 객체를 결정한다.
- 팩토리 메서드 패턴은 사용할 객체를 직접 생성하지 않는다.
- 팩토리 메서드 패턴은 **객체 생성**을 **다른 클래스**에 **위임**하는 패턴이다.
- 팩토리 메서드 패턴은 **'객체 생성하는 시점을 자식클래스로 미루는 패턴이다'**라고도 한다.
- 팩토리 메서드 패턴은 **분기**에 따른 **객체 생성**을 직접 수행하지 않는다.
- 책임 분산: 하나의 클래스가 모든 객체를 생성하지 않는다.
- 개방-폐쇄 원칙(OCP) 준수: 새로운 객체가 필요하면 기존 코드를 수정할 필요가 없다.

// 팩토리 메서드 패턴 장단점

장점	• 객체 생성 코드를 실제 사용하는 코드와 분리해 유지보수성이 좋다.**(느슨한 결합도)** • 기존 코드를 수정하지 않고 새로운 유형의 제품을 쉽게 추가할 수 있다.
단점	• 패턴 적용은 새로운 자식클래스들을 많이 만들어야 하므로 코드가 복잡해질 수 있다.

예제	팩토리 매서드 패턴 - 공장에서 로봇 A와 로봇 B 만들기

〈팩토리 매서드 패턴 클래스 다이어그램〉



- 객체를 사용할 Client가 직접 객체를 생성하지 않고 Factory(구체적인 Factory 클래스)에게 위임
- RobotA, RobotB는 객체가 생성되는 대상이다.(객체 생성 주체: RobotAFactory RobotBFactory)

↓
↓ 자바로 구현하면
↓

```
interface Robot { public String getName(); }        // 인터페이스(로봇)
class RobotA implements Robot { public String getName() { return "로봇 A"; } }
class RobotB implements Robot { public String getName() { return "로봇 B"; } }
interface Factory { public Robot createRobot(); } // 팩토리 인터페이스: 생성 메서드 선언
class RobotAFactory implements Factory {                  // 로봇 A를 만드는 전용 공장
  public Robot createRobot() { return new RobotA(); }   // 로봇 A 객체 생성
}
class RobotBFactory implements Factory {                  // 로봇 B를 만드는 전용 공장
  public Robot createRobot() { return new RobotB(); }   // 로봇 B 객체 생성
}
public class Client {
  public static void main(String args[]) {
    Factory factoryA = new RobotAFactory();       // RobotAFactory 객체 생성
    Robot robotA = factoryA.createRobot();        // Client가 로봇 A 객체 생성 위임
    System.out.println(robotA.getName());         // 출력: 로봇 A
    Factory factoryB = new RobotBFactory();       // RobotBFactory() 객체 생성
    Robot robotB = factoryB.createRobot();        // Client가 로봇 B 객체 생성 위임
    System.out.println(robotB.getName());         // 출력: 로봇 B
  }
}
```

기출문제 분석

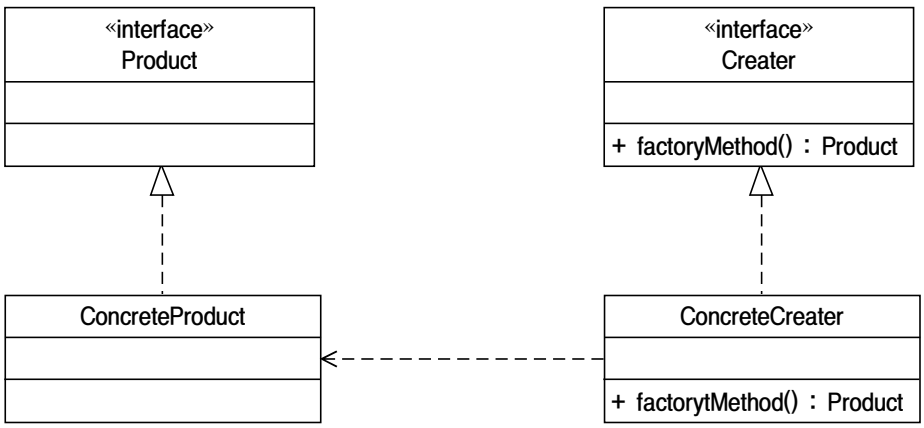
1. 다음 설명에 해당하는 소프트웨어 디자인 패턴으로 가장 적합한 것은? [2023년 군무 7급]

상위클래스에서 객체를 생성하는 인터페이스를 정의하고,
하위클래스에서 인스턴스를 생성하도록 하는 방식이다.

---

① 복합체 패턴(composite pattern)
② 팩토리 메서드 패턴(factory method pattern)
③ 적응자 패턴(adaptor pattern)
④ 데코레이터 패턴(decorator pattern)

♤ **팩토리 메서드 패턴**



- 팩토리 메서드 패턴은 사용할 객체를 직접 생성하지 않는다.
- 팩토리 메서드 패턴은 **객체 생성**을 **다른 클래스**에 **위임**하는 패턴이다.
- 팩토리 메서드 패턴은 **'객체 생성하는 시점을 자식클래스로 미루는 패턴이다.**'라고도 한다.
- 팩토리 메서드 패턴은 객체를 만드는 공장(Factory)을 만드는 패턴이다.
- 팩토리 메서드 패턴은 **분기**에 따른 **객체 생성**을 직접하지 않고, 팩토리 클래스에 **위임**하여 팩토리 클래스가 객체를 생성하도록 하는 방식을 말한다.
- 팩토리(Factory)는 말 그대로 객체를 만드는 공장을 의미한다.
- 객체를 생성하는 키워드 new는 객체를 생성하는 클래스에 존재하게 된다.(당연)
- 팩토리 메서드 패턴은 조건에 따라 객체를 다르게 생성해야 할 때 사용할 수 있다.
- 객체마다 하는 일이 다르므로 조건에 따라 객체를 다르게 생성하는 것은 당연한 일이다.

정답 : ②

2. 보기>에서 설명하는 디자인 패턴은? [2019년 서울 7급]

----<보기>----------

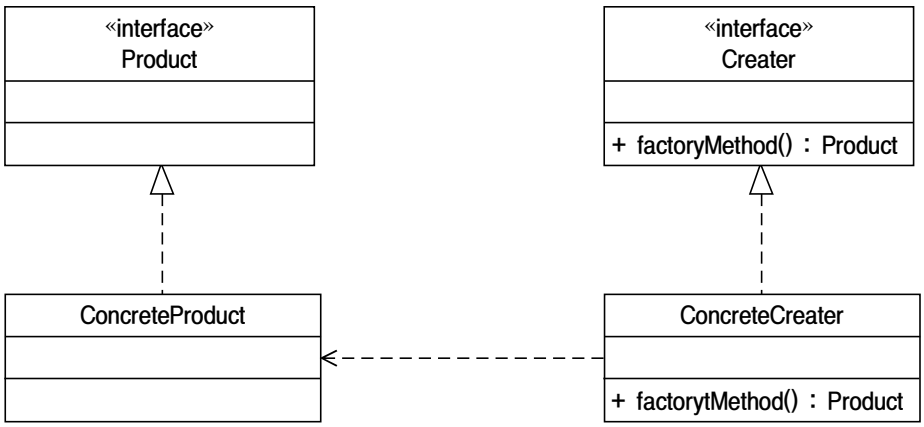
- 객체 생성 인터페이스를 정의하기 위한 디자인 패턴이다.
- 상위클래스에서는 객체를 만드는 인터페이스를 정의하고, 하위클래스에서는 구체적인 인스턴스를 생성하도록 한다.
- 객체를 생성하는 인터페이스와 실제 객체를 생성하는 클래스를 분리할 수 있다.

---

① factory method 패턴　② prototype 패턴
③ builder 패턴　④ abstraction factory 패턴

♤ 디자인 패턴

〈팩토리 매서드 패턴 클래스 다이어그램〉



- 팩토리 메서드 패턴은 사용할 객체를 **직접 생성**하지 않는다.
- 팩토리 메서드 패턴은 **객체 생성**을 **다른 클래스**에 **위임**하는 패턴이다.
- 팩토리 메서드 패턴은 **'객체 생성하는 시점을 자식클래스로 미루는 패턴이다.'**라고도 한다.
- 팩토리 메서드 패턴은 객체를 만드는 공장(Factory)을 만드는 패턴이다.
- 팩토리 메서드 패턴은 **분기**에 따른 **객체 생성**을 직접하지 않고, 팩토리 클래스에 **위임**하여 팩토리 클래스가 객체를 생성하도록 하는 방식을 말한다.
- 팩토리 메서드 패턴은 조건에 따라 객체를 다르게 생성해야 할 때 사용할 수 있다.

// 추상 팩토리(abstract factory)

- 구체적 클래스는 지정하지 않고, 서로 관련이 있는 구성요소 별로 **객체 집합**을 **생성**한다.
- 서로 독립적인 객체들의 집합을 생성할 수 있는 인터페이스를 제공하는 패턴이다.
- 팩토리 메서드 패턴을 좀 더 캡슐화한 방식이라고 볼 수 있다.

정답 : ①

3. GoF(Gang of Four)가 제시한 디자인 패턴에 대한 설명으로 가장 옳지 않은 것은? [2021년 서울 7급]

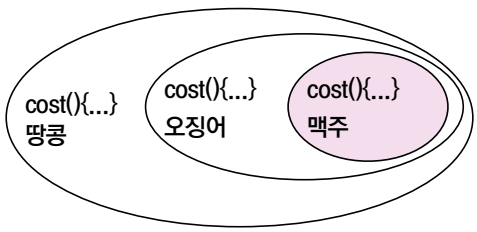
① Factory Method - 객체를 생성하는 처리를 파생클래스로 분리하여 처리하도록 캡슐화하는 패턴
② Adapter - 한 클래스의 인터페이스를 클라이언트에서 필요로 하는 인터페이스로 변환해주는 패턴
③ State - 객체의 상태에 따라 객체의 행위 내용을 변경 해주는 패턴
④ Strategy - 이미 고정된 자료구조에 행위를 쉽게 추가할 수 있도록 해주는 패턴

♤ **디자인 패턴**

• Strategy - 이미 고정된 자료구조에 행위를 쉽게 추가할 수 있도록 해주는 패턴**(×)**
→ **전략 패턴 : 알고리즘 교체**에 유용(알고리즘 변형이 필요한 경우에 유용)
• **데코레이터 패턴 : 상속을 사용하지 않고 객체의 기능을 동적으로 확장**할 수 있다.

// 전략 패턴
• 전략 패턴은 알고리즘을 객체화하여 같은 문제에 **다양한 알고리즘**을 적용할 수 있다.
• 전략 패턴은 알고리즘을 사용하는 클라이언트와 독립적으로 알고리즘을 변경할 수 있다.
• 전략 패턴은 참조하는 객체가 자주 변경되는 경우에 적용할 수 있다.
• 전략 패턴은 위임(delegation)을 통해서 어떤 행동을 사용할지 것인지 결정한다.

// 데코레이터 패턴
• 데코레이터 패턴은 **상속을 사용하지 않고 객체의 기능을 동적으로 확장**할 수 있다.
• 데코레이터 패턴은 객체의 추가적인 요건을 동적으로 추가한다.
• 데코레이터 패턴은 기존 코드를 수정하지 않고도 행동을 확장할 수 있다.
• 데코레이터 패턴은 주어진 상황에 따라 어떤 객체에 책임을 덧붙이는 패턴이다.
• 데코레이터 패턴(decorator pattern)은 래퍼 패턴(wrapper pattern)이라고도 한다.



주어진 그림은
맥주를 주문한 후에 오징어와 땅콩을 추가로 주문한 것을 나타낸다.

정답 : ④

4. 다음 상황에 가장 적절한 디자인 패턴은? [2026년 군무 7급]

<보기>	어느 소프트웨어 개발팀은 PDF, Word, HTML 형식의 문서를 생성하는 시스템을 개발하고 있었다. 초기 구현에서는 문서 형식에 따라 조건문(if/switch)을 사용하여 객체를 직접 생성하였다. 그러나 새로운 문서 형식이 추가될 때마다 기존 생성 코드를 반복적으로 수정해야 했고, 유지보수 부담이 증가하였다. 이에 개발팀은 객체 생성 책임을 별도의 구조로 분리하여 새로운 문서 형식을 쉽게 확장할 수 있도록 시스템을 개선하였다.

① Factory Method Pattern
② Observer Pattern
③ Adapter Pattern
④ Decorator Pattern

♤ **팩토리 메서드 패턴(factory method pattern)**

팩토리 메서드 패턴

팩토리 메서드 패턴은 "객체 생성 코드를 별도의 메서드로 분리하여, 구체적인 클래스에 의존하지 않고 확장에 유연하게 대처"할 수 있다.

- 팩토리 메서드 패턴은 **객체 생성**을 **다른 클래스**에 **위임**하는 패턴이다.
- 책임 분산: 하나의 클래스가 모든 객체를 생성하지 않는다.
- 개방-폐쇄 원칙(OCP) 준수: 새로운 객체가 필요하면 기존 코드를 수정할 필요가 없다.
- 팩토리 메서드 패턴은 **분기**에 따른 **객체 생성**을 직접 수행하지 않는다.

문제점	• 초기 구현은 클라이언트 코드가 Pdf, Word, HTML 형식의 문서를 생성하는 구체적인 클래스들을 직접 알아야 한다. • 기존 코드에서는 새로운 문서 형식(예: Excel 등)이 추가될 때마다 객체를 생성하는 조건문(if/switch)을 직접 수정해야 한다. • 이는 기존 코드를 변경해야 하므로 시스템을 불안정하게 만든다.
↓	**팩토리 메서드 패턴을 적용하면**
해결책	• 팩토리 메서드 패턴을 적용하면, 클라이언트는 구체적인 클래스가 무엇인지 알 필요 없이 공통 인터페이스만 보고 코드를 작성할 수 있다. 즉, 의존성이 크게 줄어든다. • 팩토리 메서드 패턴을 적용하면, 새로운 문서 형식이 추가되어도 기존의 문서를 생성하는 코드 부분을 수정할 필요 없다. • 새로운 문서 형식이 추가되면 클라이언트는 새로운 문서 형식의 객체를 생성하도록 추가하면 된다. • 즉, 기존 코드를 수정하지 않고 새로운 문서 형식을 언제든지 쉽게 추가할 수 있다.

정답: ①