제12장 : 설계 원칙, 디자인 패턴

좋은 설계 원칙이란?

널리 알려진 설계 원칙을 배우고 의식하며 개발을 하자

좋은 코드를 위해 의식해야 하는 6가지 코드 작성의 원칙

1. DRY
2. PIE
3. SRP
4. OCP
5. SDP
6. ADP

DRY - Don’t Repeat Yourself : 같은 것을 몇번씩 반복하지 말라

- 중복 코드가 있다면 메소드로 분리한다

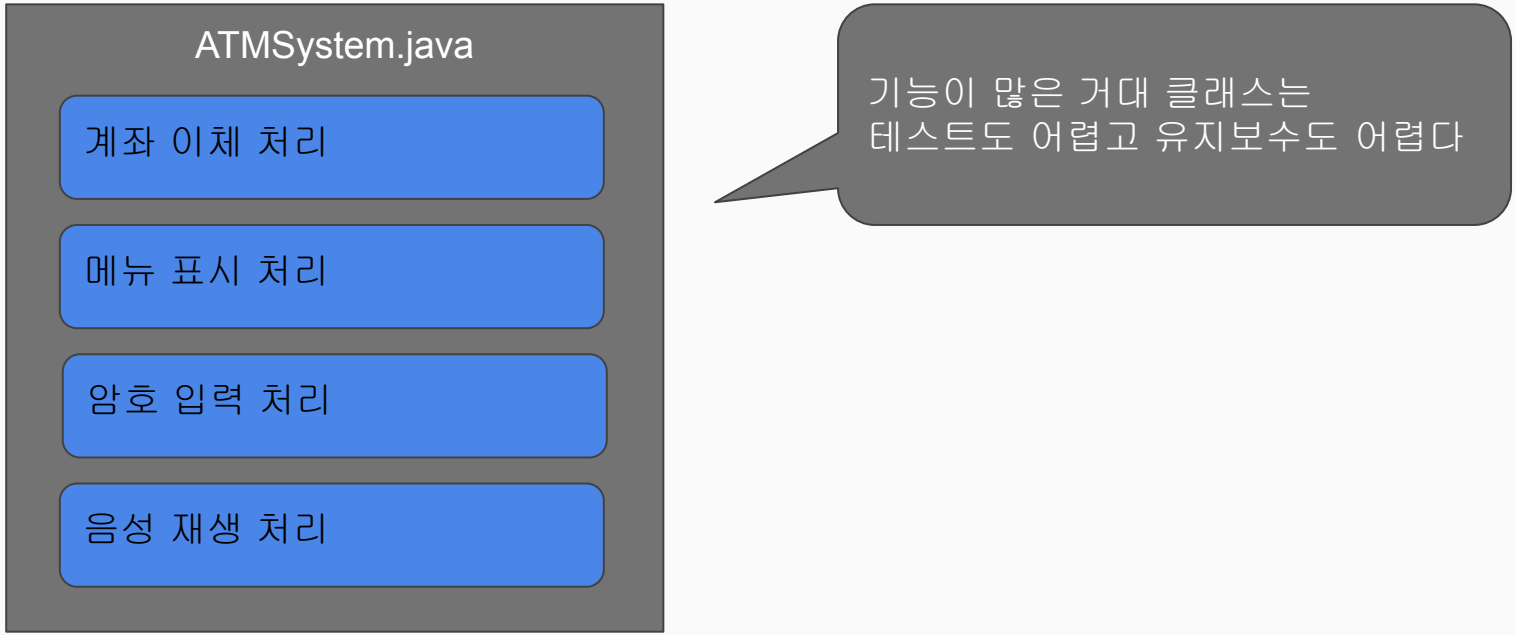
PIE - Program Intently and Expressively : 명확하고 표현력 있게 기술하자

- 애매한 이름은 쓰지 말자
- 누가 봐도 알기 쉬운 이름을 쓰자
- 컨벤션을 따르자
- 매직 넘버에 이름을 붙이자

```
static final double TAX = 1.1;
double result = value * TAX;  // value * 1.1 보다 알기 쉬움
```

SRP - Single Responsibility Principle : 클래스에 주어진 책임은 1개뿐

- 단일 책임 원칙
- 1개의 클래스는 1개의 일만한다.
- 한 부분의 에러를 수정하기 위해서는 그 클래스만 수정하면 된다.
- 하지만 클래스 분리가 심해지면 오히려 관리가 어렵기도 하다.



OCP - Open Closed Principle : 개방 폐쇄 원칙

- 확장에 대해서는 열려있고 (확장은 자유롭고), 변경에 대해서는 닫혀있다 (의존 부분의 변경은 불필요)
- 즉, 수정 없이 확장 가능하도록 하자.
- ArrayList, HashMap 등이 좋은 예. 얼마든지 나도 비슷한 클래스를 만들 수 있다.
- String 의 경우는 상속 금지이므로 OCP에 반하는 클래스의 대표적인 예.
- 추상 클래스와 인터페이스를 적극 활용하여 확장 가능하게 하자.

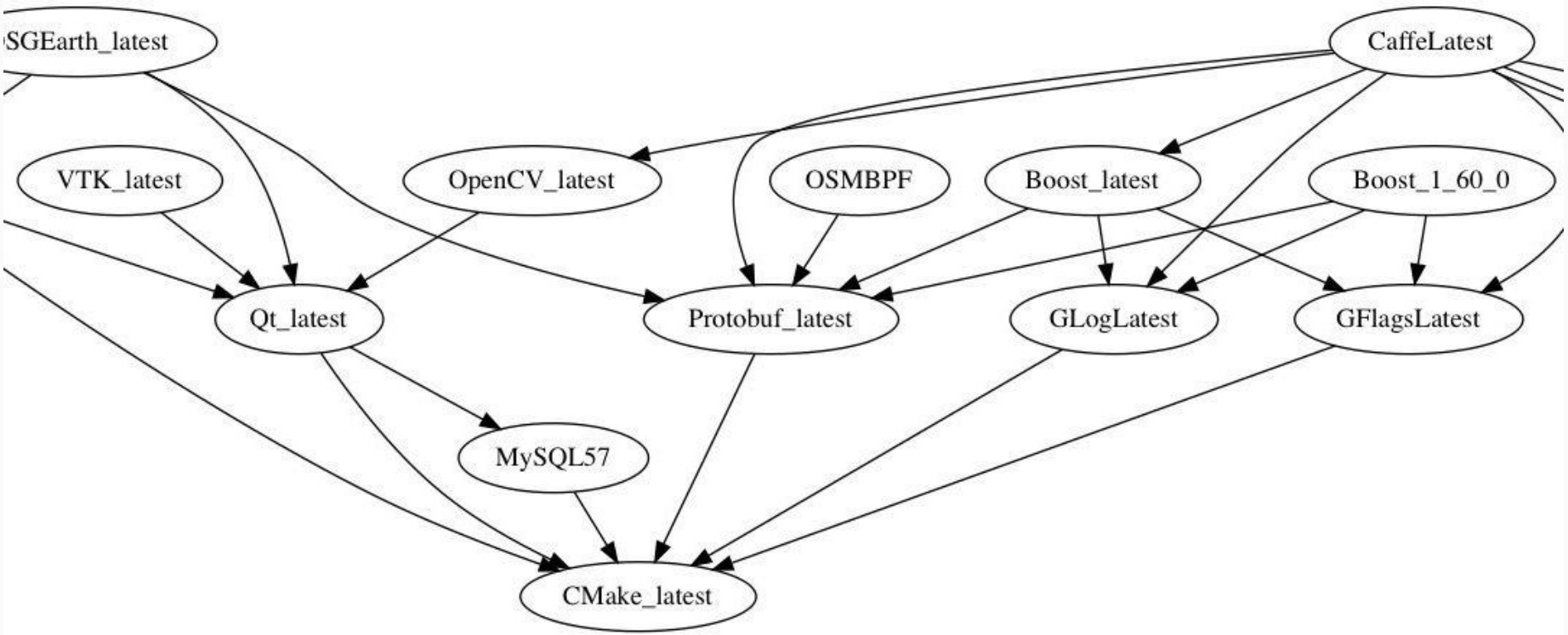
클래스의 의존 관계

- Account 의 생성자의 파라미터 수가 변경되면 Main 클래스도 수정해야 함
- 따라서 Main 은 Account 에 의존한다고 할 수 있다

```
public class Account {
  public Account(String owner) { ... }
}

public class Main {
  public static void main(String[] args) {
    Account account = new Account("홍길동");
  }
}
```

클래스의 의존 관계와 수정의 영향

- 의존성 지옥

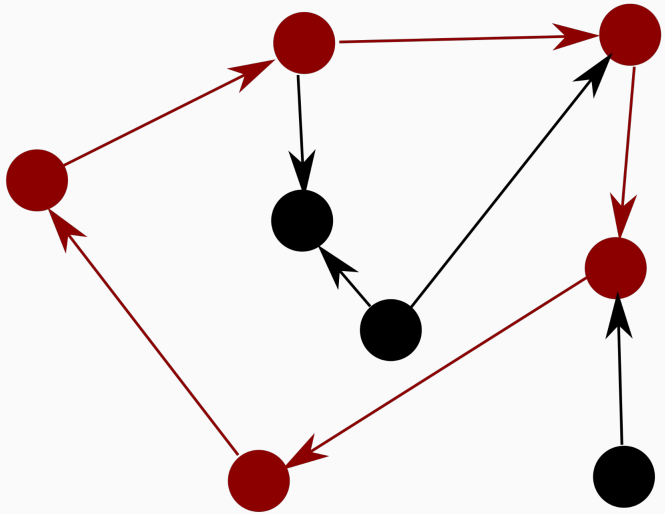


SDP - Stable Dependencies Principle : 안전한 것에 의존하라

- ATM 시스템을 예를 들면 암호 처리 같이 한번 완성되면 수정될 가능성이 없는 클래스에 의존할 만 하다
- 하지만 가장 좋은 것은 특정 클래스가 아니라 인터페이스에 의존하는 것이다
- 클래스는 생성자가 변하거나 할 수 있으나 인터페이스는 거의 그대로이니까.

ADP - Acyclic Dependencies Principle

- 의존성 비순환 원칙
- 의존 관계에 사이클이 발생되지 않게 한다



또 다른 코드 설계 원칙

- SOLID 원칙

두문자	약어	개념
S	SRP	**단일 책임 원칙 (Single responsibility principle)** 한 클래스는 하나의 책임만 가져야 한다.
O	OCP	**개방-폐쇄 원칙 (Open/closed principle)** "소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다."
L	LSP	**리스코프 치환 원칙 (Liskov substitution principle)** "프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다." 계약에 의한 설계를 참고하라.
I	ISP	**인터페이스 분리 원칙 (Interface segregation principle)** "특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다."[4]
D	DIP	**의존관계 역전 원칙 (Dependency inversion principle)** 프로그래머는 "추상화에 의존해야지, 구체화에 의존하면 안된다."[4] 의존성 주입은 이 원칙을 따르는 방법 중 하나다.

디자인 패턴

소프트웨어 디자인 패턴(software design pattern)은 소프트웨어 공학의 소프트웨어 디자인에서 특정 문맥에서 공통적으로 발생하는 문제에 대해 재사용 가능한 해결책이다.

디자인 패턴은 프로그래머가 어플리케이션이나 시스템을 디자인할 때 공통된 문제들을 해결하는데에 쓰이는 형식화 된 가장 좋은 관행이다.

결론 : 설계 원칙과 노하우를 정리한 것. 선배님들이 정리한 것을 공부하자.

디자인 패턴을 공부했을 때의 장점

- 개발자간에 커뮤니케이션이 원만해 진다
- 객체지향 설계 원칙의 이해도가 좋아진다

GoF의 23가지 디자인 패턴

Gang Of Four 라는 4명의 개발자가 정리한 디자인 패턴

사실상 지금까지 디자인 패턴이라고 부르는 것은 다 여기서 나온 것이다.

생성(Creational) 패턴	구조(Structural) 패턴	행위(Behavioral) 패턴
• 추상 팩토리(Abstract Factory) • 빌더(Builder) • 팩토리 메서드(Factory Methos) • 프로토타입(Prototype) • 싱글턴(Singleton)	• 어댑터(Adapter) • 브리지(Bridge) • 컴퍼지트(Composite) • 데커레이터(Decorator) • 퍼사드(Facade) • 플라이웨이트(Flyweight) • 프록시(Proxy)	• 책임 연쇄(Chain of Responsibility) • 커맨드(Command) • 인터프리터(Interpreter) • 이터레이터(Iterator) • 미디에이터(Mediator) • 메멘토(Memento) • 옵서버(Observer) • 테이트(State) • 스트래티지(Strategy) • 템플릿 메서드(Template Method) • 비지터(Visitor)

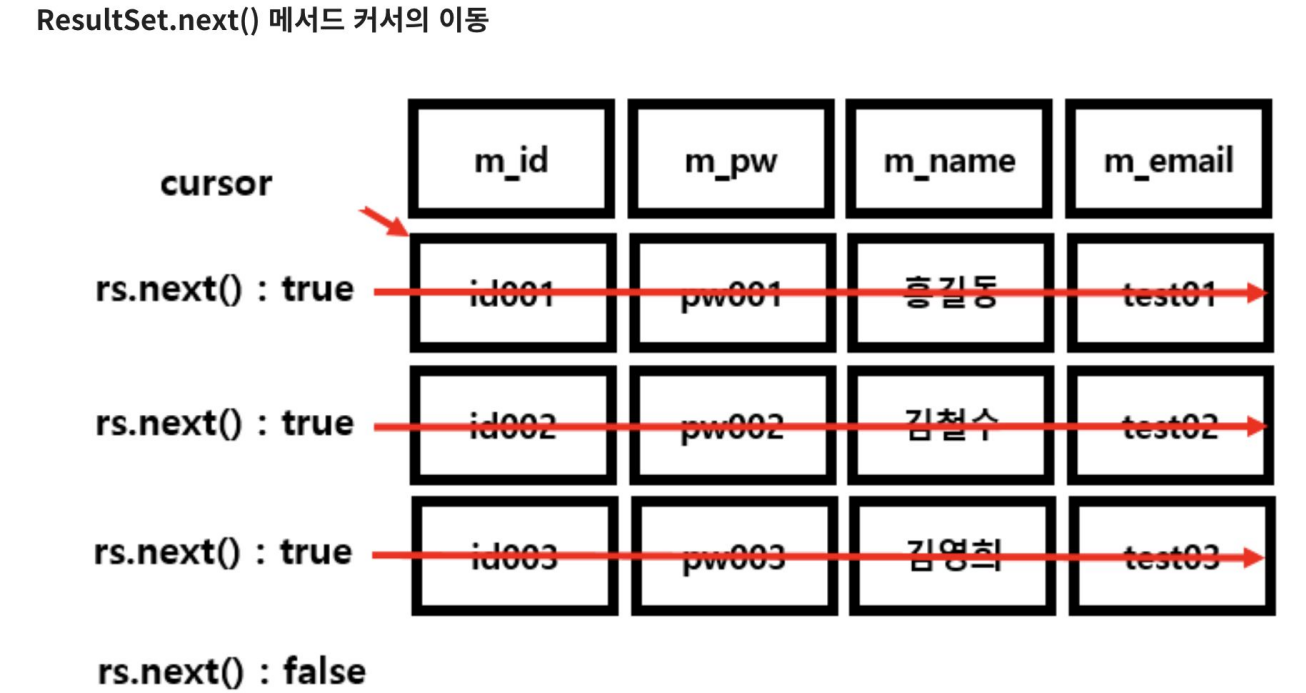
초보 때 주로 사용하는 패턴

당장은 이 정도만 알아도 충분합니다.

생성(Creational) 패턴	구조(Structural) 패턴	행위(Behavioral) 패턴
• 추상 팩토리(Abstract Factory) • 빌더(Builder) • 팩토리 메서드(Factory Methos) • 프로토타입(Prototype) • 싱글턴(Singleton)	• 어댑터(Adapter) • 브리지(Bridge) • 컴퍼지트(Composite) • 데커레이터(Decorator) • 퍼사드(Facade) • 플라이웨이트(Flyweight) • 프록시(Proxy)	• 책임 연쇄(Chain of Responsibility) • 커맨드(Command) • 인터프리터(Interpreter) • 이터레이터(Iterator) • 미디에이터(Mediator) • 메멘토(Memento) • 옵서버(Observer) • 테이트(State) • 스트래티지(Strategy) • 템플릿 메서드(Template Method) • 비지터(Visitor)

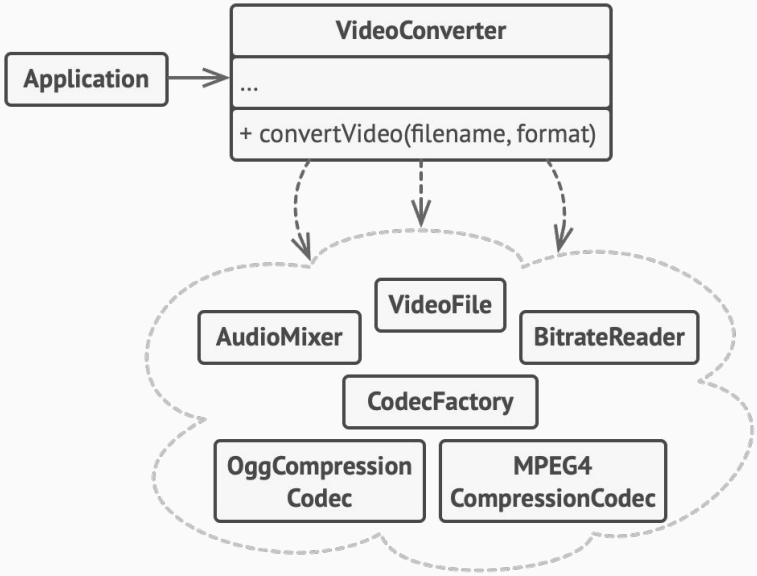
Iterator - 반복 구조

- java.util.Iterator
- java.sql.ResultSet



Facade - 내부를 감추고 심플하게

- 이사를 가면 주소를 변경할 때, 주민센터, 카드회사, 학교 등에 알려야 한다. 이를 하나하나 일일이 하는 것은 매우 귀찮기 때문에 한번에 해 주는 서비스가 있다면 그것을 이용하면 된다는 이론



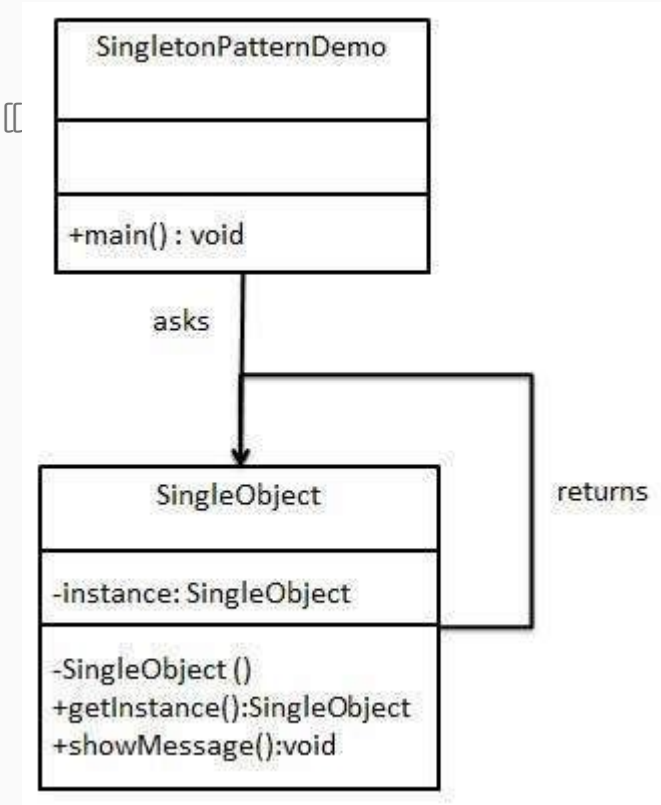
Singleton - 유일무이한 존재

객체의 인스턴스가 **오직 1개만** 생성되는 패턴을 의미한다.

주로 빈번히 사용하는 Database 등의 인스턴스 생성을 제한할 때

많이 사용한다.

Java 클래스 중에서는 Calendar 클래스가 있다.



유일한 꽃이 2개인 오류

- 휴먼 에러 유발

```
class OnlyOneFlower { ... } // 하나뿐인 꽃

public class Main {
  public static void main(String[] args) {
    OnlyOneFlower flower1 = new OnlyOneFlower();
    OnlyOneFlower flower2 = new OnlyOneFlower();
  }
}
```

싱글턴 패턴 적용

- private 생성자로 new 금지
- static 메서드로 1개의 인스턴스 사용 강제화

```
public final class OnlyOneFlower {
  private static OnlyOneFlower instance;

  private OnlyOneFlower() {}  // new 금지

  public static OnlyOneFlower getInstance() {
    if (instance == null) {
      instance = new OnlyOneFlower();
    }
    return instance;
  }
}
```

```
OnlyOneFlower flower = new OnlyOneFlower();    // error
OnlyOneFlower flower = OnlyOneFlower.getInstance(); // OK
```

Strategy - 갈아 끼우기 쉽게

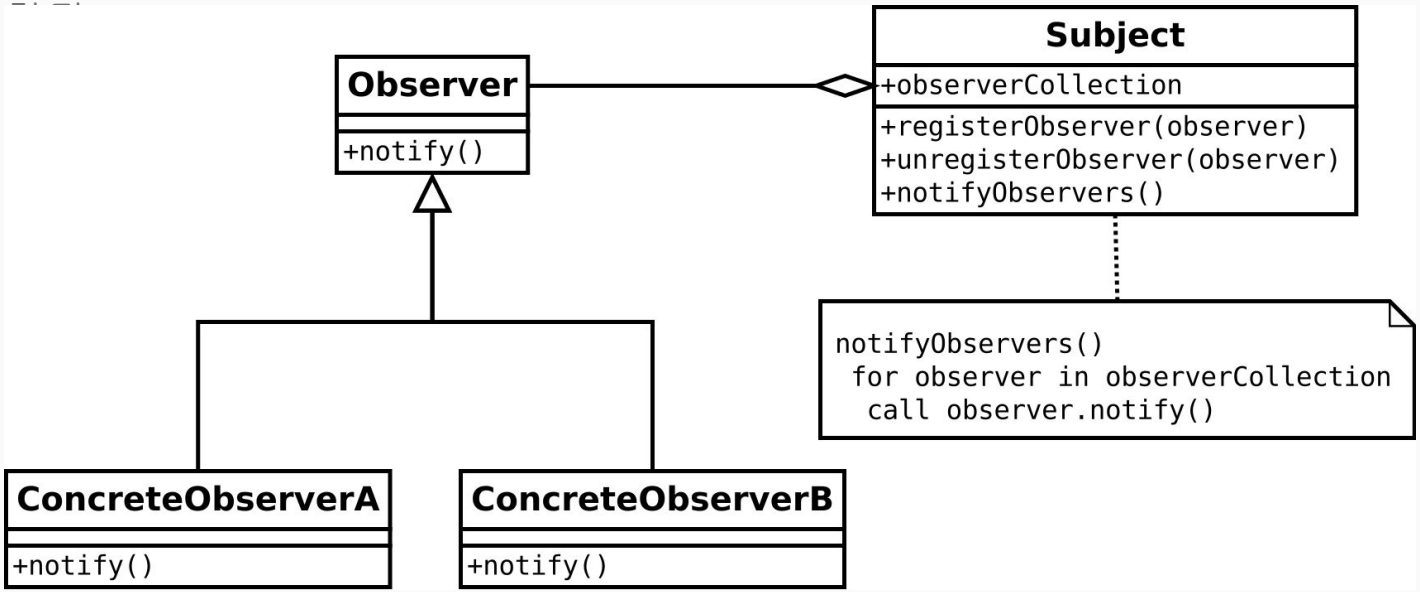
여러가지 중에서 고를 수 있도록

```
interface GameAi {
  ...
}

class NormalAi implements GameAi {
  ...
}

class HardAi implements GameAi {
  ...
}


Game game = new Game(NormalAi()); // normal 난이도
Game game = new Game(HardAi()); // hard 난이도
```

Observer 패턴

옵서버 패턴(observer pattern)은 객체의 상태 변화를 관찰하는 관찰자들, 즉 옵저버들의 목록을 객체에 등록하여 상태 변화가 있을 때마다 메서드 등을 통해 객체가 직접 목록의 각 옵저버에게 통지하도록 하는 디자인 패턴이다.

주로 분산 이벤트 핸들링 시스템을 구현하는 데 사용된다. 발행/구독 모델로 알려져 있기도



옵저버 패턴을 활용하기 위해서는 콜백 인터페이스를 이해해야 한다

간단한 콜백 인터페이스를 활용한 예.

```
interface OnClickEvent {
    void onClick();
}

class Button {

    void click(OnClickEvent listener) {
        listener.onClick();
    }
}
```

연습문제

디자인 패턴을 활용하여 다음 조건에 맞는 MyLogger 클래스를 만드시오.
File을 닫는 처리는 생략해도 됨.

- 인스턴스화와 동시에 dummylog.txt 파일을 연다
- 인수로 전달하는 문자열을 파일에 쓰는 log() 메소드를 가진다
- 다음과 같이 사용해도 에러를 내지 않고 2개의 로그 메시지가 동일 파일에 순서대로 출력되어야 한다.

```
MyLogger logger1 = // 로거 인스턴스를 얻기
logger1.log("first");
MyLogger logger2 = // 로거 인스턴스를 얻기
logger2.log("second");
```