26. 디자인 패턴 개요

• 디자인 패턴(design pattern)은 건축 및 컴퓨터 과학에 사용되는 용어이다.
• 소프트웨어 개발에서 디자인 패턴을 **처음** 제시한 곳은 GoF(Gang of Four)이다.
• GoF는 "에릭 감마, 리차드 헬름, 랄프 존슨, 존 블리시디스" **4**명을 지칭한다.

// 디자인 패턴 – 용도에 따른 분류

유 형	특징 및 종류	
생성 패턴 (creational pattern)	//객체 생성 방식을 결정하는 패턴 – 5개	
	• 원형 패턴(prototype) • 싱글턴 패턴(singleton)	• 빌더 패턴(builder) • 팩토리 메서드 패턴(factory method) • 추상 팩토리 패턴(abstract factory)
구조 패턴 (structural pattern)	//객체를 조직화하는데 유용한 패턴(합성에 관여) – 7개	
	• 브리지 패턴(bridge) • 적응자 패턴(adaptor) • 복합체 패턴(composite) • 데코레이터 패턴(decorator)	• 프록시 패턴(proxy) • 퍼사드 패턴(facade) • 플라이웨이트 패턴(flyweight)
행위 패턴 (behavioral pattern)	//객체들의 상호작용을 조정 관리하는 패턴 – 11개	
	• 반복자 패턴(iterator) • 옵저버 패턴(observer) • 중재자 패턴(mediator) • 메멘토 패턴(memento) • 명령어 패턴(command) • 해석자 패턴(interpreter)	• 상태 패턴(state) • 전략 패턴(strategy) • 방문자 패턴(visitor) • 템플릿 메서드 패턴(template method) • 책임연쇄 패턴(chain of responsibility)

• 디자인 패턴 종류 = 5 + 7 + 11 = **23**(개) – GoF에서 발표한 것(1994년)
• 현재, 디자인 패턴은 **수천여**개 발표되었다.
• 본 교재에서는 GoF에서 발표한 **23**개의 디자인 패턴에 대해서 설명한다.

• 디자인 패턴은 설계 문제에 대한 해답을 문서화하기 위해 고안된 방법이다.
• 디자인 패턴은 **프로그램 개발에서 자주 나타나는 과제를 해결**하기 위한 방법 중 하나이다.
• 디자인 패턴은 과거 소프트웨어 개발 과정에서 발견된 설계의 노하우를 축적하여
• 이름을 붙여, 이후에 재사용하기 좋은 형태로 정리한 것이다.

2 http://cafe.daum.net/pass365(홍재연)

1. 원형(prototype)

미리 만들어진 객체를 **복제(clone)**하여 새로운 객체를 생성하는 패턴이다.
객체 생성에 **원형**이 되는 **견본을 이용하는 패턴**이다.
다수의 객체 생성 **비용**을 효과적으로 줄일 수 있다.

2. 싱글턴(singleton) - 단일체

어떤 클래스에 대한 객체(instance)는 **오직 하나임을 보장**하는 패턴이다.
생성된 객체(instance)에 접근할 수 있는 전역적인 접근점을 제공하는 패턴이다.

3. 빌더(builder)

빌더 패턴은 **복합객체를 생성하는 방법**과 **객체를 조립하는 방법**을 **분리**하는 것이 목적이다.
빌터 패턴은 복합객체를 생성하는 절차를 좀 더 세밀하게 나눌 수 있다.
동일한 객체 생성 절차에서 조립 방법에 따라 다른 결과를 만들 수 있게 하는 패턴이다.
즉, 객체 조립 순서에 따라 다른 결과를 만들 수 있게 하는 패턴이다.

4. 팩토리 메서드(factory method)

팩토리 메서드 패턴은 모든 **객체 생성을 팩토리 클래스에 위임하는 패턴**이다.
팩토리 메서드 패턴은 객체를 만드는 공장(팩토리 클래스)을 만드는 패턴이다.
팩토리 메서드 패턴은 **'객체 생성하는 시점을 자식클래스로 미루는 패턴이다.**'라고 한다.
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.

5. 추상 팩토리(abstract factory)

구체적 클래스는 지정하지 않고, **서로 관련이 있는 구성요소** 별로 **객체 집합을 생성**한다.
서로 독립적인 객체들의 집합을 생성할 수 있는 인터페이스를 제공하는 패턴이다.
많은 수의 연관된 **자식클래스**를 특정 그룹으로 묶어 한번에 교체할 수 있도록 한다.
팩토리 메서드 패턴을 좀 더 캡슐화한 방식이라고 볼 수 있다.

6. 퍼사드(facade)

Facade : 건물 **정면** 또는 **외관**을 의미한다.
퍼사드 패턴은 많은 분량의 코드에 접근할 수 있는 **단순한 인터페이스를 제공**한다.
퍼사드 패턴은 **서브시스템**의 인터페이스 집합에 대해서 하나의 **통합된 인터페이스**를 제공한다.
퍼사드는 서브시스템에 있는 객체들을 사용할 수 있도록 인터페이스 역할을 한다.
퍼사드는 서브시스템의 가장 **앞쪽**에 위치하여 인터페이스 역할을 한다.

7. 적응자(adapter)

적응자 패턴은 인터페이스를 사용자가 기대하는 **다른 인터페이스로 변환**하는 패턴이다.
적응자 패턴은 객체를 감싸서 다른 인터페이스를 제공한다.**(wrapper pattern이라고도 함)**
적응자 패턴은 인터페이스가 호환되지 않는 클래스들을 함께 이용할 수 있도록 다른 클래스의 인터페이스를 기존 인터페이스에 덧붙인다.
적응자 패턴은 **일관성 있는 인터페이스를 사용**할 수 있도록 한다.

8. 가교(bridge)

구현부에서 **추상층**을 분리하여 각각 독립적으로 변형할 수 있는 패턴이다.
추상과 **구현**을 **분리**하여 각각 독립적으로 변경해도 서로 영향을 주지 않는다.

9. 데코레이터(decorator) - 장식자

먼저, decorator의 뜻은 도장 및 도배업자, 장식가 등이다.
데코레이터 패턴은 **상속을 사용하지 않고 객체의 기능을 동적으로 확장**할 수 있다.
데코레이터 패턴은 주어진 상황 및 용도에 따라 어떤 객체에 책임을 덧붙이는 패턴이다.

10. 복합체(composite)

복합체 패턴(composite pattern)은 0개 또는 그 이상의 객체를 묶어 **하나의 객체로 이용**한다.
복합체 패턴은 복합객체와 단일객체를 같은 방법으로 사용할 수 있는 패턴이다.
복합체 패턴은 **트리구조**로 관리고자 할 때 자주 사용이 된다.
예 : 폴더(복합객체)와 파일(단일객체)을 같은 방법으로 사용할 수 있는 패턴이다.

11. 프록시(proxy)

프록시 패턴(proxy pattern)은 원본 객체를 대리하여 **대신** 처리하는 패턴이다.
프록시 패턴은 인터페이스를 이용한 **위임** 방식으로 구현한다.
프록시 패턴은 실제 기능을 수행하는 객체 대신 **가상**의 객체 사용한 **흐름제어** 패턴이다.

12. 플라이급(flyweight)

크기가 작은 객체가 여러 개 있을 때, 공유를 통해 이들을 효율적으로 지원하는 패턴이다.
객체의 상태를 공유 풀(pool)에 형성하여 **메모리 절약**하기 위한 패턴이다.

13. 전략(strategy)

전략 패턴(strategy pattern) : **알고리즘 교체**에 유용(알고리즘 변형이 필요한 경우에 유용)
전략 패턴은 알고리즘을 객체화하여 같은 문제에 다양한 알고리즘을 적용할 수 있다.

14. 반복자(iterator)

반복자 패턴은 어떤 **집합체의 구성요소들을 차례로 접근**하기 위한 것이다.
여기서, 집합체는 배열, 연결리스트, 스택, 큐 등을 의미한다.
반복자 패턴은 내부표현을 노출하지 않고, 집합체 원소들의 접근 방법을 제공하는 패턴이다.

15. 옵저버(observer) - 감시자

옵저버(observer)는 "보는 사람, 목격자, 관찰자, 감시자"라는 뜻이다.
옵저버 패턴은 어떤 **객체의 상태가 변할 때** 그 객체에 의존성을 가진 다른 객체들이 **그 변화를 통지받고 자동으로 갱신**될 수 있게 만드는 패턴이다.
옵저버 패턴은 객체 사이에 일대다의 의존관계를 정의하고, 어떤 객체의 상태가 변할 때 이 객체와 의존관계인 다른 객체들이 그 변화를 통지받고 자동 갱신될 수 있게 만드는 패턴이다.

16. 중재자(mediator)

중재자 패턴(mediator pattern)은 조정자 패턴이라고도 한다.
중재자 패턴을 사용하면 객체 사이의 통신은 중재자 객체 안에 함축된다.
객체들은 더 이상 다른 객체와 서로 직접 통신하지 않으며 대신 **중재자를 통해 통신**한다.
중재자 패턴은 **객체 사이의 관계가 매우 복잡**하여 객체 재사용에 부담 갈 경우에 사용한다.

17. 메멘토(memento)

먼저, 영어단어 memento는 (사람·장소를 기억하기 위한) 기념품이다.
메멘토 패턴은 객체의 상태 보존과 관련이 있다.
메멘토 패턴은 **객체를 이전 상태로 되돌릴 수 있는 기능**을 제공하는 패턴이다.
메멘토 패턴은 롤백(rollback)을 이용하여 실행을 취소한다.

18. 명령어(command)

명령어 패턴(command pattern)은 **요구사항(요청) 자체를 캡슐화** 한다.
명령어 패턴은 하나의 **명령(기능)**을 **객체**화한 패턴이다.
객체는 보관해 둘 수 있고, 전달할 수도 있다.
명령을 **객체**화하면 명령 자체를 데이터처럼 전달할 수 있고 저장해 둘 수도 있다.
명령어의 배치 실행, undo/redo, 우선순위별 명령어 실행 등이 가능하다.
명령어 패턴은 **클라이언트**, **명령**(command), **수신자**(receiver), **발동자(invoker)**로 구성된다.
발동자(invoker)는 데이터통신에서 **호출자**를 의미한다.
명령어 패턴을 적용하면 **의존성**을 제거할 수 있다.
즉, 명령어 패턴은 **호출자**와 **수신자** 클래스 사이의 **의존성을 제거**한다.

19. 해석자(interpreter)

해석자 패턴(interpreter pattern)은 언어적 **문법**을 **표현**하는 패턴이다.
해석자 패턴은 컴파일러 구현에 사용될 수 있다.**(미니 컴파일러 구현)**
해석자 패턴은 문법에 맞게 작성된 표현식을 해석한다.
해석자 패턴은 SQL과 같은 언어를 해석하기 위해 계층구조로 표현할 수 있다.
해석자 패턴은 하위 객체가 처리한 결과를 조합하여 새로운 결과를 만든다.
복합체 패턴이 사용되는 곳에 해석자 패턴을 사용할 수 있다.

20. 방문자(visitor)

방문자 패턴(visitor pattern)은 **데이터 구조**와 **데이터 처리**를 **분리**시키는 패턴이다.
어떤 클래스의 데이터 처리는 해당 클래스의 **메서드**를 이용하는 것이 당연한 것 같지만
방문자 패턴은 어떤 클래스의 데이터 처리를 해당 클래스의 메서드를 이용하지 않는다.
방문자 패턴은 어떤 클래스의 데이터 처리를 **새로운 클래스**를 **추가**하는 방식으로 구현한다.

21. 템플릿 메서드(template method)

템플릿 메서드 패턴은 **알고리즘의 뼈대를 정의**하는 패턴이다.
알고리즘 구조를 변경하지 않고 알고리즘의 각 단계들을 다시 정의할 수 있도록 한다.
각 단계의 처리를 자식클래스에서 재정의할 수 있다.

22. 책임 연쇄(chain of responsibility)

책임 연쇄 패턴(chain of responsibility pattern)은 **요청**을 처리할 수 있는 객체 집합 패턴이다.
책임 연쇄 패턴은 여러 책임들을 **동적**으로 연결해서 **순차적**으로 **실행**하는 패턴이다.
책임 연쇄 패턴의 각 객체는 **연결리스트** 구조의 **체인**을 형성한다.
책임은 무언가를 처리하는 **기능**을 말한다.
책임 연쇄 패턴은 **책임을 다른 객체에게 떠넘기는 것**과 같다.라고도 한다.
그런데, 책임 연쇄 패턴에서 각 객체는 자신이 맡은 책임만을 충실하게 수행한다.
책임 연쇄 패턴은 하나의 클래스가 **하나**의 **책임**만을 맡아야 하는 원칙을 준수한다.
책임 연쇄 패턴은 객체 간의 결합도를 낮추고, 요청을 처리할 객체를 동적으로 결정한다.
책임 연쇄 패턴은 각 객체가 책임을 나누어 처리하므로, **유지보수**성과 **확장성**을 높일 수 있다.

23. 상태(state)

상태 패턴(state pattern)은 어떤 **상태**를 클래스로 표현하여 **객체**로 취급하는 패턴이다.
상태 패턴은 객체의 **내부 상태**에 따라서 객체의 동작이 달라지는 패턴이다.
상태 패턴은 상태를 **조건문**으로 검사해서 동작을 달리하는 것이 아니고, 상태를 **객체**화하여 상태가 동작할 수 있도록 **위임**하는 패턴이다.

// 디자인 패턴 핵심 정리

주제(thema)	패턴명	핵심 내용
객체 생성	prototype	복사해서 객체를 생성
	singleton	하나의 객체만 생성
	builder	복잡한 객체를 조립
	abstract factory	관련된 객체 집합을 생성
자식클래스에 맡김	factory method	객체 생성을 자식클래스에 맡김
	template method	구체적 처리를 자식클래스에 맡김
분리해서 생각	bridge	추상(기능)과 구현을 분리
	strategy	알고리즘 교체가 용이
단순하게 함	facade	단순한 창구 제공
	mediator	하나의 중재자가 모두를 상대함
상태를 관리	observer	상태 변화를 통지
	memento	상태를 보존
	state	상태를 클래스로 표현
동일하게 취급	decorator	내용물과 장식을 동일하게 취급
	composite	내용물과 그릇을 동일하게 취급
	iterator	다른 집합체를 차례로 동일하게 접근
구조 안을 배회	visitor	구조 안을 돌아다니면서 일을 함
	chain of responsibility	책임을 다른 객체에게 떠넘김
낭비 제거	proxy	필요할 때 대리자를 만듦
	flyweight	동일한 것은 공유해서 낭비 제거
	adaptor	한 꺼풀 감싸서 재사용
클래스로 표현	command	명령을 클래스로 만듦
	interpreter	문법규칙을 클래스로 표현

// 디자인 패턴 - 클래스와 객체에 따른 분류

클래스 패턴	• 클래스 패턴은 **컴파일시간**에 관계가 결정됨 • 클래스 패턴은 **상속**을 통한 클래스 사이의 관계 설정 • 클래스 사이의 관계가 상속으로 어떻게 정의되는지를 취급 • 상속을 남용하면, 부작용을 초래한다.	• factory method pattern • interpreter pattern • template method pattern • adapter pattern
객체 패턴	• 객체 패턴은 객체 사이의 관계를 취급 • 일반적으로, 객체 패턴은 **실행시간**에 관계가 결정됨 • 객체가 실행시간에 동적으로 생성되므로 더 유연함 • 보통, 객체 사이의 관계는 **합성(composition)**으로 정의됨	• 나머지 패턴들

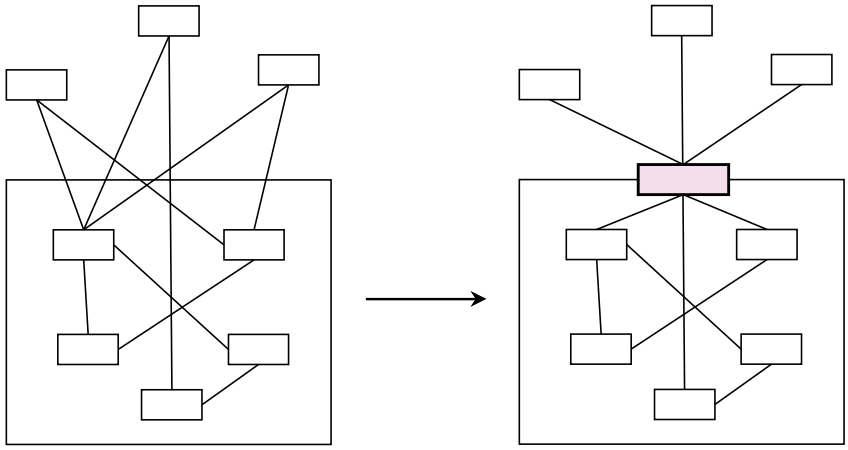
// 현재, 디자인 패턴 문제는 어떻게 출제되고 있는가?

① 서술식으로 출제

옵저버 패턴 (observer)	1대 다(多)의 객체 의존관계를 정의한 것으로 한 객체가 상태를 변화시켰을 때, 의존관계에 있는 다른 객체들에게 자동적으로 통지하고 변경시킨다.
장식자 패턴 (decorator)	상속을 사용하지 않고도 객체의 기능을 동적으로 확장할 수 있는 설계 패턴 어떤 객체에 책임(responsibility)을 동적으로 추가할 수 있도록 한다.
가교 패턴 (bridge)	추상화와 구현을 분리하여, 실행시간에 동적으로 다른 구현들을 사용할 있다. 구현에서 추상을 분리하여, 이들이 독립적으로 다양성을 가질 수 있도록 한다.
중재자 패턴 (mediator)	서로의 존재를 모르는 상태에서도 메시지를 주고받으며 협력할 수 있다. 객체의 상호작용을 캡슐화하는 객체를 정의한다.

- 서술식으로 출제되면, 영어단어 뜻만 알면 맞출 수 있는 것도 있다.
- 하지만, 명확하게 기술하지 않으면 중복된 답이 존재할 수 있으므로 조심해야 한다.
- 출제자가 명확하게 문제를 출제하는 것이 중요하다.

② 그림으로 출제



- 주어진 그림은 **퍼사드 패턴**(facade)에 해당하는 그림이다.
- 디자인 패턴은 비슷한 그림이 많으므로 출제자는 명확하게 출제해야 할 것이다.

③ 원시코드로 출제

- 과거에는 디자인 패턴 문제에서 **원시코드**는 출제되지 않았다.
- 현재는 디자인 패턴 문제에서 **원시코드**가 점점 더 많이 출제되고 있다.
- 디자인 패턴을 정확하게 이해하려면, **원시코드**를 분석할 수 있어야 한다.
- 말로 이야기하면 뜬구름 잡는 공부가 될 수 있다.

// 디자인 패턴 적용 규칙

① 결합(coupling)을 최소화한다.

- 어떤 **하나**의 클래스 변화가 **전체** 클래스를 변화시키지 않도록 해야 한다.
- 전지전능하신 클래스(god class)를 만들지 않는다.
- god class : 다수의 기능을 처리하는 클래스(나쁜 클래스)

② 상속(inheritance)이 아니라 위임(delegation)을 이용하여 프로그래밍 한다.

- 객체지향 프로그램에서 상속을 **남용**하는 것은 좋지 않다.
- **위임**은 상속의 문제점을 해결할 수 있는 대안이다.

③ 인터페이스(interface)를 이용하여 프로그래밍 한다.

- **인터페이스**를 이용한다는 것은 구현된 **클래스**를 직접 이용하지 않는다는 것이다.
- 다르게 말하면, 인터페이스에 선언된 메서드를 호출하는 원리로 프로그래밍 한다.
- 인터페이스를 이용하면, 구현한 클래스 내부 변화에 영향을 받지 않는다.
- 구현(implementation) 클래스를 직접 이용하면, 프로그램 변경이 어렵게 된다.
- 프로그램 변경이 어렵다는 것은 **유지보수**가 어렵다는 것이다.

---〈인터페이스〉---

- 인터페이스는 일련의 일관된 공통 특징과 의무의 선언을 나타내는 클래스이다.
- 인터페이스는 선언이므로 즉각적으로 사용할 수 없다.
- 인터페이스는 계약을 명시한다.
- 인터페이스를 실현하는 클래스는 인터페이스에 선언된 그 계약을 이행해야 한다.
- 인터페이스는 다수의 서로 다른 클래스에 의해 구현될 수 있다.

// 디자인 패턴 장단점

장점	단점
· 코드 품질이 향상된다.	· 잘못 사용된 패턴은 개발 생산성을 저하시킨다.
· 시스템 개발에 공통 언어 역할을 한다.	· 잘못 해석된 패턴은 재사용성을 저하시킨다.
· 유지보수가 용이하다.	· 잘못 사용된 패턴은 유지보수를 저하시킨다.
· 향후 변화에 대비할 수 있다.	· 설계자가 패턴을 숙지하는 오랜 시간이 필요하다.

탐구

인터페이스(interface)

먼저, 인터페이스(interface)는 잘 알고 있는 내용이다. 하지만 다시 한 번 더 정리한다.

인터페이스는 국어사전에 다음처럼 설명되어 있다.

〈인터페이스〉

① 서로 다른 두 시스템, 장치, 소프트웨어 따위를 서로 이어주는 부분. 또는 그런 **접속장치**
② 사용자인 **인간과 컴퓨터를 연결**하여 주는 장치. 키보드나 디스플레이 따위를 이른다.

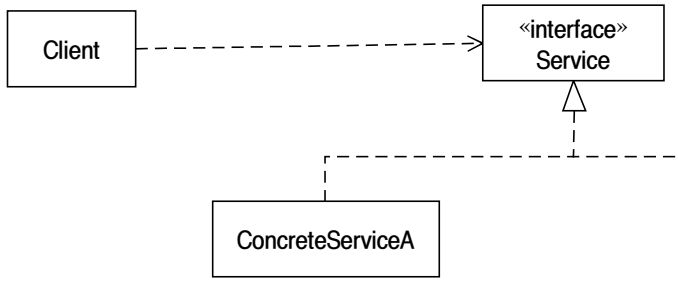
↓그러면, 프로그래밍 언어에서 인터페이스란?

〈프로그래밍 언어에서 인터페이스〉

인터페이스는 선언과 구현을 분리하고, 기능을 사용할 수 있는 통로이다.

디자인 패턴에서 인터페이스(interface) 사용은 기본이다. 관련 용어를 살펴본다.

〈인터페이스를 이용한 프로그래밍〉



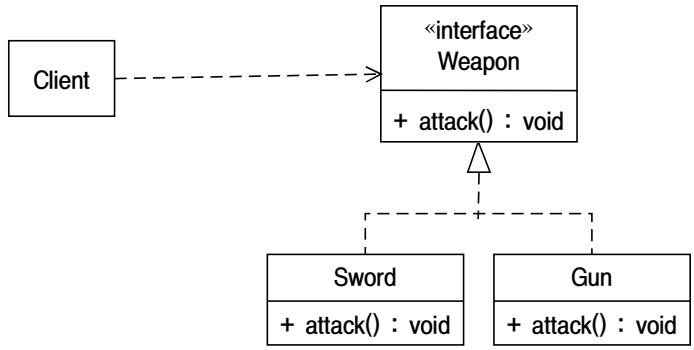
〈"GoF의 디자인패턴" 교재〉

Client	• Client는 객체와 **동작을 시작하는 클래스**이다. • Client 코드는 구체적인 클래스에 의존하지 않게 된다. • Client : 고객, 사용자, 의뢰인, 위임인 등
Service	• 인터페이스 Service는 **메서드를 선언**한다. • Service는 Client가 ConcreteServiceA를 사용하는 통로 역할을 한다.
ConcreteServiceA ConcreteServiceB	• 클래스 ConcreteServiceA는 인터페이스에 **선언된 메서드를 구현**한다. • 해서, 구현된 클래스 또는 구체적인 클래스라는 용어를 사용한다.

• 영어단어 Concrete는 '구체적인, 콘크리트로 된'이라는 의미를 가진다. (abstract와 반대)
• 해서, 인터페이스에 선언된 메서드를 구현하는 클래스 이름에 관례적으로 Concrete를 붙인다.

[예제 1] 인터페이스 사용과 디자인 패턴을 이해하기 위한 기본 프로그램

〈인터페이스가 포함된 UML의 클래스 다이어그램〉



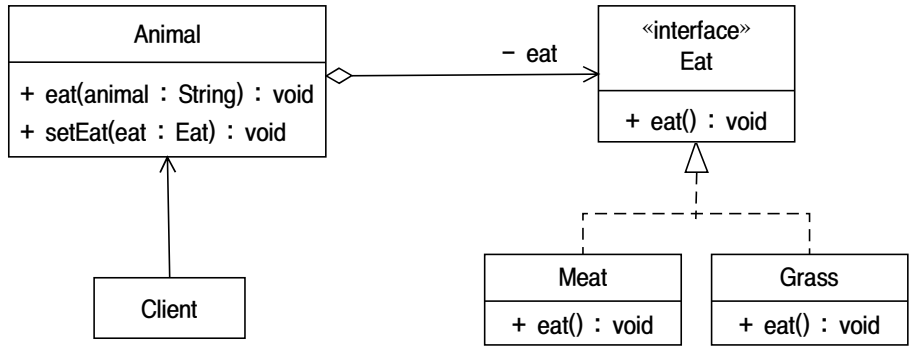
↓자바로 프로그래밍하면

〈자바에서 인터페이스 선언과 구현〉

```
interface Weapon                                    //인터페이스
{
  public void attack();                             //메서드 선언
}
class Sword implements Weapon                       //인터페이스 상속
{
  public void attack(){ System.out.println("검으로 공격"); }   //메서드 구현
}
class Gun implements Weapon                         //인터페이스 상속
{
  public void attack(){ System.out.println("총으로 공격"); }   //메서드 구현
}
public class Client
{
  public static void main(String args[]){
    Weapon s = new Sword();          //클래스 Sword의 객체 s 생성
    s.attack();                      //기능을 사용하는 통로 역할 (출력 : 검으로 공격)
    Weapon g = new Gun();            //클래스 Gun의 객체 g 생성
    g.attack();                      //기능을 사용하는 통로 역할 (출력 : 총으로 공격)
  }
}
```

[예제 2] 디자인 패턴을 이해하기 위한 기본 프로그램

〈인터페이스가 포함된 UML의 클래스 다이어그램〉



↓ 프로그램

〈자바에서 인터페이스 선언과 구현, 그리고 위임〉

```
interface Eat { public void eat(String animal); }          //인터페이스
class Meat implements Eat                                  //고기를 먹는 동물을 처리
{
    public void eat(String animal) { System.out.println("고기를 먹는 동물 : " + animal); }
}
class Grass implements Eat                                 //목초를 먹는 동물을 처리
{
    public void eat(String animal) { System.out.println("목초를 먹는 동물 : " + animal); }
}
class Animal {                                             //위임 처리 클래스
    private Eat eat;                                       //Eat 참조(연관)
    public void eat(String animal) { eat.eat(animal); }    //동물이 먹는 것을 위임 처리
    public void setEat(Eat eat) { this.eat = eat; }        //동물이 먹는 것을 설정
}
public class Client {
    public static void main(String args[]){
        Animal lion = new Animal();                        //사자 객체 생성
        lion.setEat(new Meat());                           //고기를 먹는 사자 객체 생성하여 호출
        lion.eat("사자");                                  //고기를 먹는 사자
        Animal bull = new Animal();                        //황소 객체 생성
        bull.setEat(new Grass());                          //목초를 먹는 황소 객체 생성하여 호출
        bull.eat("황소");                                  //목초를 먹는 황소
    }
}
```

기출문제 분석

1. ㉠에 들어갈 용어로 옳은 것은? [2018년 계리직]

(㉠)(은)는 유사한 문제를 해결하기 위해 설계들을 분류하고 각 문제 유형별로 가장 적합한 설계를 일반화하여 체계적으로 정리해 놓은 것으로 소프트웨어 개발에서 효율성과 재사용성을 높일 수 있다.

---

① 디자인 패턴
② 요구사항 정의서
③ 소프트웨어 개발 생명주기
④ 소프트웨어 프로세스 모델

♤ **디자인 패턴**

- 디자인 패턴은 설계 문제에 대한 해답을 문서화하기 위해 고안된 방법이다.
- 디자인 패턴은 **프로그램 개발에서 자주 나타나는 과제를 해결**하기 위한 방법 중 하나이다.
- 디자인 패턴은 과거 소프트웨어 개발 과정에서 발견된 설계의 노하우를 축적하여
- 이름을 붙여, 이후에 재사용하기 좋은 형태로 정리한 것이다.

정답 : ①

2. 디자인 패턴에 대한 설명으로 옳지 않은 것은? [2015년 지방 9급]

① 일반적으로 디자인 패턴을 이용하면 좋은 설계나 아키텍처를 재사용하기 쉬워진다.
② 패턴은 사용 목적에 따라서 생성 패턴, 구조 패턴, 행위 패턴으로 분류할 수 있다.
③ 생성 패턴은 빌더(builder), 추상 팩토리(abstract factory) 등을 포함한다.
④ 행위 패턴은 가교(bridge), 적응자(adapter), 복합체(composite) 등을 포함한다.

♤ **디자인 패턴**

- 행위 패턴은 가교(bridge), 적응자(adapter), 복합체(composite) 등을 포함한다.**(×)**

- 생성 패턴 : 원형, 싱글턴, 빌더, 추상 팩토리, 팩토리 메서드 패턴

정답 : ④

3. 다음 디자인 패턴들을 GoF(Gang of Fours) 패턴 분류에 따라 구분하였을 때, 생성 패턴, 구조 패턴, 행위 패턴의 개수를 순서대로 바르게 나열한 것은? [2022년 국가 7급]

○ Abstract Factory	○ Adapter	○ Bridge
○ Builder	○ Composite	○ Decorator
○ Iterator	○ Mediator	○ Observer
○ Prototype	○ Proxy	○ Singleton
○ Strategy	○ Visitor	

	생성 패턴	구조 패턴	행위 패턴
①	3	5	6
②	4	5	5
③	5	3	6
④	6	4	4

♤ 디자인 패턴 – 용도에 따른 분류

유 형	특징 및 종류	
생성 패턴 (creational pattern)	//객체 생성 방식을 결정하는 패턴 – 5개	
	• 원형 패턴(prototype) • 싱글턴 패턴(singleton)	• 빌더 패턴(builder) • 팩토리 메서드 패턴(factory method) • 추상 팩토리 패턴(abstract factory)
구조 패턴 (structural pattern)	//객체를 조직화하는데 유용한 패턴(합성에 관여) – 7개	
	• 브리지 패턴(bridge) • 적응자 패턴(adaptor) • 복합체 패턴(composite) • 데코레이터 패턴(decorator)	• 프록시 패턴(proxy) • 퍼사드 패턴(facade) • 플라이웨이트 패턴(flyweight)
행위 패턴 (behavioral pattern)	//객체들의 상호작용을 조정 관리하는 패턴 – 11개	
	• 반복자 패턴(iterator) • 옵저버 패턴(observer) • 중재자 패턴(mediator) • 메멘토 패턴(memento) • 명령어 패턴(command) • 해석자 패턴(interpreter)	• 상태 패턴(state) • 전략 패턴(strategy) • 방문자 패턴(visitor) • 템플릿 메서드 패턴(template method) • 책임연쇄 패턴(chain of responsibility)

• 디자인 패턴 전체 종류 = 5 + 7 + 11 = **23**(개)
• 디자인 패턴은 설계 문제에 대한 해답을 문서화하기 위해 고안된 방법이다.
• 디자인 패턴은 **프로그램 개발에서 자주 나타나는 과제를 해결**하기 위한 방법 중 하나이다.

정답 : ③

4. 〈보기 1〉의 디자인 패턴 분류와 〈보기 2〉의 디자인 패턴을 바르게 연결한 것은? [2018년 국가 7급]

〈보기 1〉	ㄱ. 생성 패턴	ㄴ. 구조 패턴	ㄷ. 행위 패턴
〈보기 2〉	A. Bridge 패턴	B. Singleton 패턴	C. Interpreter 패턴

	ㄱ	ㄴ	ㄷ
①	A	B	C
②	B	A	C
③	B	C	A
④	C	A	B

♤ **디자인 패턴**

A. Bridge 패턴 → 구조 패턴
B. Singleton 패턴 → 생성 패턴
C. Interpreter 패턴 → 행위 패턴

정답 : ②

5. 다음 중 성격이 다른 설계 패턴은? [2011년 국가 7급]

① 브리지(bridge) 패턴
② 팩토리 메서드(factory method) 패턴
③ 프로토타입(prototype) 패턴
④ 싱글톤(singleton) 패턴

♤ **설계 패턴(design pattern)**

---

① 브리지(bridge) 패턴 → 구조 패턴
② 팩토리 메서드(factory method) 패턴 → 생성 패턴
③ 프로토타입(prototype) 패턴 → 생성 패턴
④ 싱글톤(singleton) 패턴 → 생성 패턴

정답 : ①