5주 3강

디자인 패턴

이번 주차에는…

상위 설계 : 디자인 패턴

- 디자인 패턴

1. 다지인 패턴

- 디자인 패턴
  - 자주 사용하는 설계 형태를 정형화해서 이를 유형별로 설계 템플릿을 만들어둔 것
  - 많은 개발자들이 경험상 체득한 설계 지식을 검증하고 이를 추상화하여 일반화한 템플릿

그림 5-28 디자인 패턴의 예

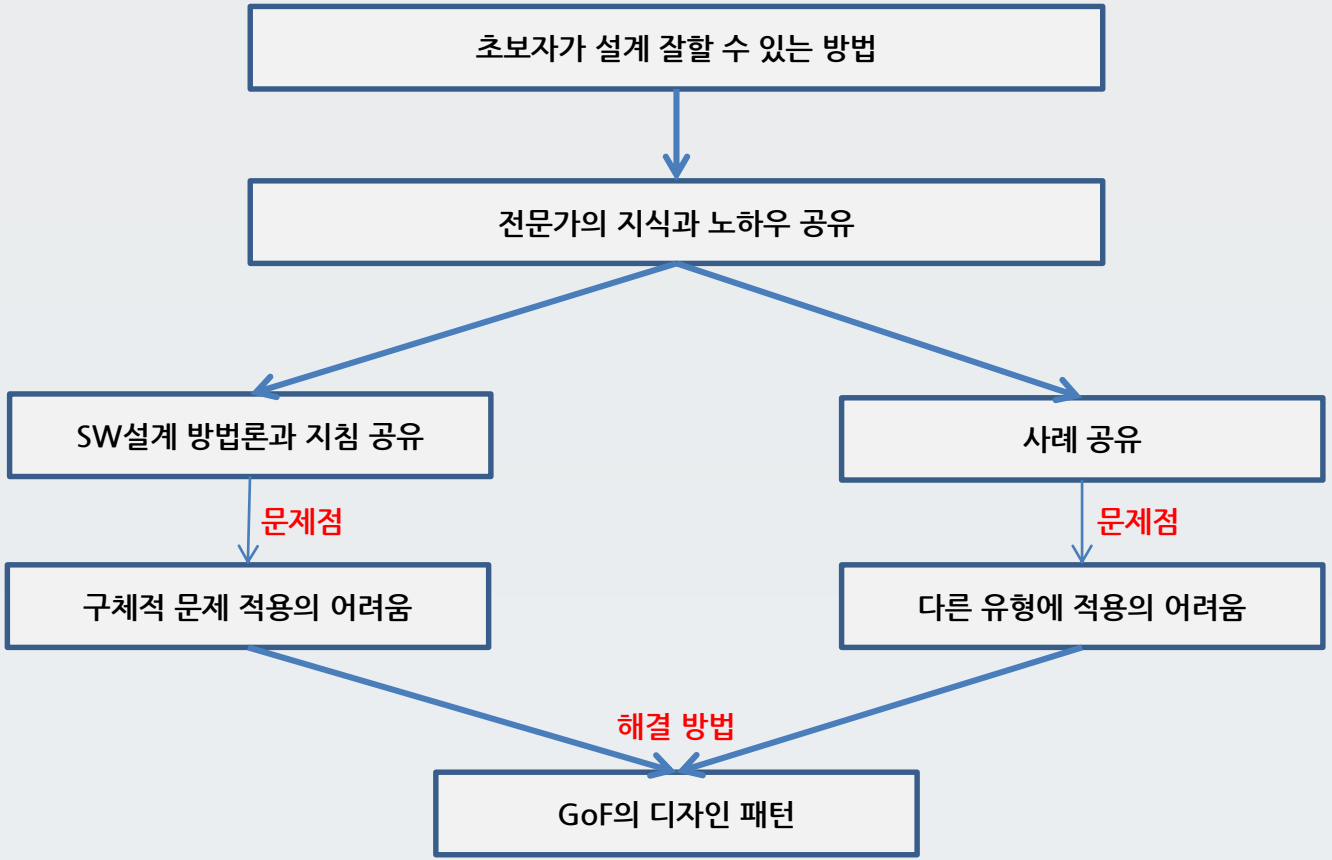
2. 다지인 패턴 사용의 장/단점

- 장점
  - 개발자(설계자) 간의 원활한 의사소통
  - 소프트웨어 구조 파악 용이
  - 재사용을 통한 개발 시간 단축
  - 설계 변경 요청에 대한 유연한 대처

- 단점
  - 객체지향 설계/구현 위주
  - 초기 투자 비용 부담

3. Gof 디자인 패턴

- 내용



4. 디자인 패턴

- 여러 가지 문제에 대한 설계 사례를 분석하여 서로 비슷한 문제를 해결하기 위한 설계들을 분류하고, 각 문제 유형별로 가장 적합한 설계를 일반화해 패턴으로 정립한 것을 의미

- 소프트웨어 설계에 대한 지식이나 노하우가 문제 유형별로 잘 구체화되어 있을 뿐 아니라, 동일한 문제 유형에 대해서는 그 해결 방법에 대한 지식이나 노하우가 패턴 형태로 충분히 일반화된 것

5. factory method 패턴

- Factory: 공장, 물건을 만드는 곳(물건-인스턴스)
- 상위 클래스에서 객체를 생성하는 인터페이스를 정의하고, 하위 클래스에서 인스턴스를 생성하도록 하는 방식
- 객체를 생성하는 시점은 알지만 어떤 객체를 생성해야 할지 알 수 없을 때, 객체 생성을 하위 클래스에 위임하여 해결

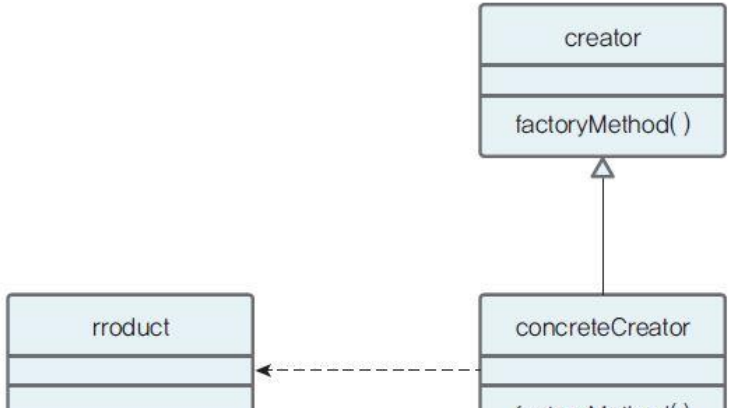


그림 5-29 factory method 패턴

6. Singleton 패턴

- Singleton: '단독 개체', '독신자'라는 뜻 말고도 '정확히 하나의 요소만 갖는 집합'
- 특정 클래스의 객체가 오직 한 개만 존재하도록 보장, 즉 클래스의 객체를 하나로 제한
- 동일한 자원이나 데이터를 처리하는 객체가 불필요하게 여러 개 만들어질 필요가 없는 경우에 주로 사용

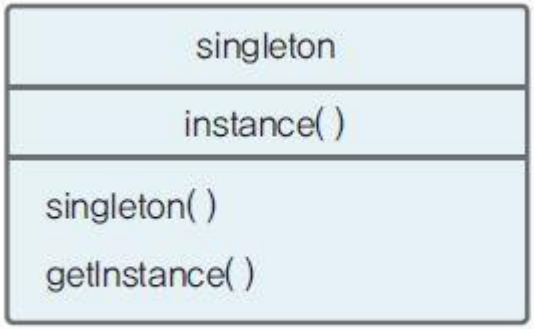


그림 5-30 singleton 패턴

7. prototype 패턴

- new Object( )보다 clone( )을 이용해 기존의 것을 복사하여 일부만 바꿔 인스턴스 생성
- 일반적인 prototype(원형)을 만들어놓고, 그것을 복사한 후 필요한 부분만 수정하여 사용
- 인스턴스를 복제하여 사용하는 구조
- 생성할 객체의 원형을 제공하는 프로토타입 인스턴스로부터 생성할 객체들의 타입 결정
- 객체를 생성할 때 갖추어야 할 기본 형태가 있을 때 사용

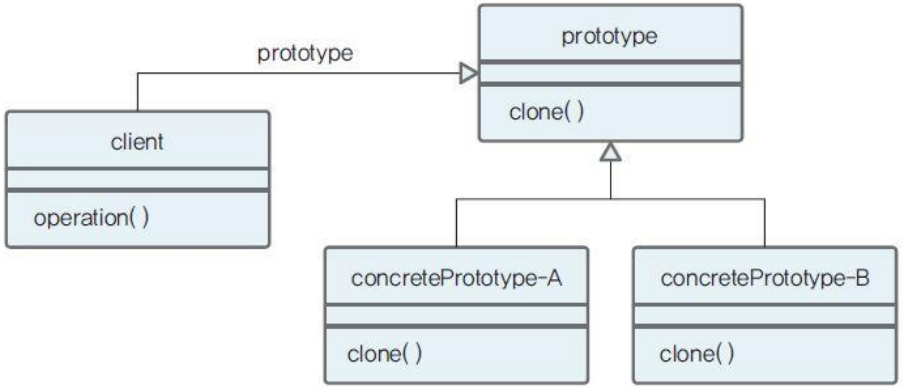


그림 5-31 prototype 패턴

8. Builder 패턴

- 복잡한 인스턴스를 조립하여 만드는 구조
- 복합 객체를 생성할 때 객체를 생성하는 방법(과정)과 객체를 구현(표현)하는 방법을 분리
- 동일한 생성 절차에서 서로 다른 표현 결과를 만들 수 있다.

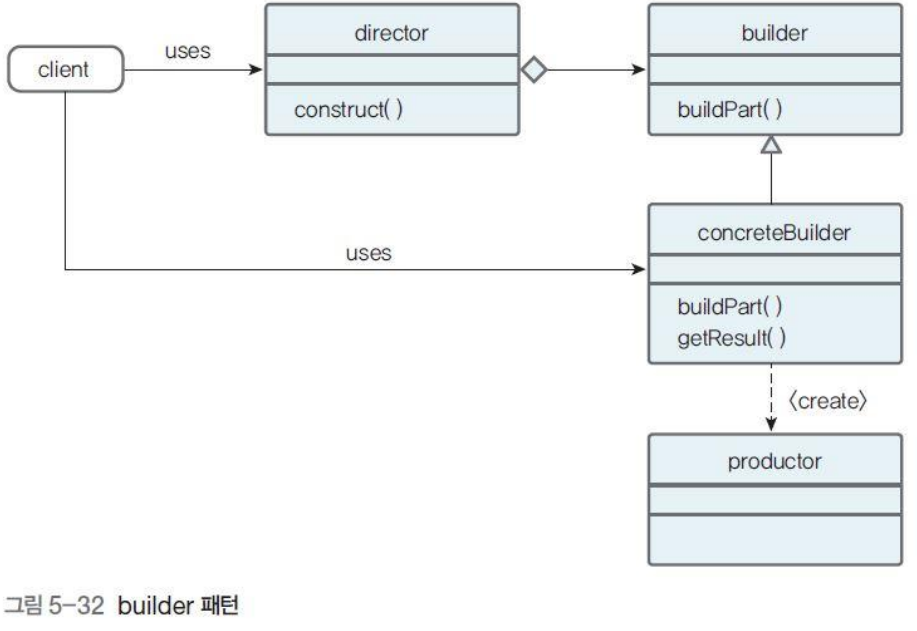


그림 5-32 builder 패턴

9. abstract factory 패턴

- abstract factory: '추상적인 공장',
- [그림 5-33]과 같이 여러 개의 concrete Product를 추상화시킨 것
- 구체적인 구현은 concreteProduct 클래스에서 이루어짐
- 사용자에게 API를 제공하고, 인터페이스만 사용해서 부품을 조립하여 만든다.
  즉 추상적인 부품을 조합해서 추상적인 제품을 만든다.

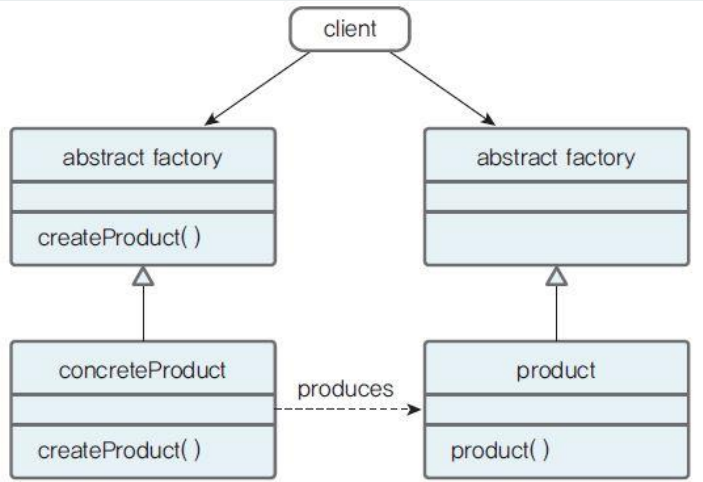


그림 5-33 abstract factory 패턴

10. Composite 패턴

- Composite: '합성의', '합성물', '혼합 양식'
- composite object: 부분-전체의 상속 구조로 표현되는 조립 객체
- 사용자가 단일 객체와 복합 객체 모두 동일하게 다루도록 한 것
- 재귀적 구조: 디렉토리 안에 파일 또는 다른 디렉토리(서브 디렉토리)가 존재할 수 있는 것
- 그릇(디렉토리)과 내용물(파일)을 동일시해서 재귀적인 구조를 만들기 위한 설계 패턴

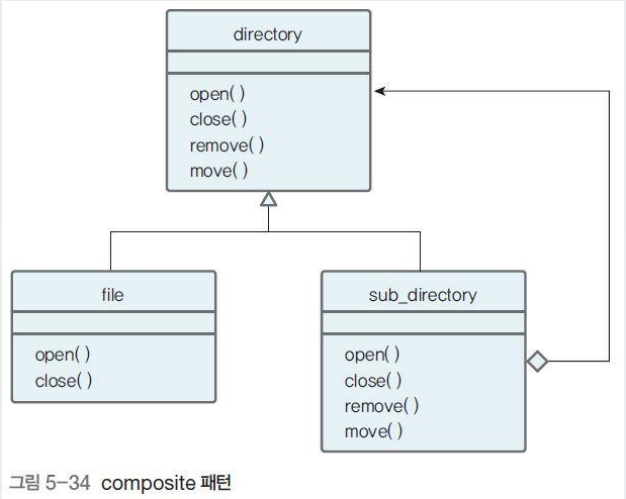


그림 5-34 composite 패턴

11. adapter 패턴(1)

- adapter
  - '접속 소켓', '확장 카드', '(물건을 다른 것에) 맞추어 붙이다', '맞추다'

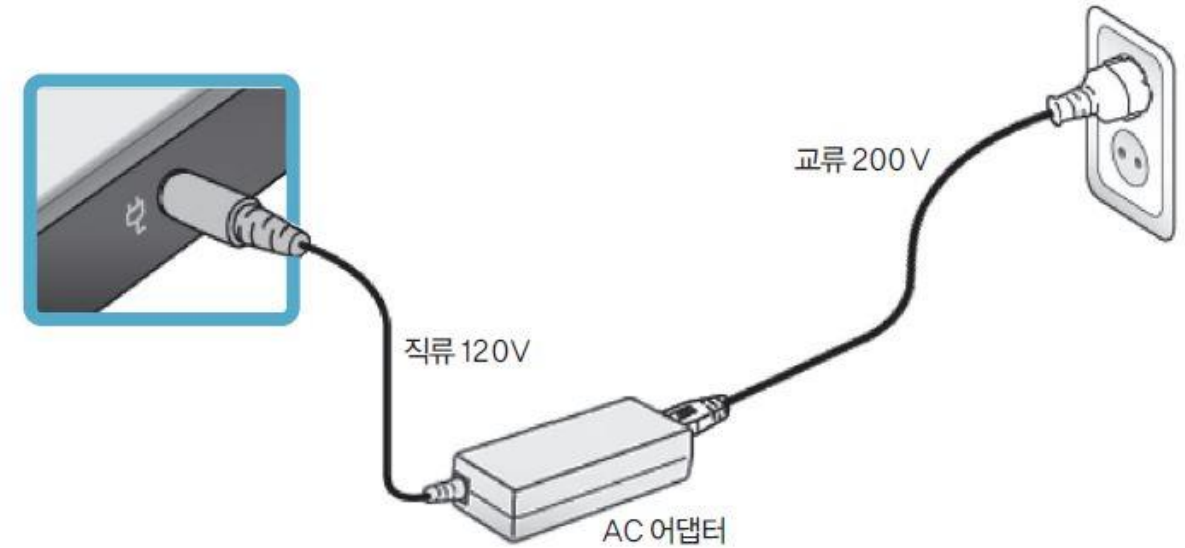


그림 5-35 어댑터의 역할 예

12. adapter 패턴(2)

- adapter 패턴
  - 기존 클래스를 재사용할 수 있도록 중간에서 맞춰주는 역할
  - 호환성이 없는 기존 클래스의 인터페이스를 변환해 재사용할 수 있도록 해준다.
    - ✓ 클래스 adapter 패턴 : 상속을 이용한 어댑터 패턴
    - ✓ 인스턴스 adapter 패턴 : 위임을 이용한 어댑터 패턴

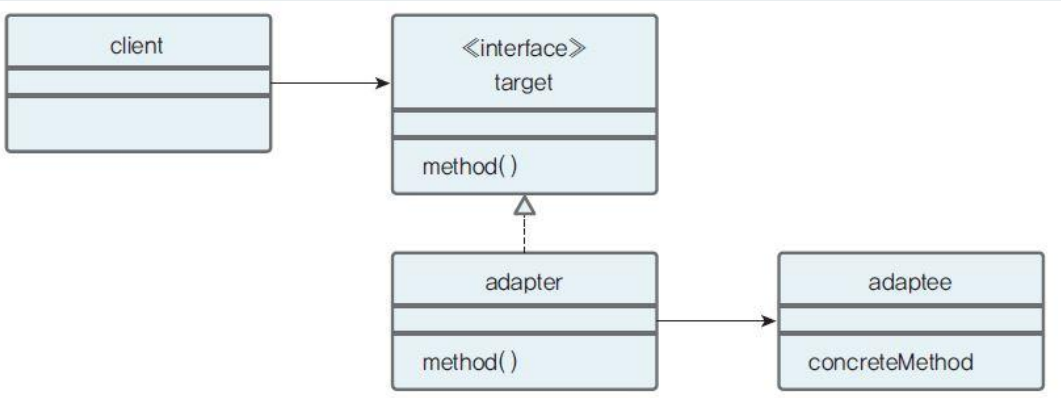


그림 5-36 adapter 패턴

13. bridge 패턴

- bridge: '무엇인가를 연결한다'
- 두 장소를 연결하는 역할
- 기능의 클래스 계층과 구현의 클래스 계층을 연결하고, 구현부에서 추상계층을 분리하여 각자 독립적으로 변형할 수 있게 해준다.
- 구현과 인터페이스(추상화된 부분)를 분리할 수 있고, 추상화된 부분과 실제 구현 부분을 독립적으로 확장할 수 있다.

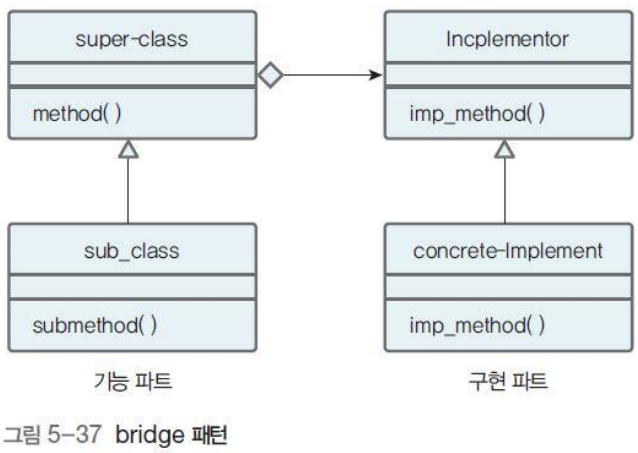


그림 5-37 bridge 패턴

14. decorator 패턴

- Decoration: '장식(포장)'
- 기존에 구현되어 있는 클래스(둥근 모양의 빵)에 그때그때 필요한 기능(초콜릿, 치즈,
- 생크림)을 추가(장식, 포장)해나가는 설계 패턴
- 기능확장이 필요할 때 상속의 대안으로 사용

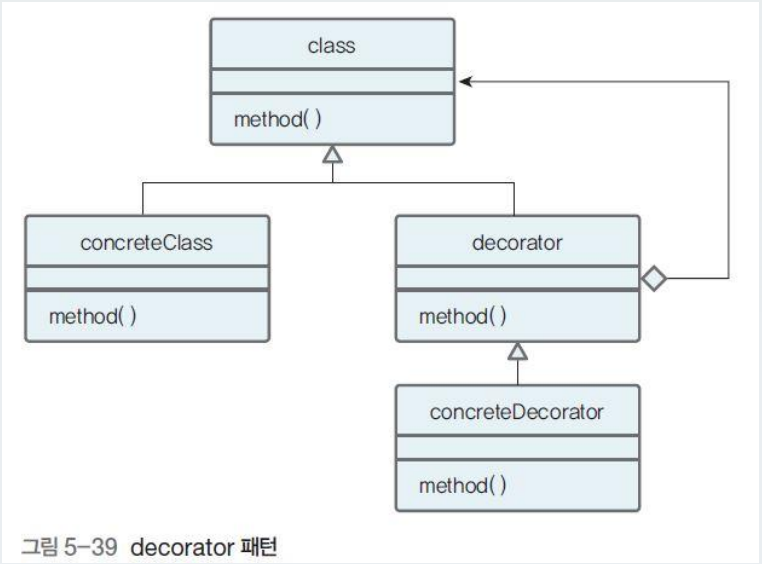


그림 5-39 decorator 패턴


그림 5-38 케이크 장식의 예

15. façade 패턴

- Façade: '건물의 앞쪽 정면(전면)'
- 몇 개의 클라이언트 클래스와 서브시스템의 클라이언트 사이에 facade라는 객체를 세워놓음으로써 복잡한 관계를 정리(구조화)한 것
- 모든 관계가 전면에 세워진 facade 객체를 통해서만 이루어질 수 있게 단순한 인터페이스를 제공(단순한 창구 역할)하는 것

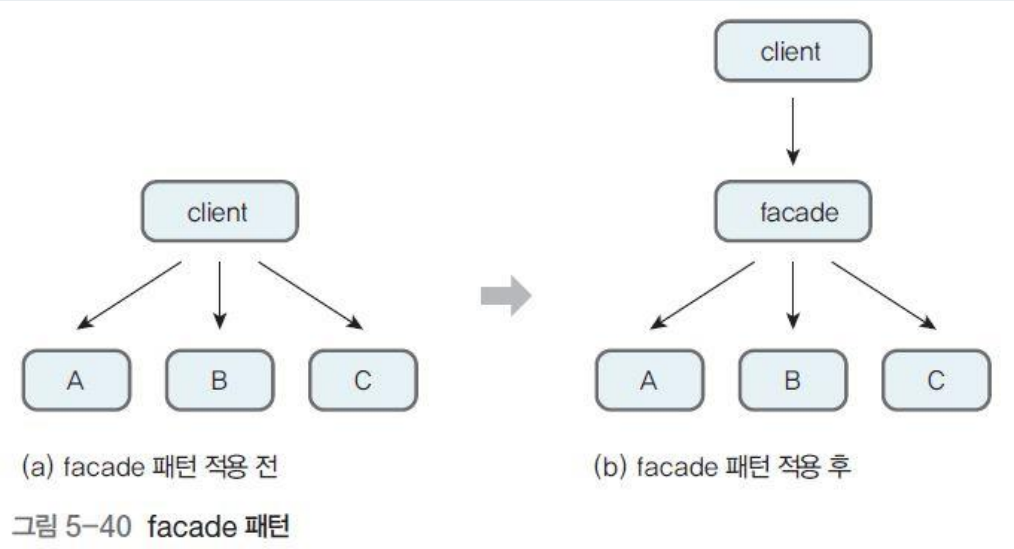


그림 5-40 facade 패턴

16. Flyweight 패턴

- Flyweight: '(권투·레슬링 등의) 플라이급 선수(보통 체중 48~51kg 사이)', 즉 가벼운 것
- 메모리를 가볍게 해준다고 짐작할 수 있다.
- 메모리 사용량을 줄이기 위한 방법으로, 인스턴스를 필요한 대로 다 만들어 쓰지 말고, 동일한 것은 가능하면 공유해서 객체 생성을 줄이자는 것

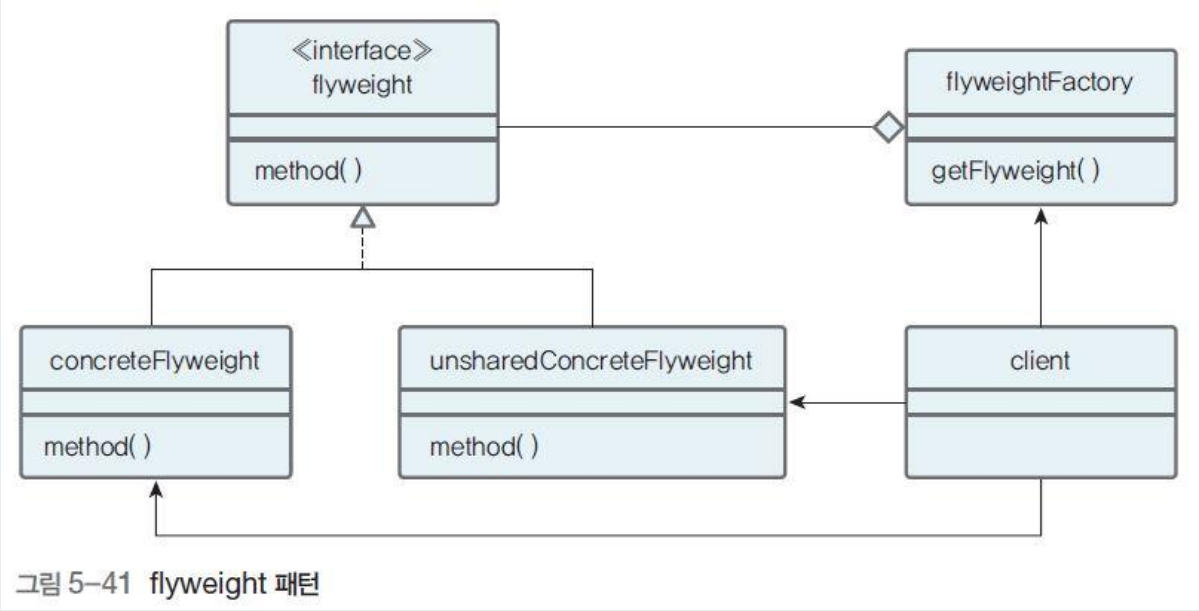


그림 5-41 flyweight 패턴

17. Proxy 패턴

- Proxy: '대리인', 즉 뭔가를 대신해서 처리하는 것
- 그림과 텍스트가 섞여있는 경우 텍스트가 먼저 나오고 나중에 그림이 나올 수 있도록 하기 위해 텍스트 처리용 프로세스, 그림 처리용 프로세스를 별도로 두고 운영하는 것 같은 설계

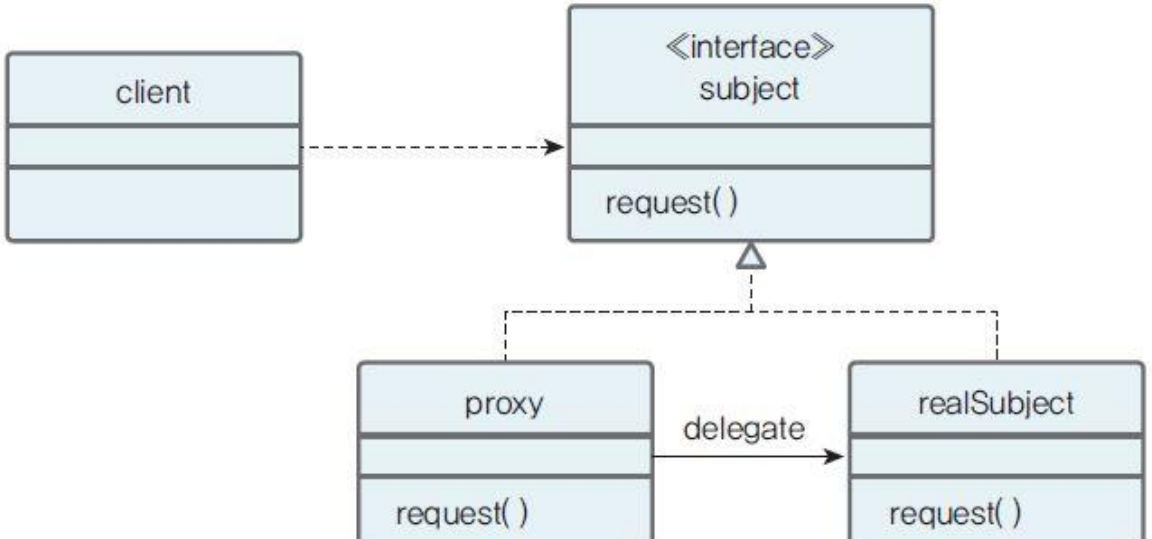


그림 5-42 proxy 패턴

18. iterator 패턴(1)

- Iterate: '반복하다', iterator: '반복자'
- 반복이 필요한 자료구조를 모두 동일한 인터페이스를 통해 접근할 수 있도록 아래 그림 처럼 iterator 객체 속에 넣은 다음, iterator 객체의 메서드를 이용해 자료구조를 활용할 수 있도록 해준다.

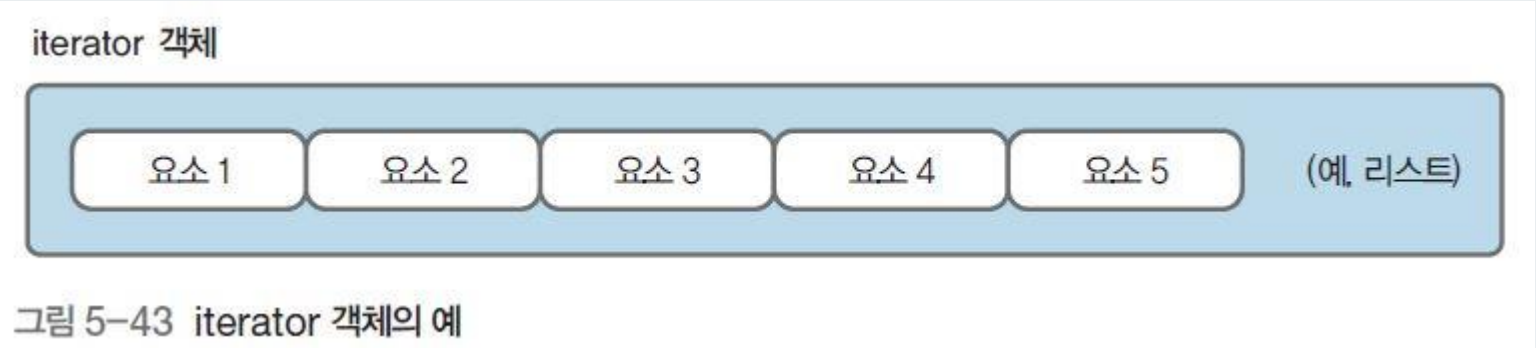


그림 5-43 iterator 객체의 예

- 데이터들의 집합체를 모두 동일한 인터페이스를 사용하여 조작함으로써 데이터들의 집합체를 쉽게 사용할 수 있게 해준다.

19. iterator 패턴(2)

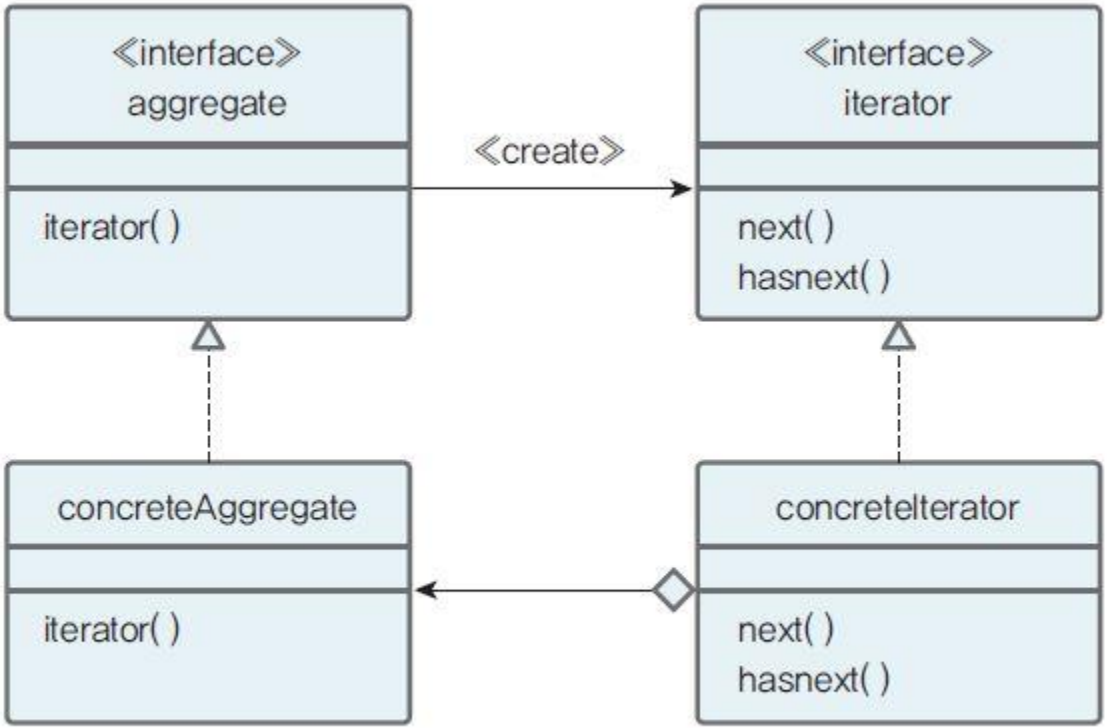


그림 5-44 iterator 패턴

20. Observer 패턴(1)

- Observer: '관찰하는 사람', '관찰자'

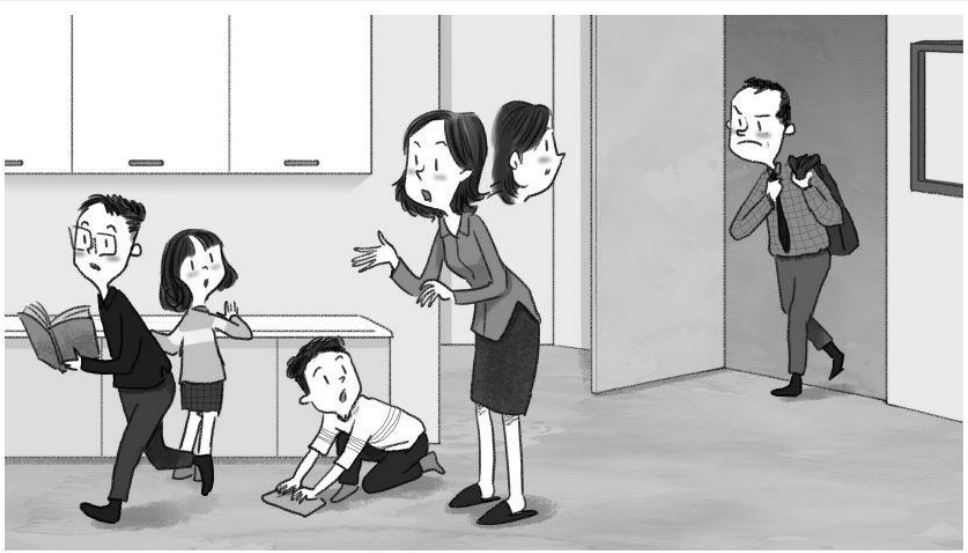


그림 5-45 observer 패턴의 비유

- 위 그림의 예처럼 어떤 클래스에 변화가 일어났을 때, 이를 감지하여 다른 클래스에 통보해주는 것

21. Observer 패턴(2)

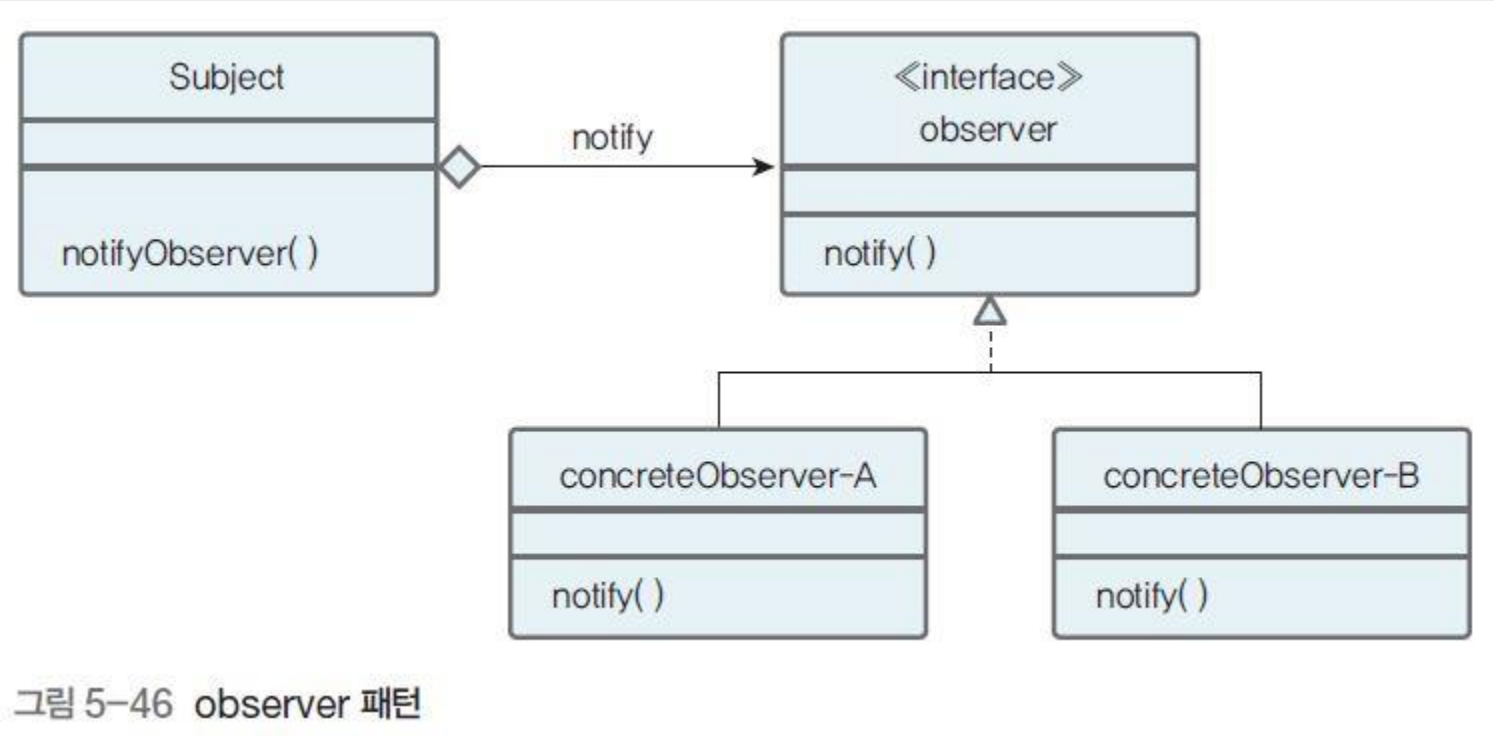


그림 5-46 observer 패턴

22. strategy 패턴(1)

- Strategy: '전략', '전술'
- 소프트웨어 개발에서 전략이나 전술은 알고리즘으로 구현

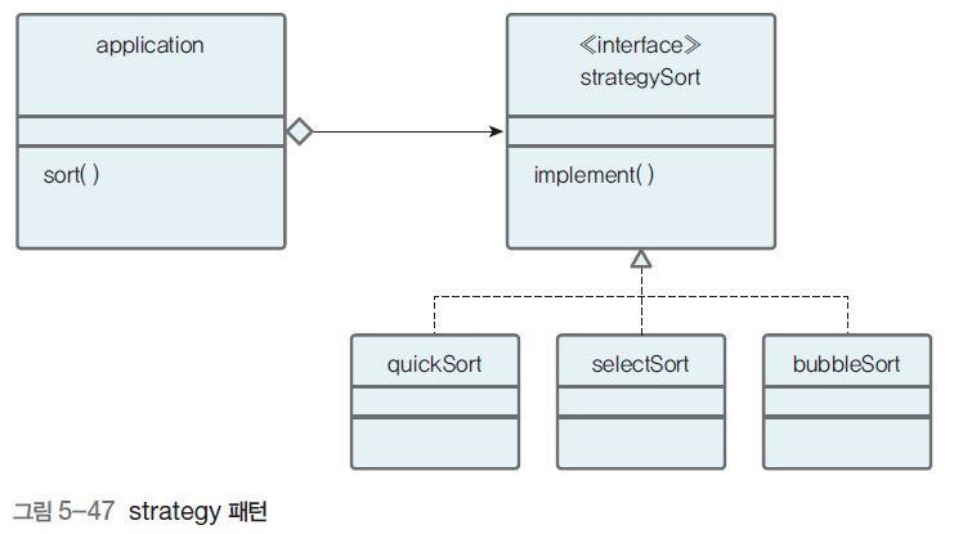


그림 5-47 strategy 패턴

- 위 그림처럼 알고리즘 군을 정의하고(strategySort 추상클래스) 같은 알고리즘(버블 정렬, 퀵 정렬, 선택 정렬 등)을 각각 하나의 클래스로 캡슐화한(quickSort 클래스, selectSort 클래스, bubbleSort 클래스) 다음, 필요할 때 서로 교환해서사용할 수 있게 해준다.

23. strategy 패턴(2)

- Strategy 패턴
  - 클라이언트와 무관하게 독립적으로 알고리즘 변경 가능(quickSort →bubbleSort), 클라이언트는 독립적으로 원하는 알고리즘 사용 가능
  - 클라이언트에게 알고리즘이 사용하는 데이터나 그 구조를 숨겨주는 역할
  - 알고리즘을 사용하는 곳과, 알고리즘을 제공하는 곳을 분리시킨 구조로알고리즘을 동적으로 교체 가능

24. template method 패턴

- Template: 하나의 '틀'
- 이런 틀 기능을 구현할 때 template method 패턴을 이용
- 상위 클래스에서는 추상적으로 표현하고 그 구체적인 내용은 하위 클래스에서 결정되는 디자인 패턴

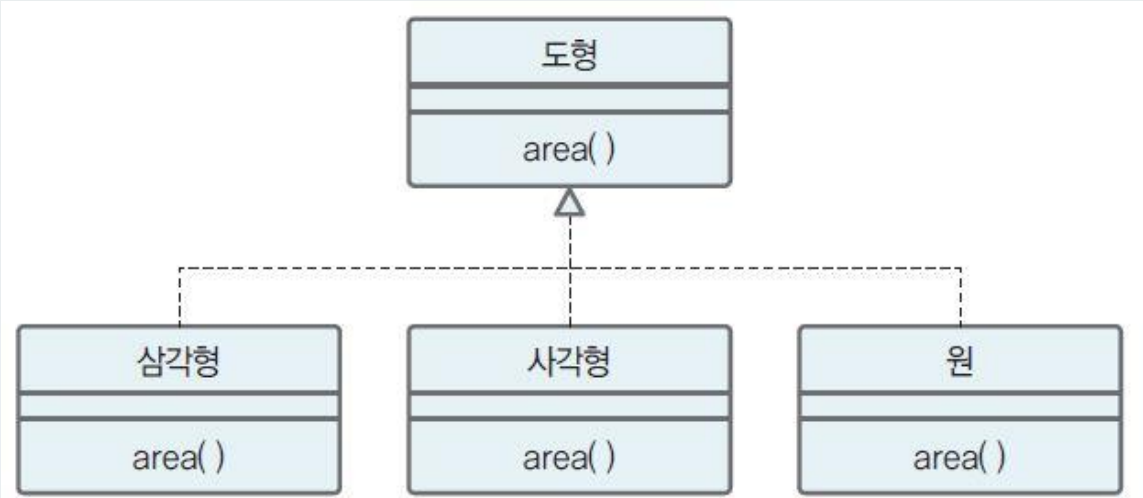


그림 5-48 template method 패턴

25. Visitor 패턴(1)

- Visitor: '방문자

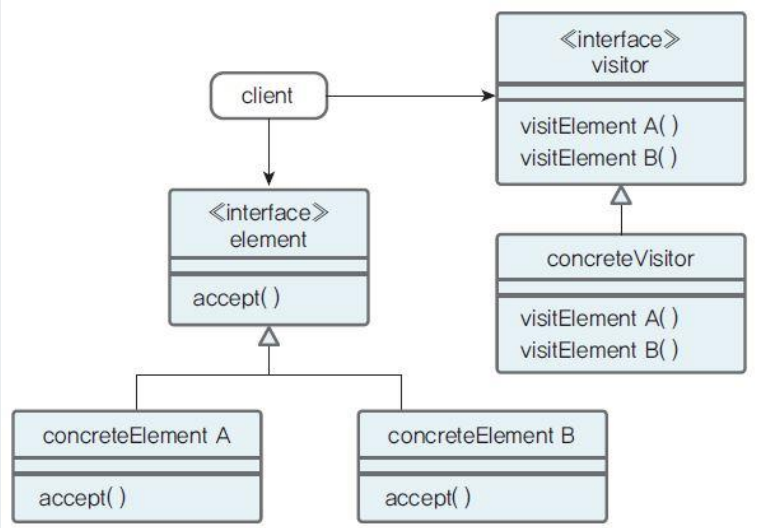


그림 5-49 visitor 패턴

- 위 그림처럼 각 클래스의 데이터 구조로부터 처리 기능을 분리하여 별도의 visitor 클래스로 만들어놓고 해당 클래스의 메서드(visitElement A, visitElement B)가 각 클래스를 돌아다니며 특정 작업을 수행하도록 하는 것

26. Visitor 패턴(1)

- Visitor 패턴
  - 객체의 구조와 기능을 분리
  - 객체의 구조는 변경하지 않으면서 기능만 따로 추가하거나 확장할 때 많이 사용
  - 장점: 클래스의 데이터 구조 변경 없이 기존 작업(기능) 외에 다른 작업을 추가하기가 수월

27. chine of responsibility 패턴(1)

- chine of responsibility
- 책임들이 연결되어 있어 내가 책임을 못 질 것 같으면 다음 책임자에게 자동으로 넘어가는 구조

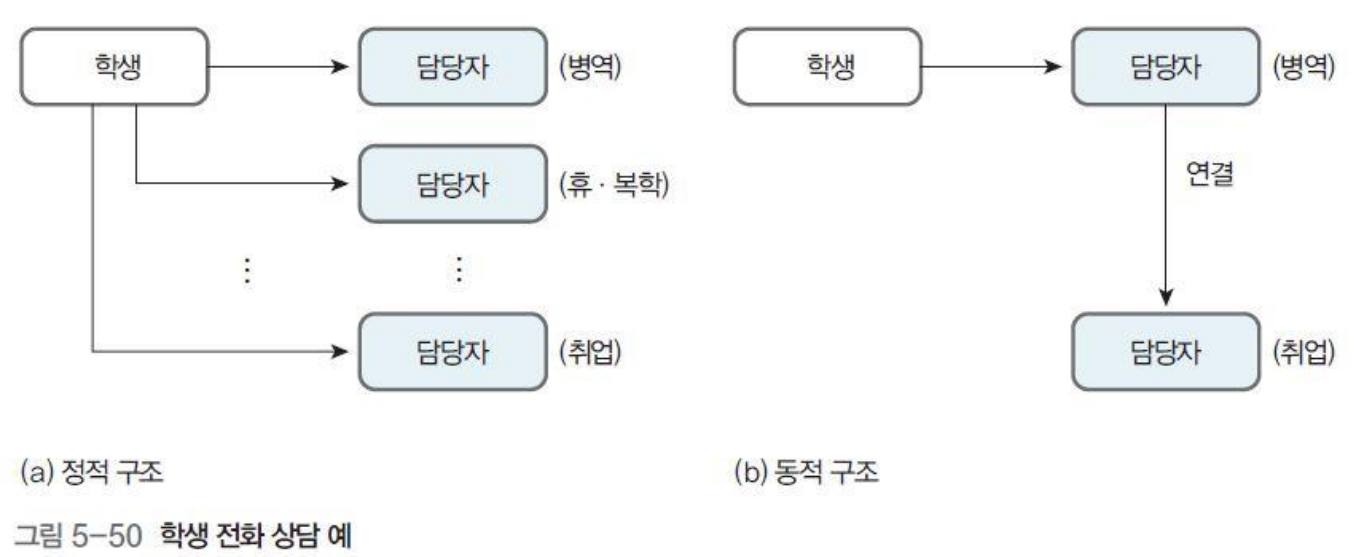


그림 5-50 학생 전화 상담 예

- 소프트웨어 개발에서도 이렇게 자동으로 연결되는 구조로 프로그램을 만들면 매우 유용한데 이 개념을 적용할 수 있는 것이 바로 chine of responsibility 패턴

28. chine of responsibility 패턴(2)

- chine of responsibility 패턴

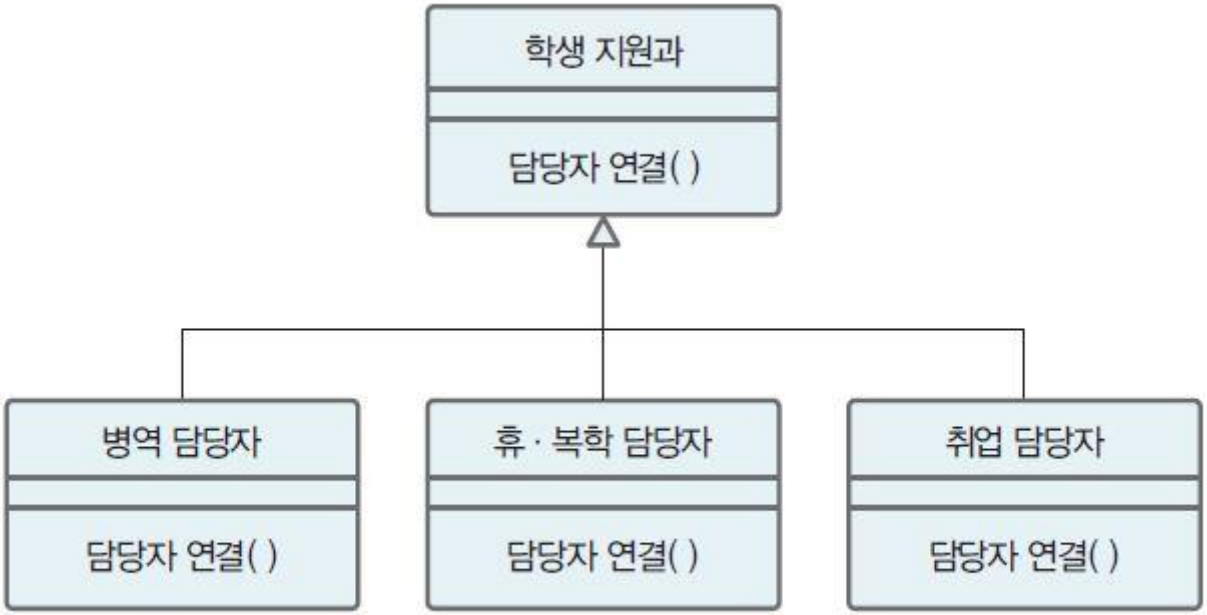


그림 5-51 chine of responsibility 패턴

29. command 패턴(1)

- Command: '명령어'

  (예) 문서편집기의 복사(copy), 붙여넣기(paste), 잘라내기(cut) 등

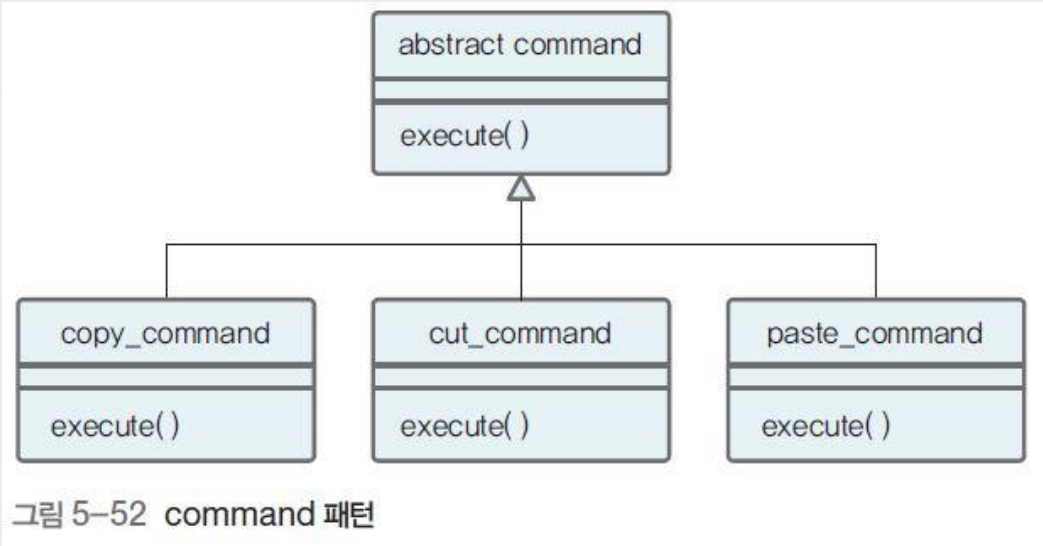


그림 5-52 command 패턴

- 위의 명령어를 각각 구현하는 것보다는 위 그림처럼 하나의 추상 클래스에 execute( ) 메서드를 하나 만들고 각 명령이 들어오면 그에 맞는 서브 클래스(copy_command)가 선택되어 실행하는 것이 효율적

30. command 패턴(2)

- Command 패턴
  - ✓ 단순히 명령어를 추상 클래스와 구체 클래스로 분리하여 단순화한 것으로 끝나지 않고, 명령어에 따른 취소(undo) 기능까지 포함
    - ❖ 이유: 사용자 입장에서는 해당 명령어를 실행했다가 취소(undo)하기도 하기 때문
  - ✓ 이렇게 프로그램의 명령어를 구현할 때 command 패턴을 활용

31. mediator 패턴(1)

- Mediator: '중재자', '조정자', '중개인'
- 부동산 중개사, 비행기의 이착륙을 통제하는 관제탑, 중고 물건을 사고파는 사이트처럼 중간에서 연결하고 통제하는 역할

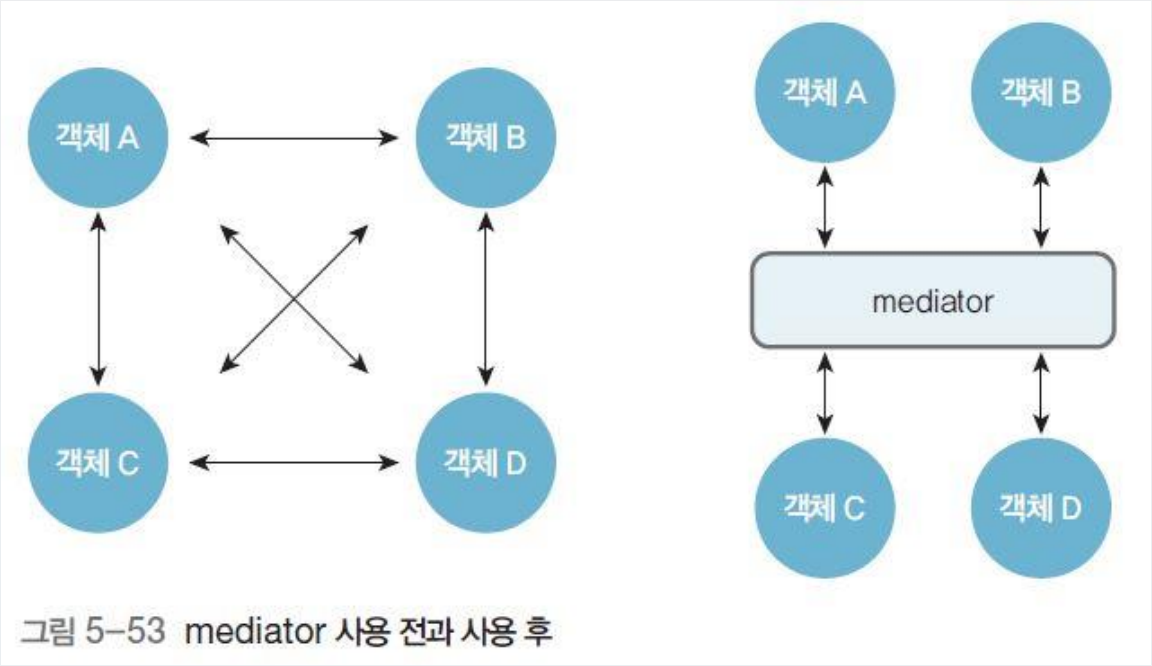


그림 5-53 mediator 사용 전과 사용 후

32. mediator 패턴(2)

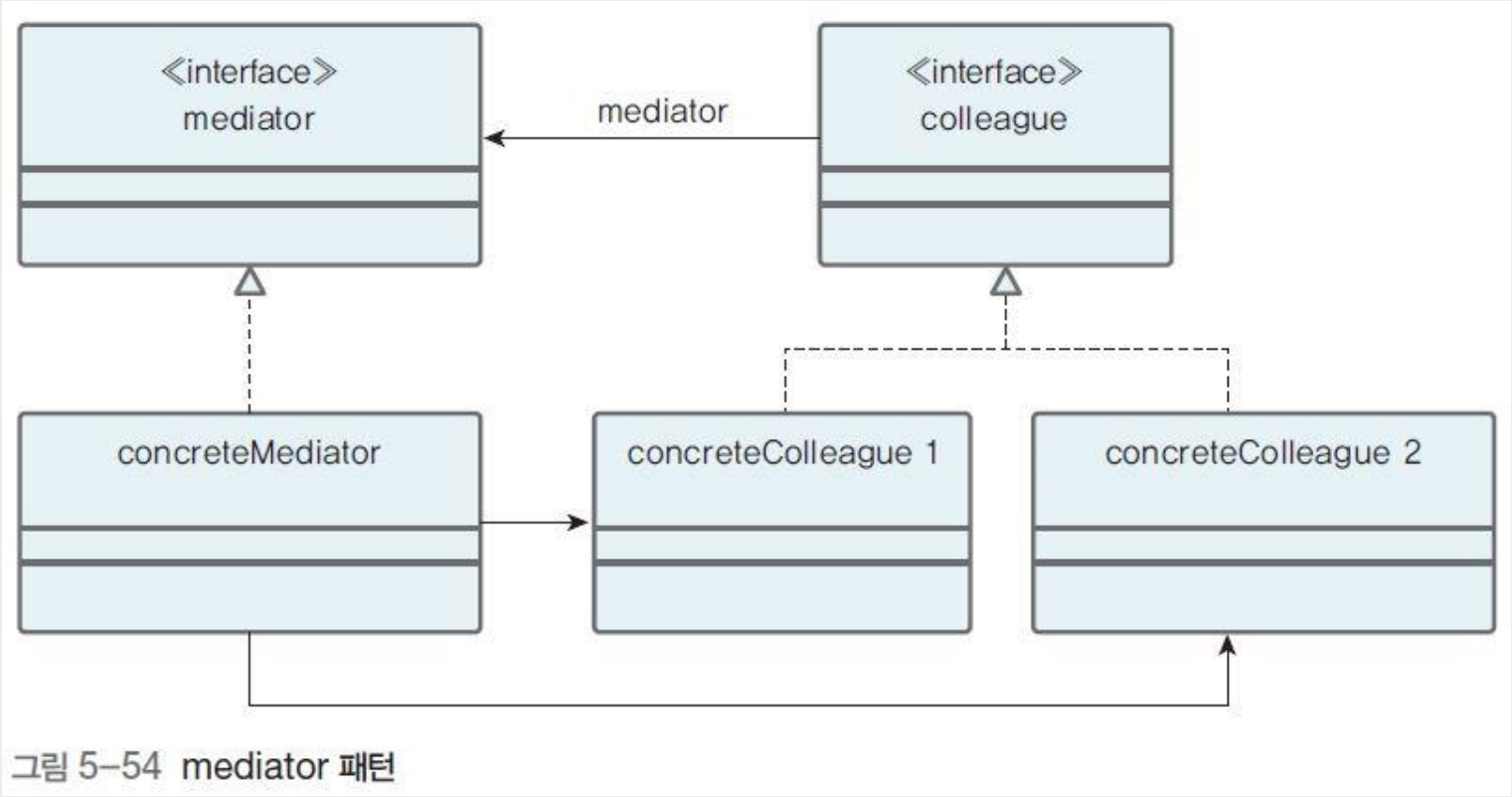


그림 5-54 mediator 패턴

33. state 패턴

- State: '상태'
- 동일한 동작을 객체 상태에 따라 다르게 처리해야 할 때 사용
- 아래 그림처럼 객체 상태를 캡슐화하여 클래스화함으로써 그것을 참조하게 하는 방식으로 상태에 따라 다르게 처리(upState, stopState, downState)할 수 있도록 한 것
- 변경 시(신규 상태 추가) 원시 코드의 수정을 최소화할 수 있고, 유지보수를 쉽게 할 수 있다.

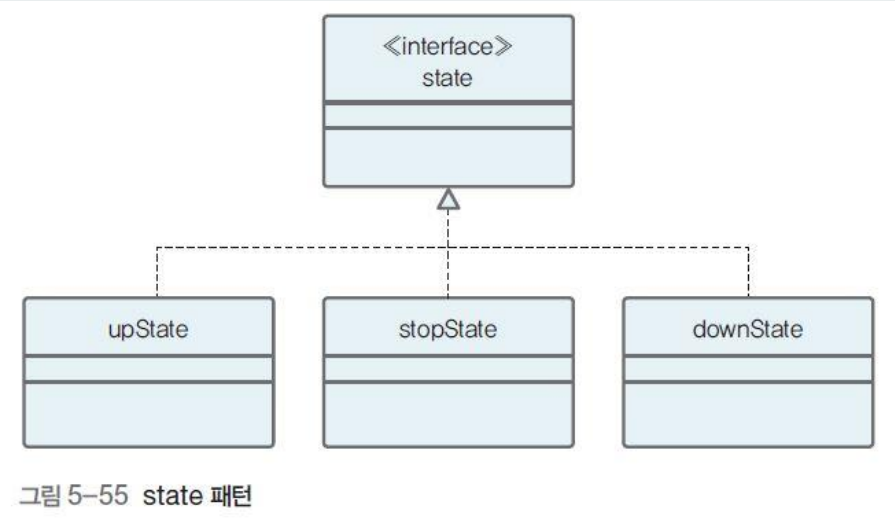


그림 5-55 state 패턴

34. memento 패턴

- Memento: '(사람, 장소를 기억하기 위한) 기념품'
- undo 기능을 개발할 때 유용
- 클래스 설계 관점에서 객체의 정보를 저장할 필요가 있을 때 적용

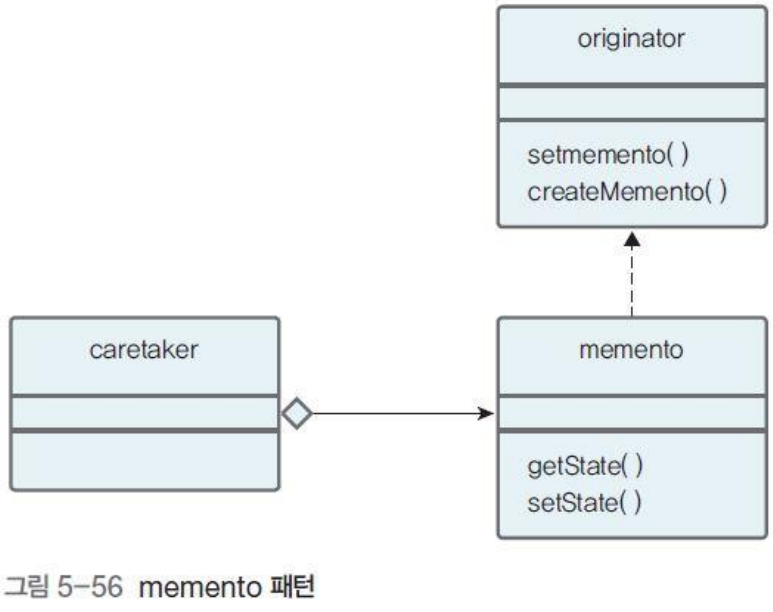


그림 5-56 memento 패턴

35. interpreter 패턴

- Interpreter: '통역자'
- 단어의 의미처럼 무언가를 번역하는 데 사용
- 간단한 언어의 문법을 정의하고 해석하는 데 사용
- 문법 규칙을 클래스화한 구조를 갖는SQL 언어나 통신 프로토콜 같은 것을 개발할 때 사용

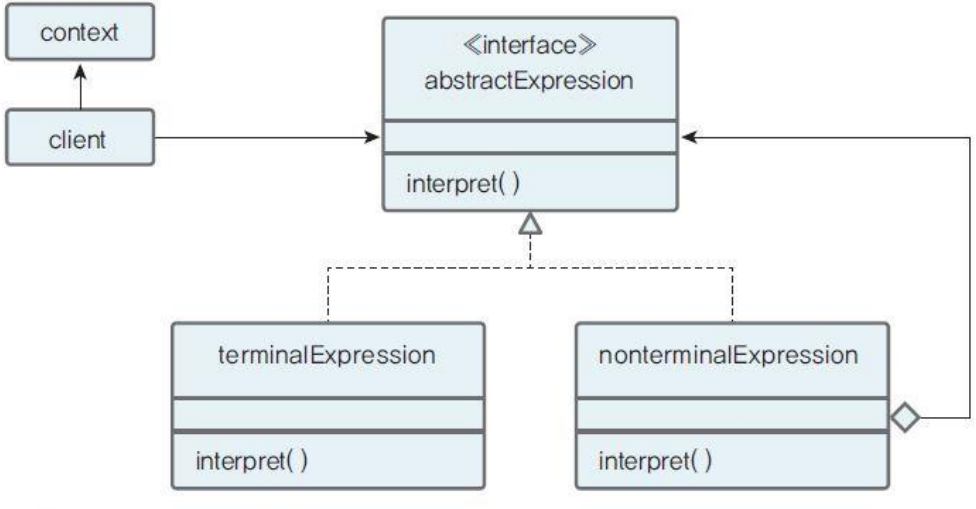


그림 5-57 interpreter 패턴

다음 시간

하위 설계