Software Engineering

6. Design Patterns
(Chapter 7)

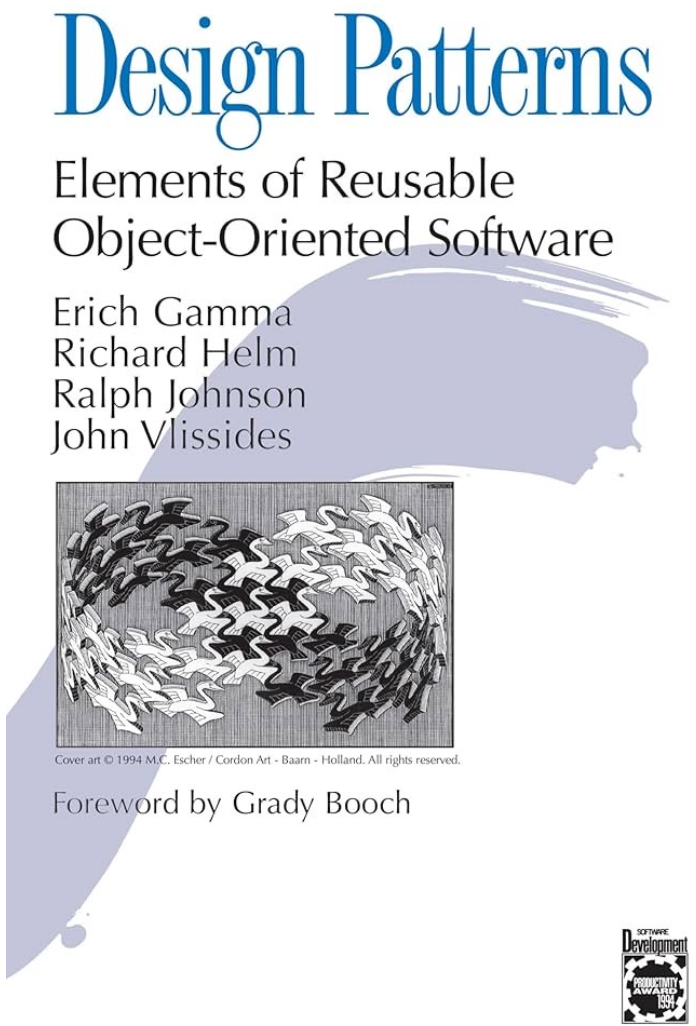
Ji Eun Kim
Fall 2024

Example: Requirements -> Architecture -> Design

- Requirements:
  - 시스템은 자료를 갱신하기 위한 컨트롤과 함께 재고 자료를 표시해야 한다”
- Architecture Level:
  - 대화형 시스템
    - 자료를 이용하기 위해 지시하고 결과를 표시
  - 적합한 architectural style: MVC
- 하위 Design Level:
  - MVC 구현을 위한 Solution은?
  - View에 최근 사항이 반영되도록 하는 방법에 Design pattern을 적용하여 효과적인 solutions을 찾음.
  - Two Options:
    1. Observer pattern에 따라 Model이 View의 상태변경을 알림
    2. Model에 대한 변경이 Controller에 의해 시작된다면 Model에 상태 변경이 있을 때 Controller가 View에 알리도록 할 수 있다.

Design Patterns

- GoF (Gang of Four) 의 Design Patterns
  - 소프트웨어 설계를 위한 지식이나 노하우를 공유할 방법 중 하나
  - 설계 중에서 재사용할 경우에 유용한 것을 디자인 패턴으로 정립
  - 동일한 유형의 문제를 해결하는 방법에 대한 지식이나 노하우가 패턴 형태로 일반화
  - 쉽게 재사용할 수 있도록 객체지향 개념에 따른 설계만을 패턴으로 지정



Design Patterns Categories

- Creational Patterns (생성 패턴)
- Structure Patterns (구조 패턴)
- Behavior Patterns (행위 패턴)

- 총 23개의 패턴

Creational Patterns (생성 패턴)

- Creational Patterns
  - 객체의 **생성**과 **참조** 과정을 **추상화**해 **특정 객체의 생성 과정을 분리**
  - 누가 언제 **변경**하더라도 전체 시스템에 미치는 영향을 최소화하도록 만들어 줌
  - 코드의 **유연성**을 높일 수 있고 **유지관리**를 쉽게 만듦
- 종류
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton

Singleton Pattern

- 객체를 강제적으로 **하나**만 생성하려는 목적
  - 예: DB 커넥션을 위한 인터페이스
- 방법
  - **클래스** 자체를 **정적** 변수로
  - 생성자는 private으로 선언
  - 유일한 객체를 접근하는 정적 메서드

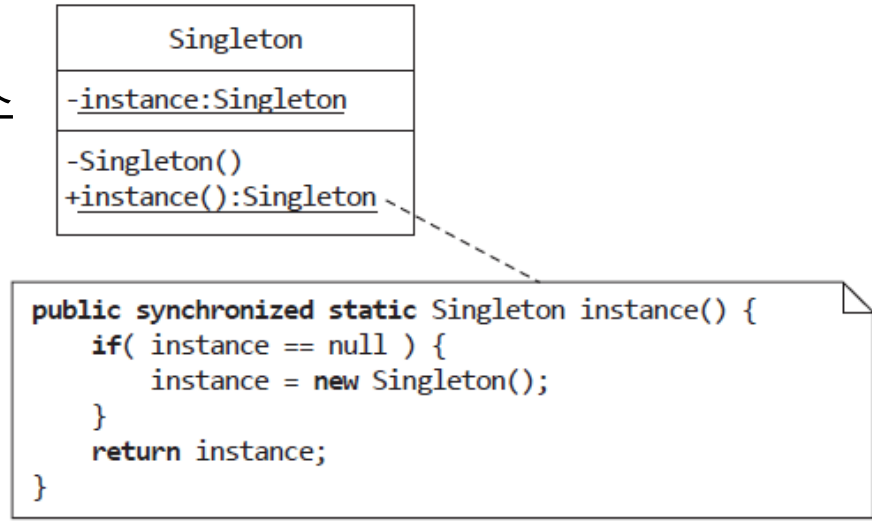


그림 7.17 싱글톤 패턴

Example

- 클라이언트 코드에서는 instance()를 호출
- DatabaseConnection dc = DatabaseConnection.instance()

```
public class DatabaseConnection{
   private DatabaseConnection() {
      // Establish a databaseconnection
      ...
   }
   private static DatabaseConnection instance = null;
   public synchronized static DatabaseConnection instance() {
      if( instance == null ) {
         instance = newDatabaseConnection( );
      }
      return instance;
   }
   // Database interface methods
   public synchronized String getAssignmentGrade(Stringstudent, String course,
String assignment) {
      ...
      return ...;
   }
}
```

Factory Method Pattern

- 클라이언트에서 사용할 **클래스의 객체를 생성하는 책임을 분리**하여 객체 생성에 변화를 대비
- 객체를 생성하기 위한 팩토리 메소드(createProduct)를 포함하는 추상 클래스(AbstractCreator)를 정의

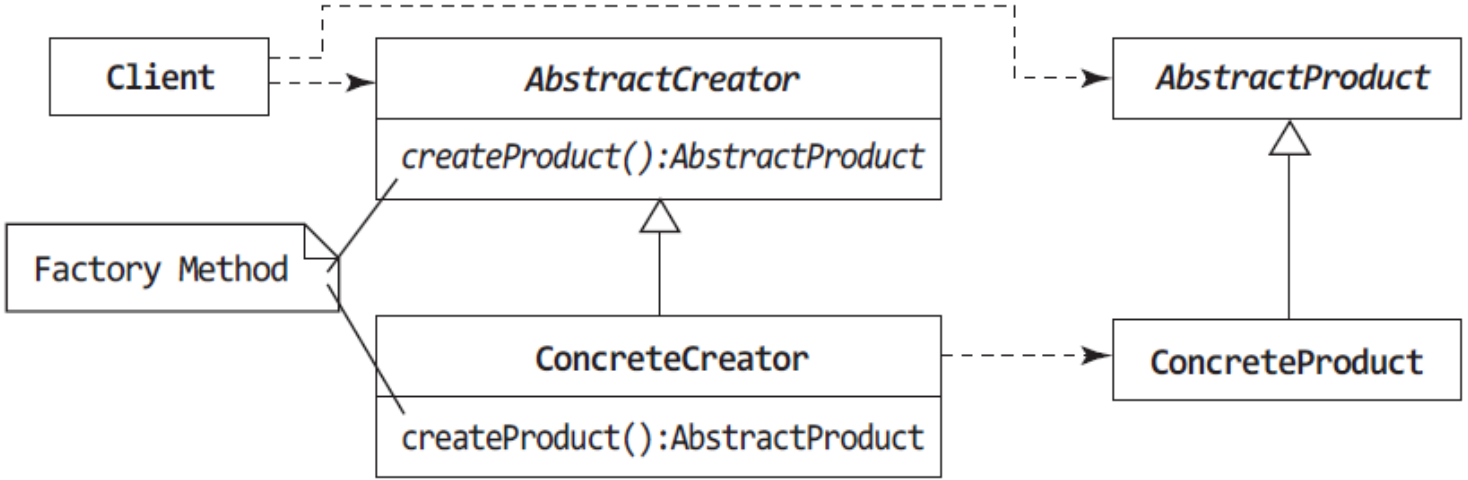
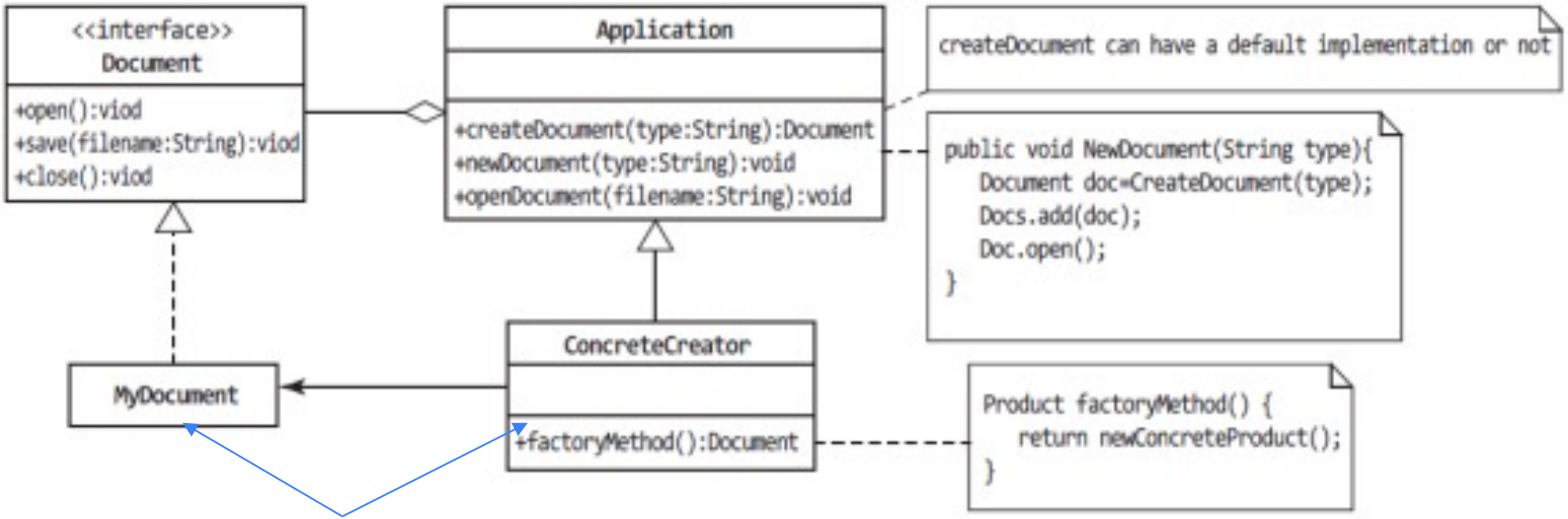


그림 7.26 팩토리 메소드 패턴

Factory Method Pattern Example

- Desktop Application Framework
  - Document 열기, 작성, 저장 등의 operation 에 대한 정의가 필요
  - 기본 Class: Application, Document 추상 클래스
  - Concreate Class: 응용 프로그램에 대한 구체적인 클래스



응용 프로그램에 해당되는 구체적인 클래스 (e.g. PDF Document, PDF Application)

Hands-on: Factory Pattern Case Study

- 게임서버
  - 게임 서버는 게임의 종류가 늘어날 때마다 추가
  - 하나의 게임을 선택하면 게임 서버는 ‘정상 연결’ 메시지와 함께 게임명, 게임 버전, ‘게임 실행 준비 완료’ 메시지를 화면에 띄움
  - 추후에는 게임 서버도 국가별로 별도로 둘 예정이며 게임도 그 나라의 특성에 맞는 게임으로 수정할 예정

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Hands-on: Factory Pattern Case Study

- 일반적인 설계

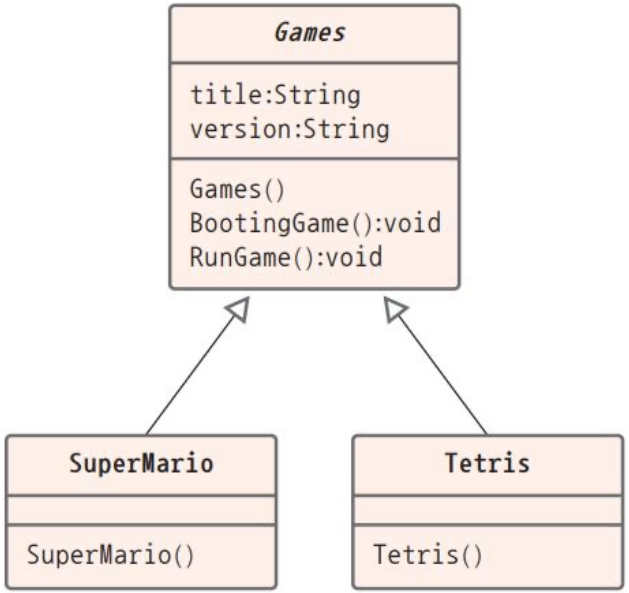


그림 7-11 게임의 상속 구조

- Factory Pattern 적용한 설계

- 게임의 종류는 Games라는 추상 클래스의 하위 클래스에 위치
- 새로운 게임 하나가 추가되면 GameServerFactory 클래스에서 객체를 생성하고 if 문에 추가
- 게임의 추가/삭제에 따른 변화가 GameServer에 영향을 미치지 않음

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Hands-on: Factory Pattern Case Study

Factory Pattern 을 적용한 Class Diagram

Abstract Factory Pattern

- 객체를 사용할 클라이언트에서 구체적인 객체 생성을 지정하는 책임을 분리하기 위하여 추상 인터페이스를 이용하여 관련 **객체 패밀리를 생성**
- 객체 패밀리: 부품 객체의 집합체 e.g.,
  - CD + CD Player
  - DVD + DVD Player

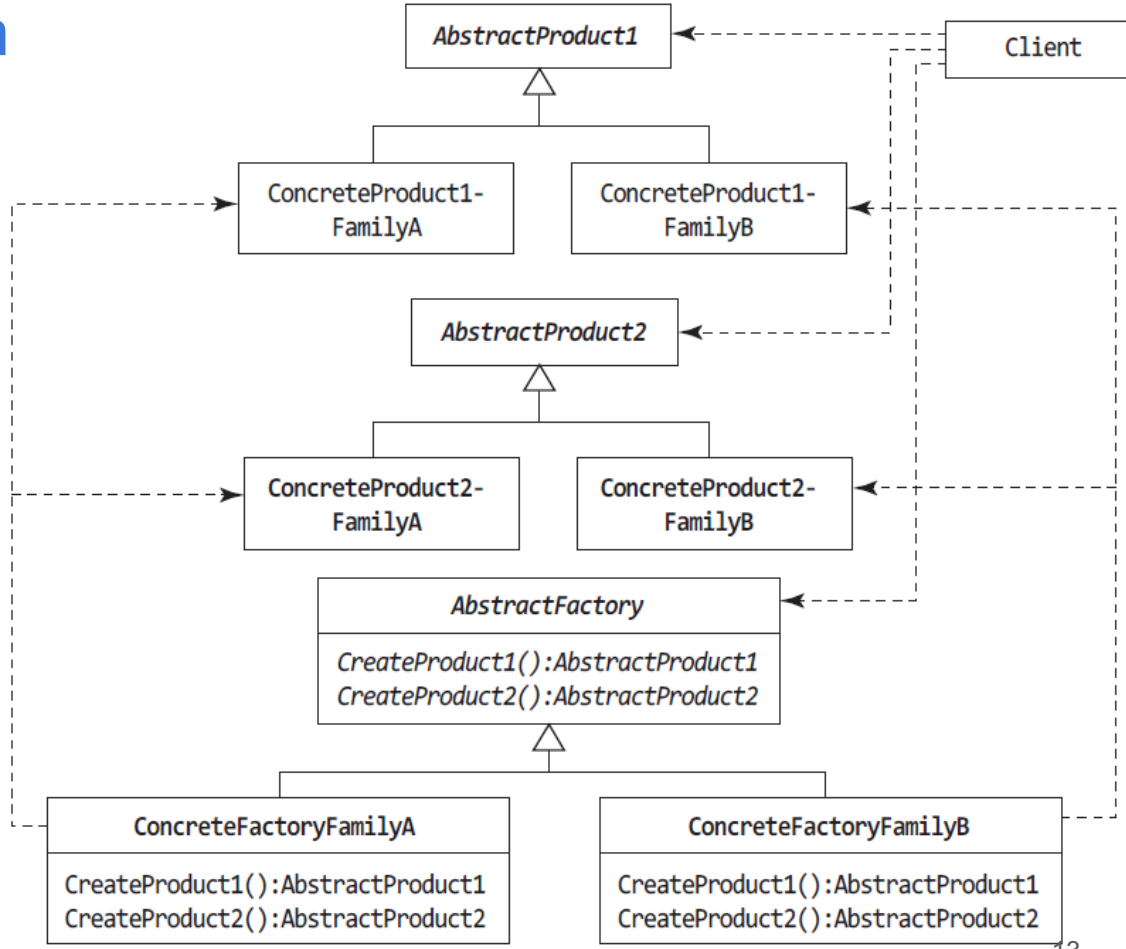


그림 7.28 추상 팩토리 패턴

Abstract Factory Pattern Example

- 음악 재생에 필요한 하드웨어 및 소프트웨어 요소를 생성
- Client는 음악 재생에만 관심 (사용되는 hardware와 software는 관심이 없음)

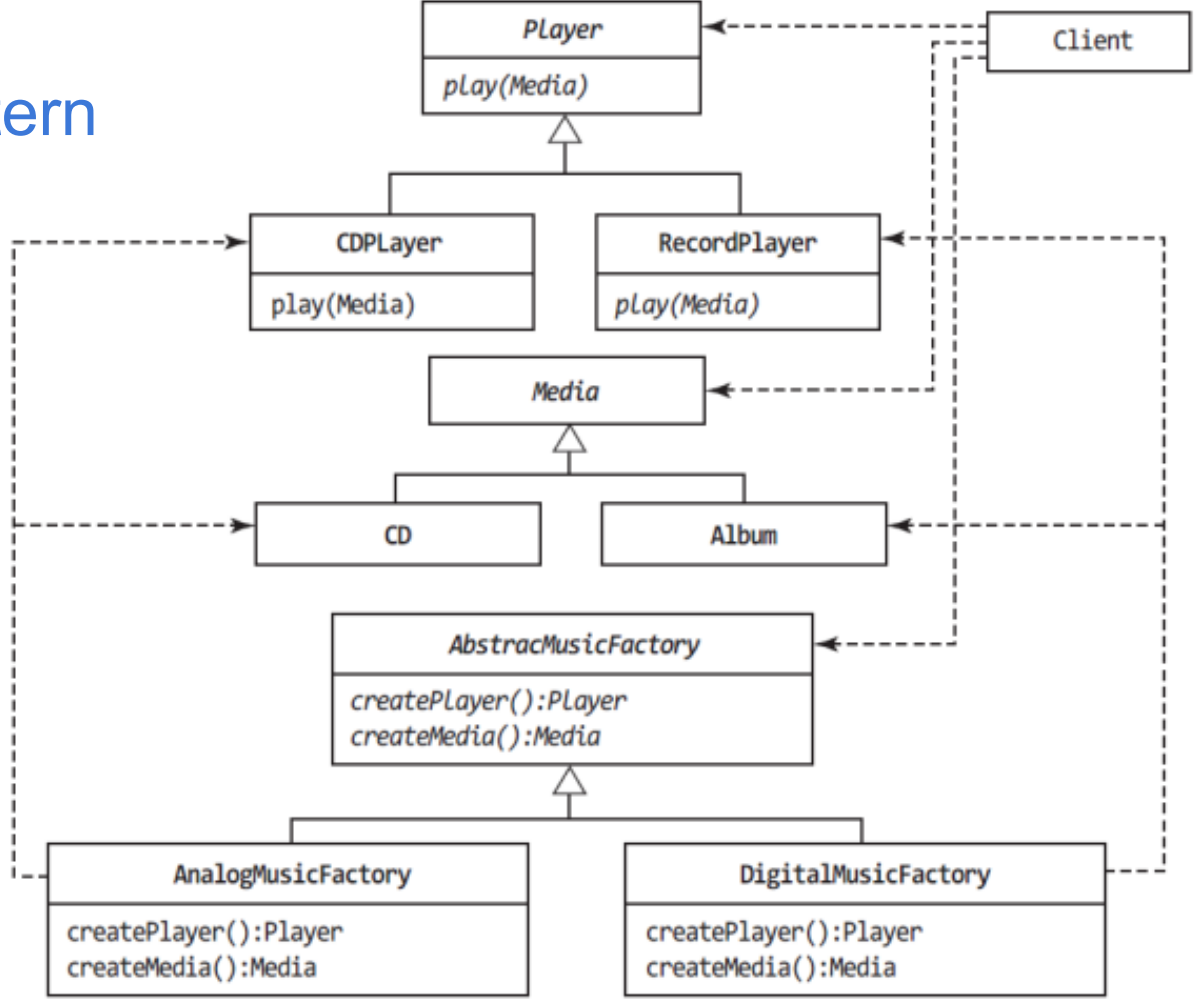


그림 7.29 추상 팩토리 패턴의 사례

Structural Patterns (구조 패턴)

- Structural Patterns
  - 프로그램 내의 데이터나 인터페이스의 구조를 설계하는 데 많이 활용
  - 클래스나 객체의 구성(합성)으로 더 큰 구조를 만들어야 할 때 유용한 디자인 패턴
- 종류
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

Adapter Pattern

- Client와 Service가 호환되지 않는 Interface가 있을 경우 적용
- 사용 가능한 Service의 Interface를 Client가 예상하는 인터페이스에 맞게 조정
- Adapter – 서비스가 제공하는 인터페이스를 클라이언트가 기대하는 인터페이스로 변환

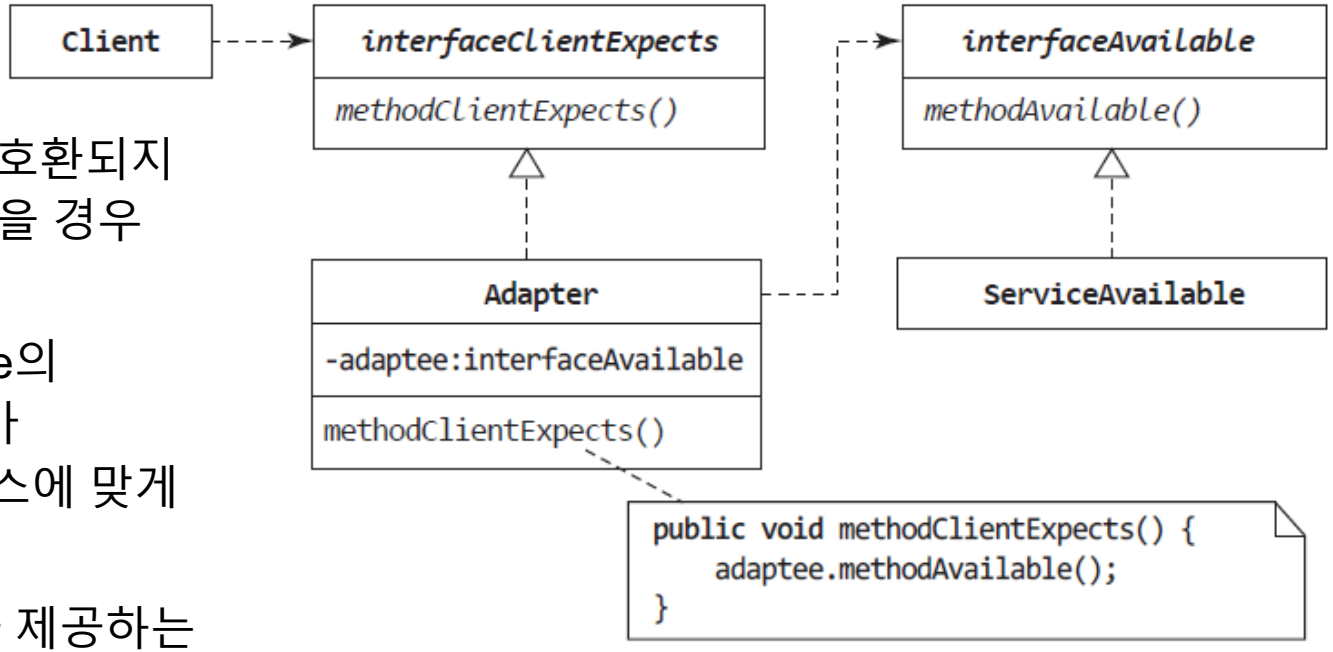


그림 7.20 어댑터 패턴

[Reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Hands-on: Adapter Pattern Case Study

- 삼성 휴대전화에 무선 이어폰 Buds 대신 애플의 무선 이어폰 (Airpods)을 사용할 수 있도록 하는 것
- 두 이어폰의 기능은 재생과 멈춤으로 한정
- Buds는 재생 기능에 play(), 멈춤 기능에 stop() 메서드를 사용
- Airpods은 재생 기능에 playing(), 멈춤 기능에 stopping() 메서드를 사용

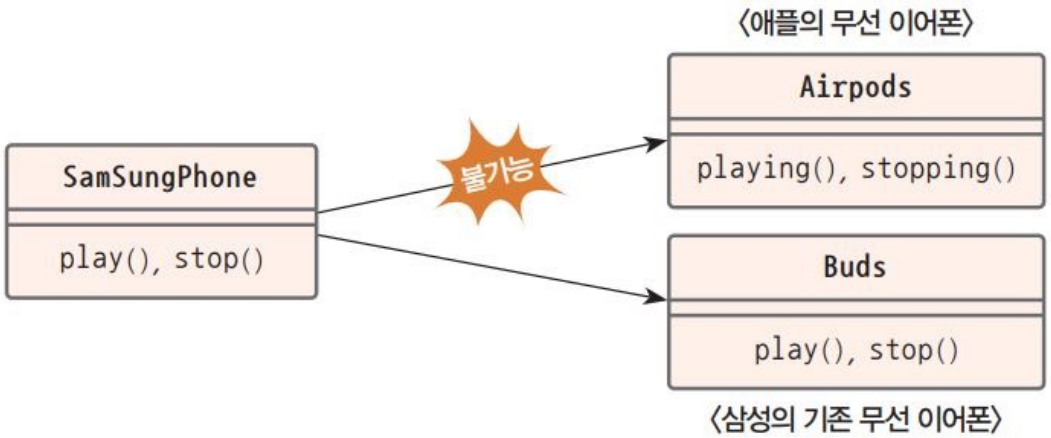


그림 7-8 삼성 휴대전화에서 애플 무선 이어폰 사용

Hands-on: Adapter Pattern Case Study

- <<interface>>타입의 AirPodsInterface 클래스를 정의하고 그 메서드는 Buds에서 사용하는 play(), stop() 메서드와 같게 함
- Adapter로 AirPodsAdapter 클래스를 만듬
- AirPods 클래스로부터 상속을 받지만 AirPodsInterface 클래스로 인터페이스 구현으로 이루어졌기 때문에 자바에서 가능

[Reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Hands-on: Adapter Pattern Case Study

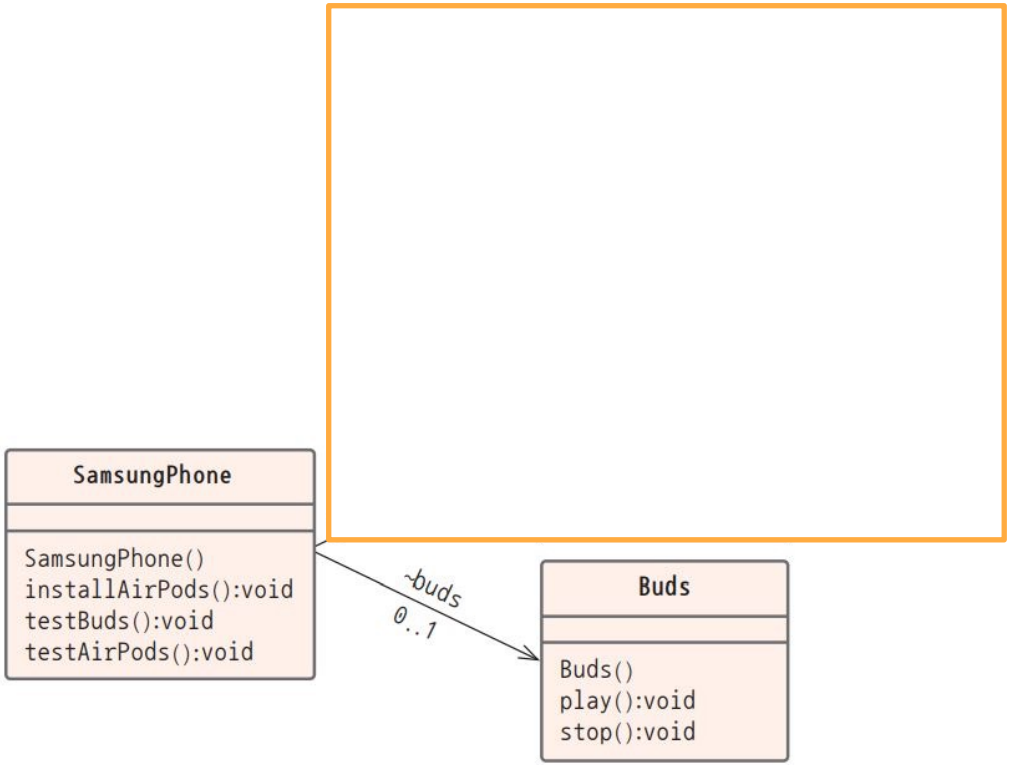


그림 7-10 adapter 패턴을 적용한 클래스 다이어그램

실행 결과

```
무선 이어폰 호환 시스템

새로 구입한 Apple AirPods입니다.
AirPods이 SamsungPhone과 호환됩니다.
Buds와 새로 구입한 AirPods을 연결합니다.

삼성 Buds 음악 재생 중…
삼성 Buds 음악을 중지합니다!

Apple AirPods 음악 재생 중…
Apple AirPods 음악을 중지합니다!
```

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Decorator Pattern

- Class의 동작을 확장하는 방법과 문제점:
  1. 수정에 의한 추가
     - OCP 원리 (확장을 위해 Open, 수정을 위해 Close)에 위배
  2. 상속에 의한 추가
     - 기능의 조합 수 만큼의 서브클래스가 필요

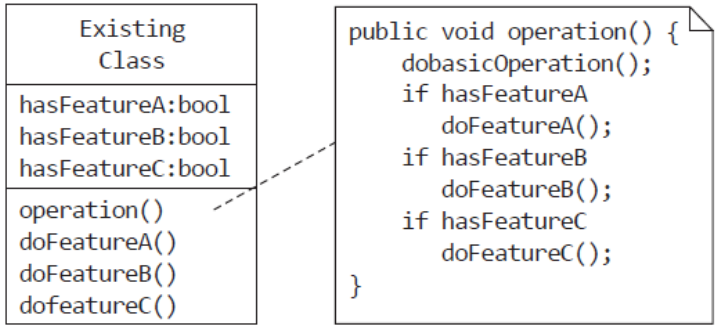


그림 7.21 수정에 의한 추가

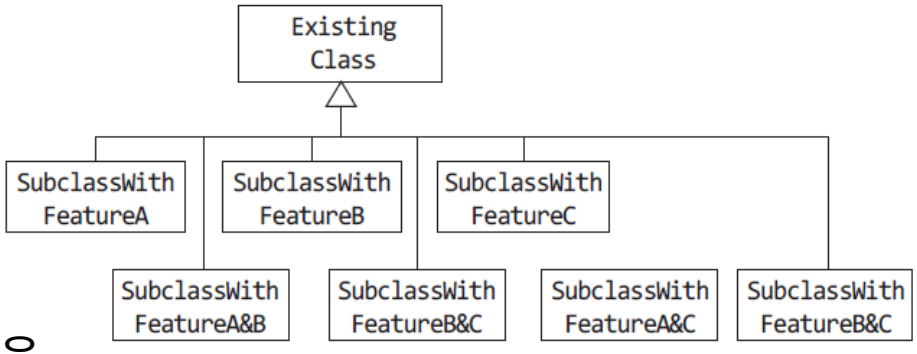
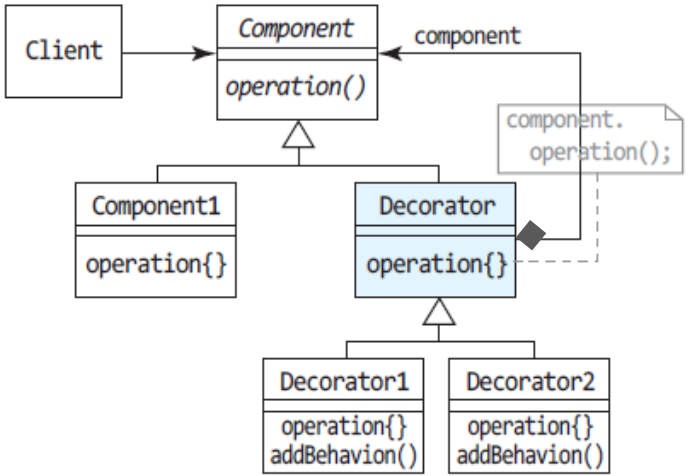


그림 7.23 상속을 이용한 기능추가 사례

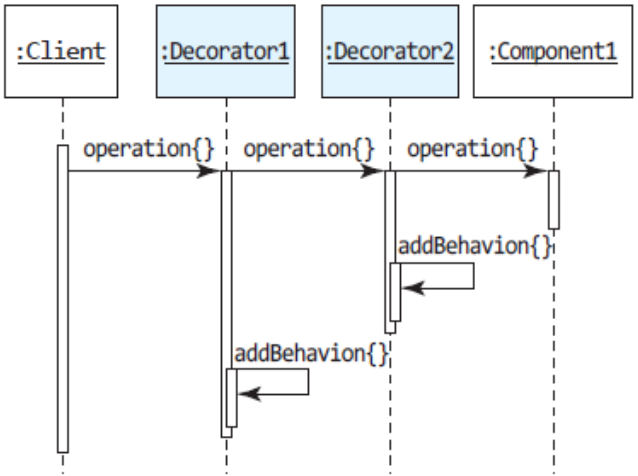
- **Decorator Pattern**: 집합 관계와 위임을 사용하여 기존 클래스의 동작을 가볍고 유연하게 확장 가능

Decorator Pattern

- 구성요소
  - 장식 대상 component 클래스와 확장 기능이 담긴 Decorator
- Decorator 객체가 재귀 합성(composite)으로 component 객체를 wrapping



(a) 정적 구조



(b) 동적인 객체 구성 및 상호작용

Hands-on: Decorator Pattern Case Study

- 토스트를 판매할 때 칼로리를 계산해 알려주는 토스트 가게가 있음
- 현재 판매하는 메뉴는 치즈 토스트, 야채 토스트, 햄 토스트로 3종류
- 앞으로 재료도 늘어날 것이고 재료를 혼합한 토스트 종류도 늘어날 예정
- 기본 재료를 조합해 만드는 토스트의 종류를 하위 클래스로 계속 추가할 수 있는 상속 구조로 만든다면 기본 재료가
  추가될 때마다 그것을 조합해 만들어지는 토스트의 종류 (클래스)는 수십 가지로 늘어날 것

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Hands-on: Decorator Pattern Case Study

- Decorator 패턴의 적용
  - Decorator 패턴을 적용하면 식빵은 상속 구조를 사용
  - 빵의 종류가 늘어나는 것에 대해서는 Toast라는 상위 클래스를 만들고 하위 클래스에 식빵의 종류를 둠
  - decorator 패턴은 재료를 기본 재료(햄, 치즈, 달걀, 야채)와 복합 재료로 나누는 것부터 시작
  - 핵심은 **혼합 재료의 숫자만큼 클래스를 만들지 않고 기본 재료만 가지고 해결하는 것**
  - 기본 재료는 하위 클래스에 놓고 상위 클래스에 ToppiongDecorator라는 이름의 클래스를 만듬
  - 혼합 재료는 기본 재료를 계속 늘려 나가는 방식으로 프로그래밍

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Hands-on: Decorator Pattern Case Study

```
//Toast 타입의 일반식빵인 toast1 객체를 생성
Toast toast1 = new NormalBread();
// '치즈'를 인자로 갖는 addTopping()을 호출
toast1.addTopping(new Cheese());
// '햄'을 인자로 갖는 addTopping()을 호출
toast1.addTopping(new Ham());
// server()를 호출해 지금까지 name에 들어간 이름과 누적된 칼로리를 출력
toast1.serve();
```

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Class diagram with decorator pattern

- Bread 종류: NormalBread, WheatBread, ButterBread, MilkBread
- Topping 종류: Ham, Vegetable, Cheese, Egg
- 예로 만들 수 있는 토스트

  WheatBread Ham Egg Toast 400 kcal

  WheatBread Egg Toast 300 kcal

  NormalBread Cheese Toast 450 kcal

  …

Hands-on: Decorator Pattern Case Study

- Java Implementation

Behavior Patterns (행위 패턴)

- Behavior Patterns
  - **반복적으로 사용되는 객체의 상호작용**을 패턴화한 것
  - 클래스나 객체가 **상호작용하는 방법과 책임을 분산**하는 방법을 정의
  - 메시지 교환과 관련된 것으로 객체 간의 행위나 알고리즘 등과 관련된 패턴
- 종류
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template method
  - Visitor

Observer Pattern

- 데이터를 보관하고 있는 Subject가 그 데이터를 이용하는 옵서버와 효과적으로 통신하면서 느슨하게 결합
- Subject 클래스 – 옵서버 목록을 유지, 변경을 고지
- Observer 클래스 – 변경을 통지 받고 접근을 요청

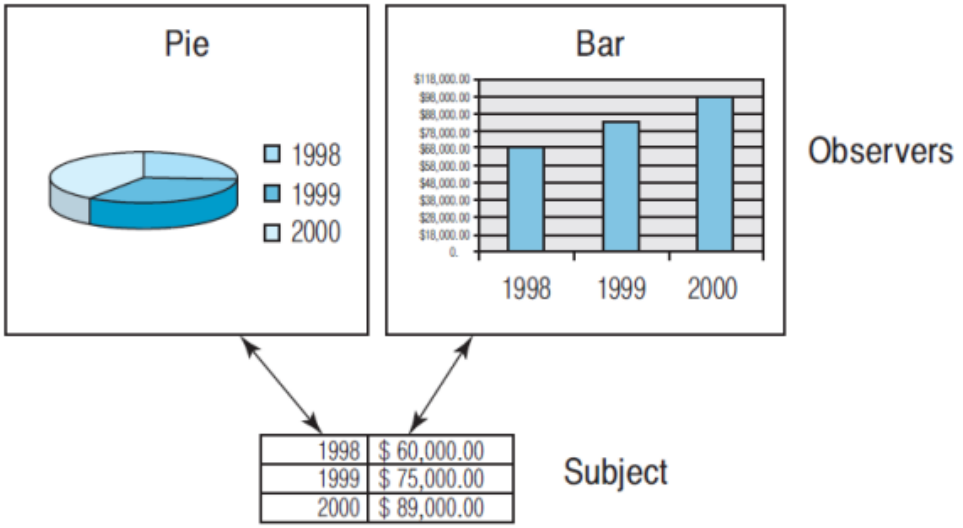
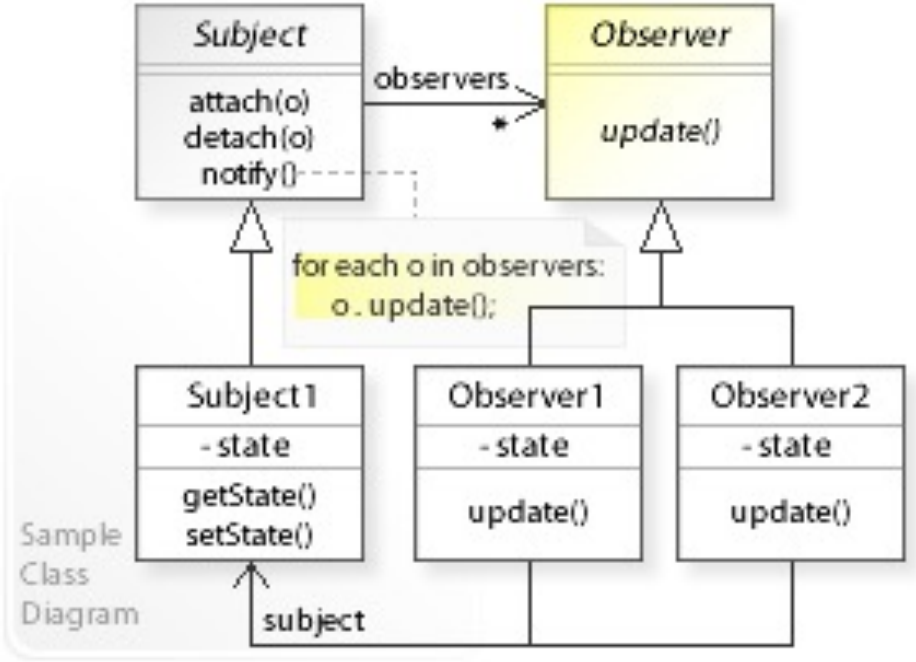
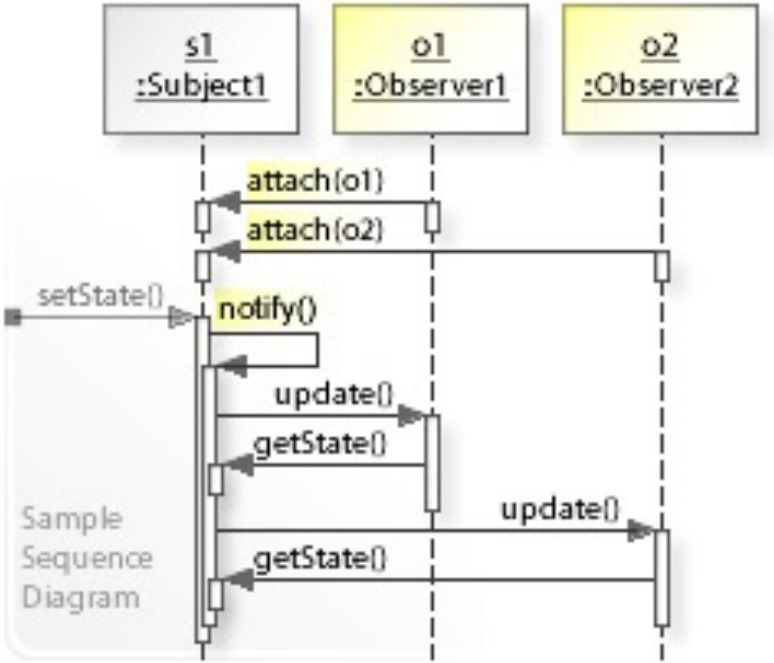


그림 7.33 옵서버 패턴이 적용될 사례

Observer Pattern





[reference: Wikipedia]

Observer Pattern Example

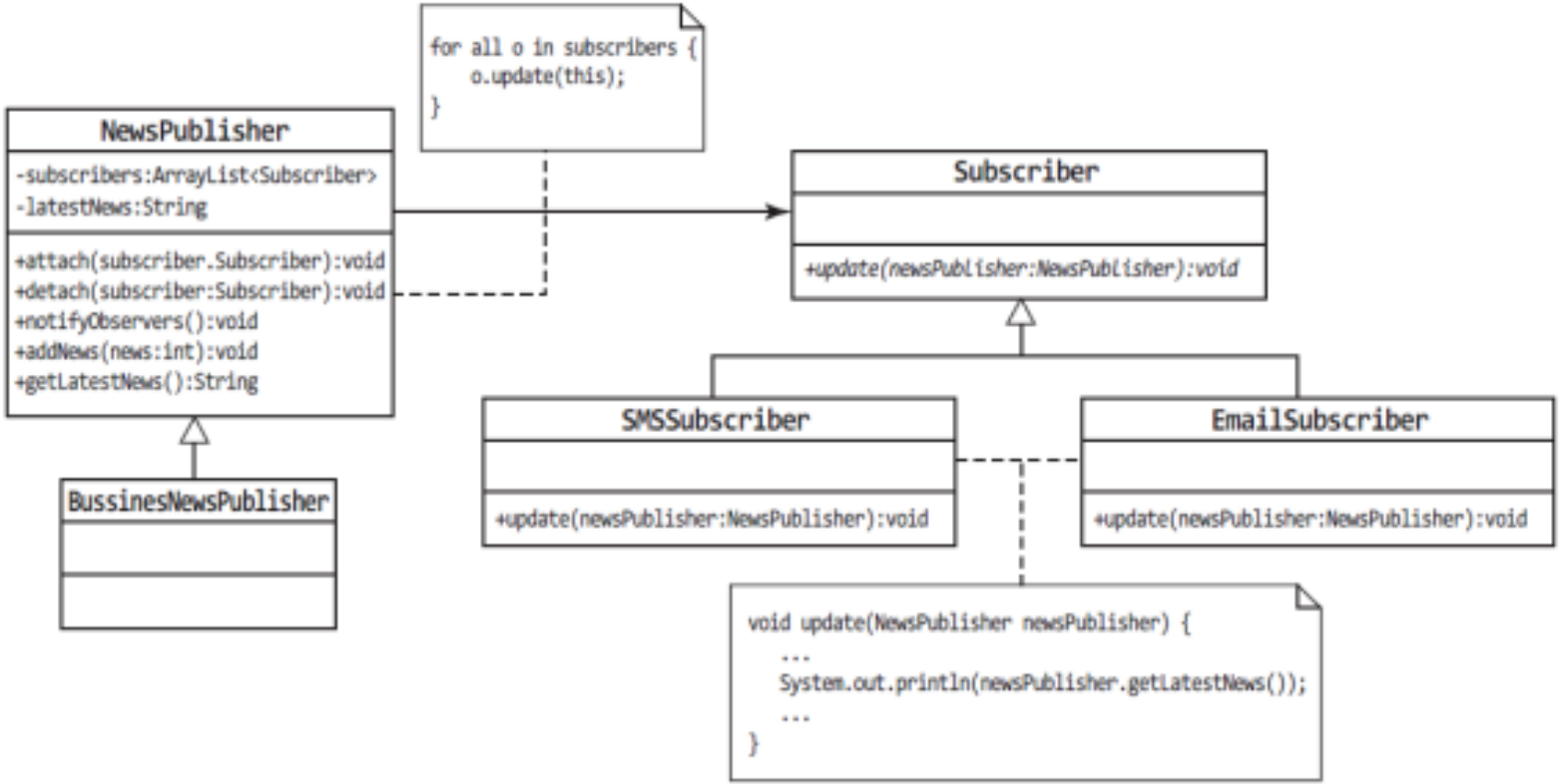
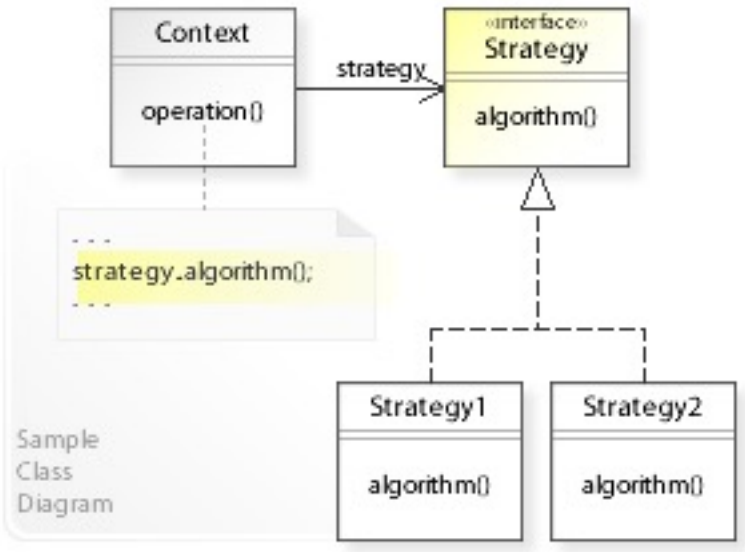
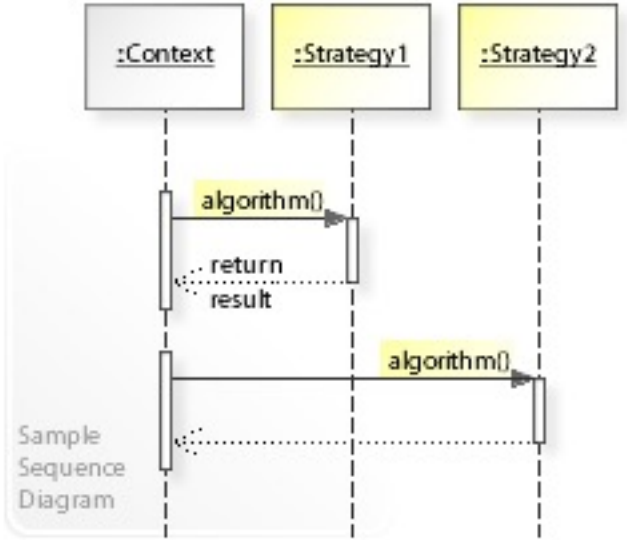


그림 7.36 옵서버 패턴 사례

Strategy Pattern

- 객체의 행위를 동적으로 바꾸고 싶은 경우 직접 행위를 수정하지 않고 전략을 바꿔주기만 함으로써 행위를 유연하게 확장하는 방법.
  - **자주 바뀌는 것이 기능일 경우, 메서드를 클래스로 바꾸고** <<interface>> 타입의 상속 구조를 만드는 패턴





[reference: Wikipedia]

Strategy Pattern: Pokemon Project

- 문제: 기능 변화
  - 문제점
    - 기술이 발전함에 따라 공격과 패시브 기술이 계속 교체되지만 이에 대한 해결책이 없음
    - 클래스 설계 원칙 중 OCP(개방 폐쇄 원칙)를 위반하는 것일 뿐 아니라 프로그램을 복잡하게 만듬
  - 해결책
    - 자주 바뀌는 것이 무엇인지 찾아 그들을 별도의 상속 구조로 만들어 사용
    - **자주 바뀌는 것이 attack(), passive()이기 때문에 이들을 클래스로 만들고 상속 구조로 설계**
    - 추가되는 공격 기술을 하위 클래스에 계속 추가할 수 있음

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Strategy Pattern: Pokemon Project

표 7-1 포켓몬의 종류

포켓몬	기능(공격/패시브)	설명
피카츄	백만 볼트	백만 볼트의 강력한 전압을 상대에게 보내 공격한다.
	스피드	한 번에 빠르게 두 번 공격한다.
푸린	노래 부르기	노래를 불러 상대를 잠재운다.
	회피	30% 확률로 공격을 회피한다.
파이리	불꽃	뜨거운 불꽃을 상대방에게 쏘아 공격한다.
	방어	방어 전략을 사용해 받는 피해를 40% 감소시킨다.

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Strategy Pattern: Pokemon Project

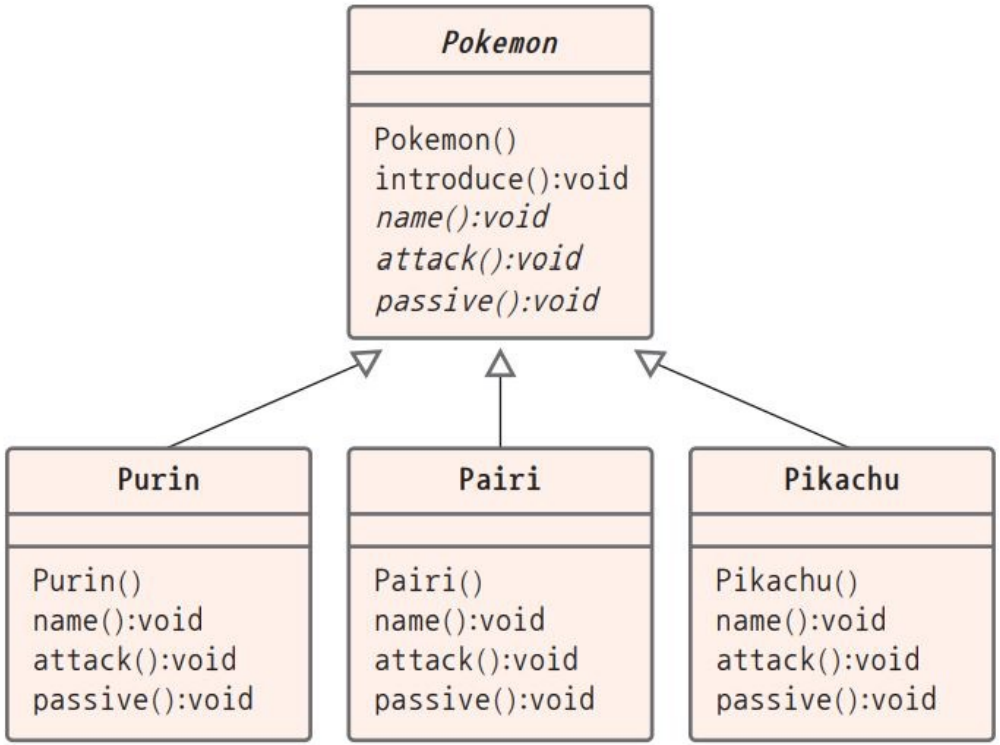


그림 7-1 strategy 패턴 사용 전 클래스 다이어그램

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

Strategy Pattern을 적용한 설계: Class Diagram

- 포케몬 종류: Purin, Pairi, Pikachu
- attack과 passive skill이 자주 변경
- 포케몬 객체를 생성하고 introduce() 메소드로 기능 내용 보여주기

Strategy Pattern: Pokemon Project

- Java implementation

State Pattern

- 상태에 따라 객체의 동작을 변경해야 하는 경우
- Context과 State를 별도로 구현하여 융통성을 달성하기 위한 체계적이고 느슨한 결합 방식
- **자주 변경되는 것이 State일 경우, State를 클래스로 바꾸고 <<interface>>타입의 상속 구조를 만든다**

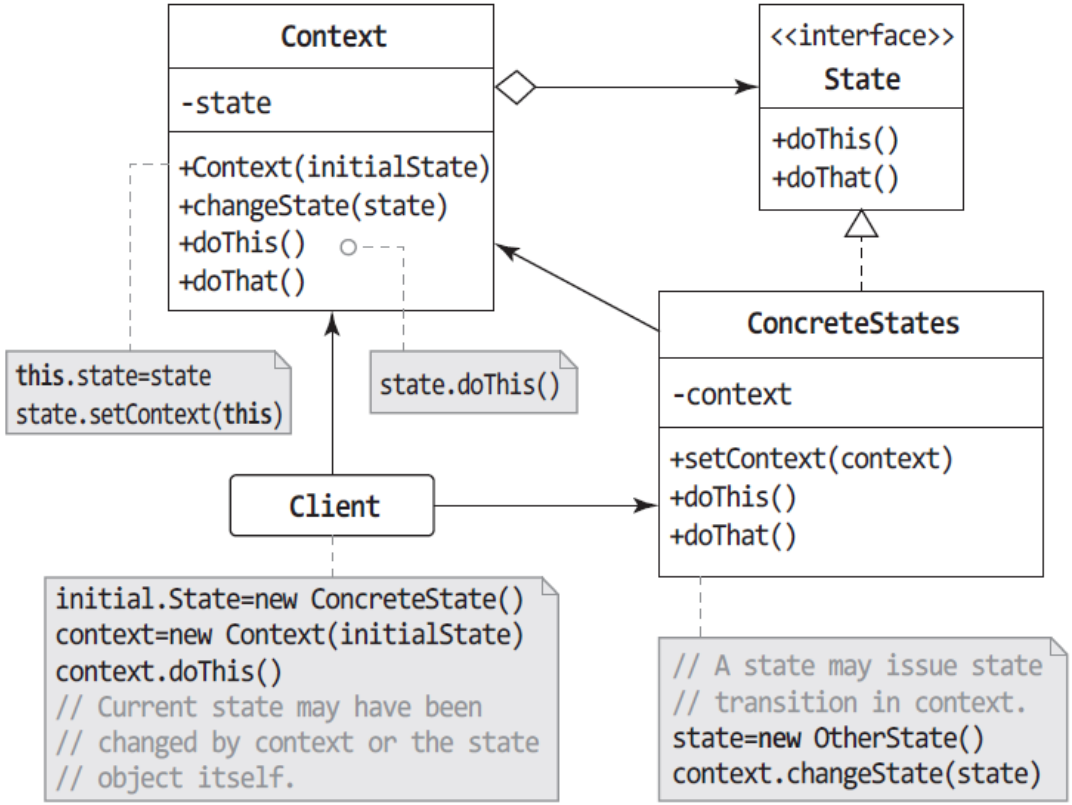
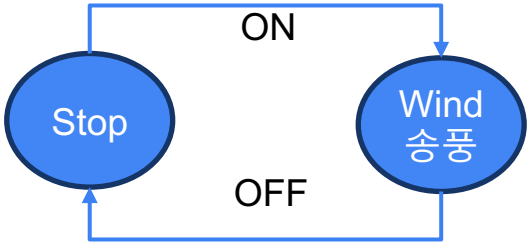


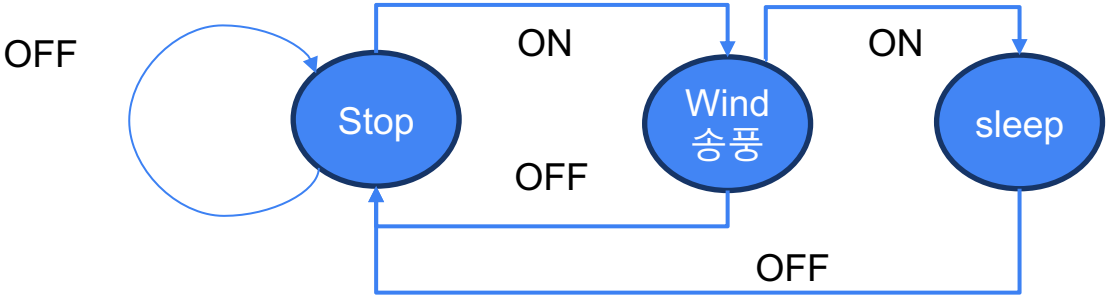
그림 7.31 상태 패턴

State Pattern Example

- 처음의 선풍기는 Stop와 송풍 기능만 있는 것
- 정지 상태 에서 ON 버튼을 누르면 송풍 상태로 변하고 송풍 상태에서 OFF 버튼을 누르면 정지 상태로 변한



- 선풍기의 바람 상태에 Sleep, (약풍, 중간풍, 강풍, ..)이 계속 추가될 것



State Pattern이 적용 안된 구현

```
public class ElecFan {
  private String State;
  public ElecFan() {
    State = "Stop";
    System.out.println("<<현재 상태: " + State + ">>");
  }
  public void setState(String state) {
    this.State = state;
  }
  public void on_button() {
    if(State == "Stop")  {
      State = "Wind";
      System.out.println("\n***on 버튼 눌림***\n" + "정지에서 송풍
상태로 바뀜\n");
      System.out.println("<<현재 상태: " + State + ">>");
    }
    else if(State == "Wind") {
      State = "Stop";
      System.out.println("\n***on 버튼 눌림***\n" + "상태 변화 없음
\n");
      System.out.println("<<현재 상태: " + State + ">>");
    }
  }
```

```
  public void off_button() {
    if(State == "Stop") {
      State = "Stop";
      System.out.println("\n***off 버튼 눌림***\n" + "상태 변화 없음
\n");
      System.out.println("<<현재 상태: " + State + ">>");
    }
    else if(State == "Wind") {
      State = "Stop";
      System.out.println("\n***off 버튼 눌림***\n" + "송풍에서 정지
상태로 바뀜\n");
      System.out.println("<<현재 상태: " + State + ">>");
    }
  }
}
```

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

State Pattern Example

[reference: 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미]

- 문제: 상태 변화
  - 문제점
    - ElecFan 클래스에서는 조건문(if ~ else if)을 사용했는데 이 코드만으로는 상태의 변화를 쉽게 읽을 수 없음
    - 상태(추가, 삭제)가 바뀌면 on_button(), off_button() 메서드에 이를 반영해야 하므로 계속해서 코드 수정이 필요
    - 클래스 설계 원칙 중 OCP(개방 폐쇄 원칙)를 위반하는 것일 뿐 아니라 프로그램을 복잡하게 만들어 유지보수를 어렵게 함
  - 해결책
    - **자주 바뀌는 것이 상태(송풍, 정지, 수면, 약풍, 중간풍, 강풍 등)이므로 조건문으로 처리하던 상태를 클래스로 변경**
    - 클래스를 상속 구조(인터페이스 포함) 형태로 만들어 하위 클래스에서 상태를 추가 및 삭제할 수 있도록 만듬

State Pattern을 적용한 설계: Class Diagram

Further Reading

- https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns
- Wikipedia pages for each pattern

References

[1] 소프트웨어 공학의 모든 것, 최은만, 생능출판
[2] 쉽게 배우는 소프트웨어 공학, 김치수, 한빛아카데미
[3] Design Patterns: Element of Reusable Object-Oriented Software, GoF
[4] Wikipedia