

Dependency Inversion Principle

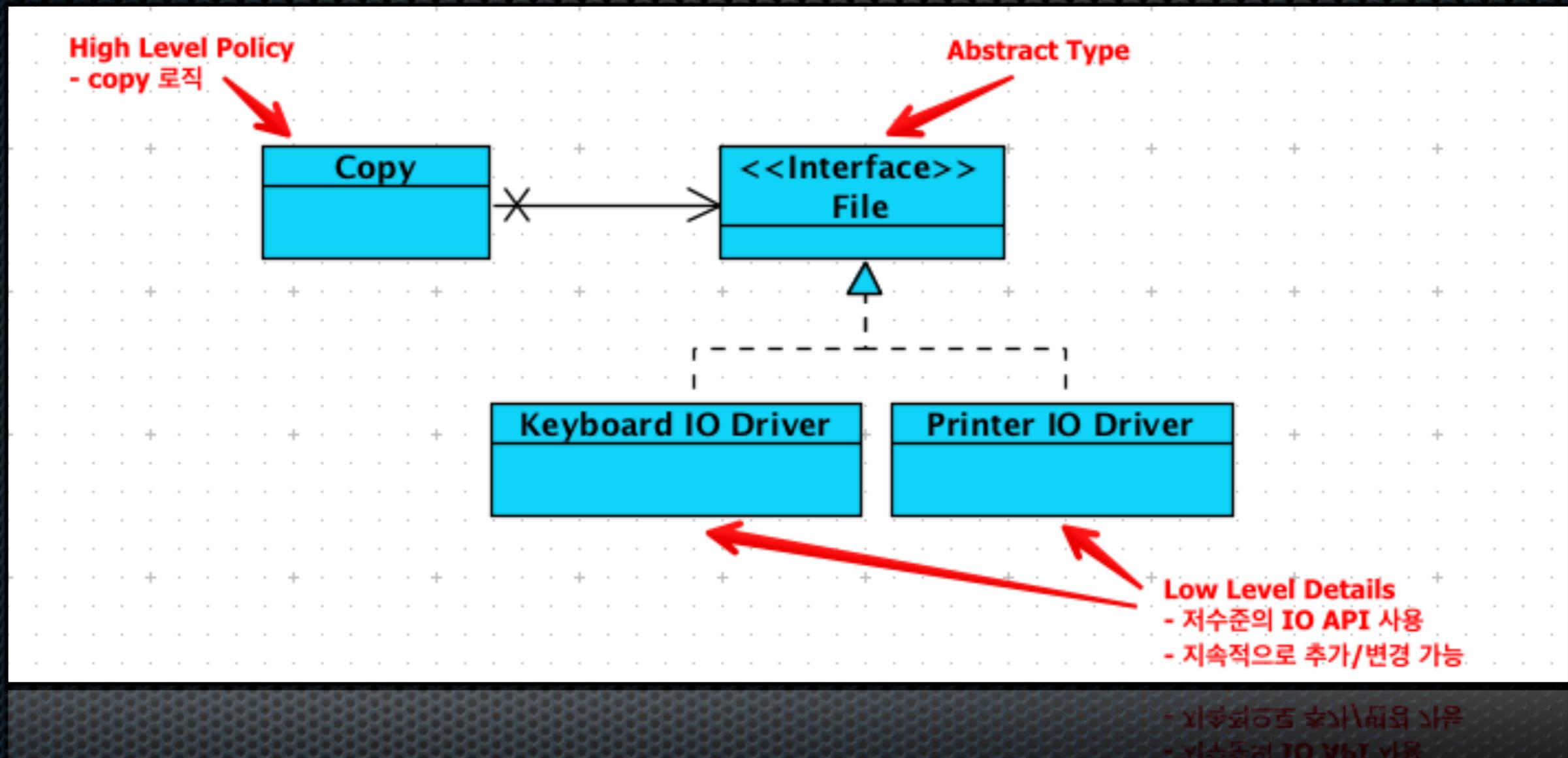
Daum Corp.
백명석

발표목차

- Dependency Inversion Principle
- OO의 핵심
- Structured Design
- Dependency Inversion
- Plugins
- Architectural Implications
- A Reusable Framework
- The Furnace Example

Dependency Inversion Principle

- High Level Policy should not depend on Low Level Details
- 둘은 Abstract Type에 의존해야 한다.

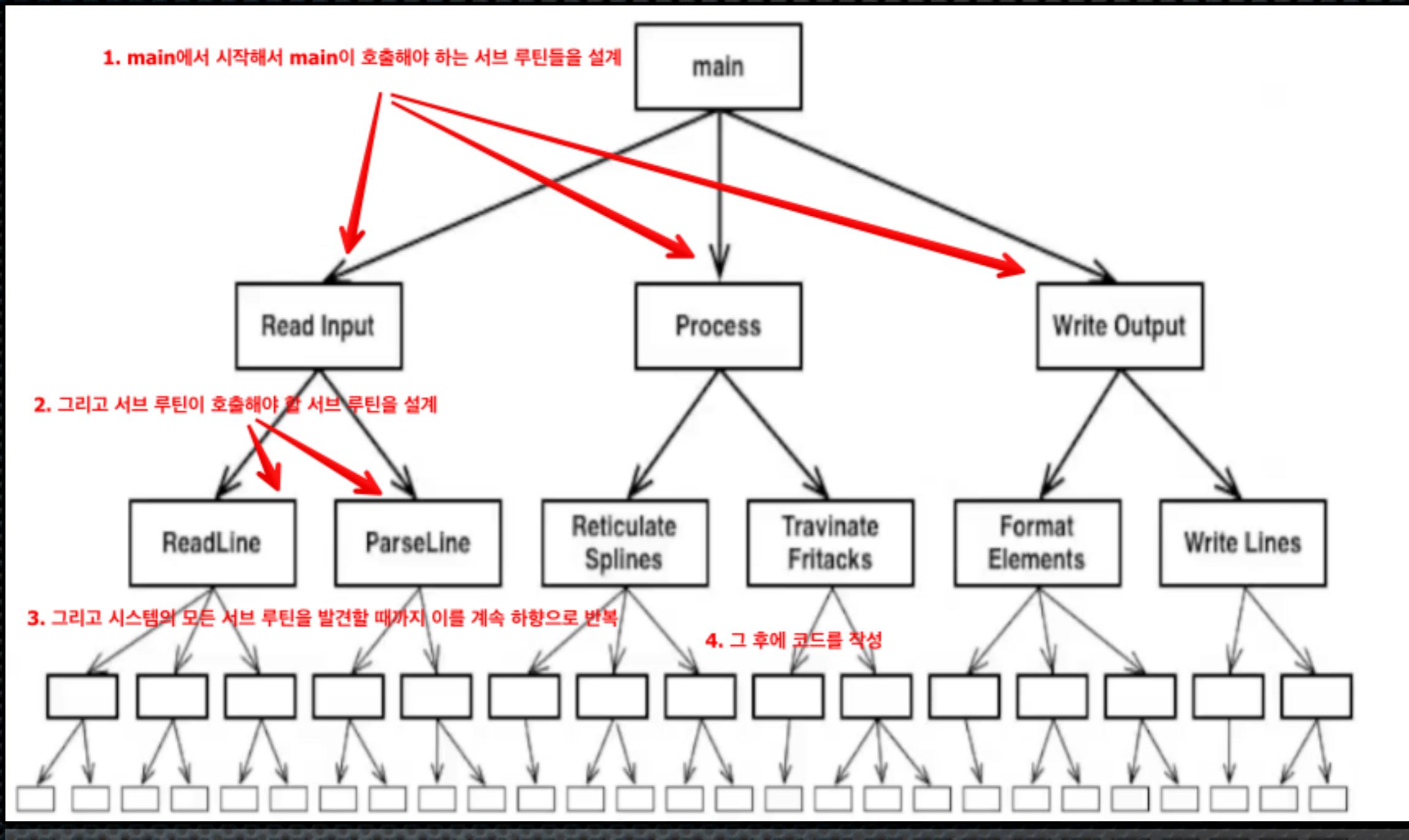


OO의 핵심

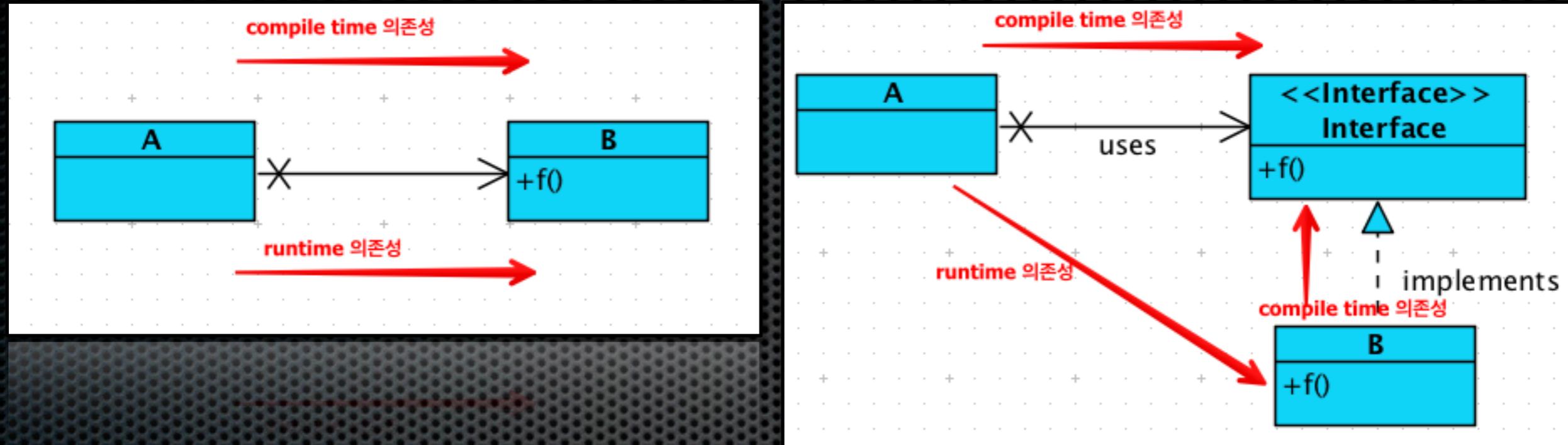
- Inheritance, Encapsulation, Polymorphism
 - 핵심이 아니라 OO의 메커니즘
- IoC를 통해 상위 레벨의 모듈을 하위레벨의 모듈로 부터 보호하는 것
 - OCP를 통해 새로운 요구사항을 반영할 수 있음
- OO design은 dependency management

Structured Design

- Top-down 방법론
 - 소스 코드 의존성의 방향 = 런타임 의존성의 방향

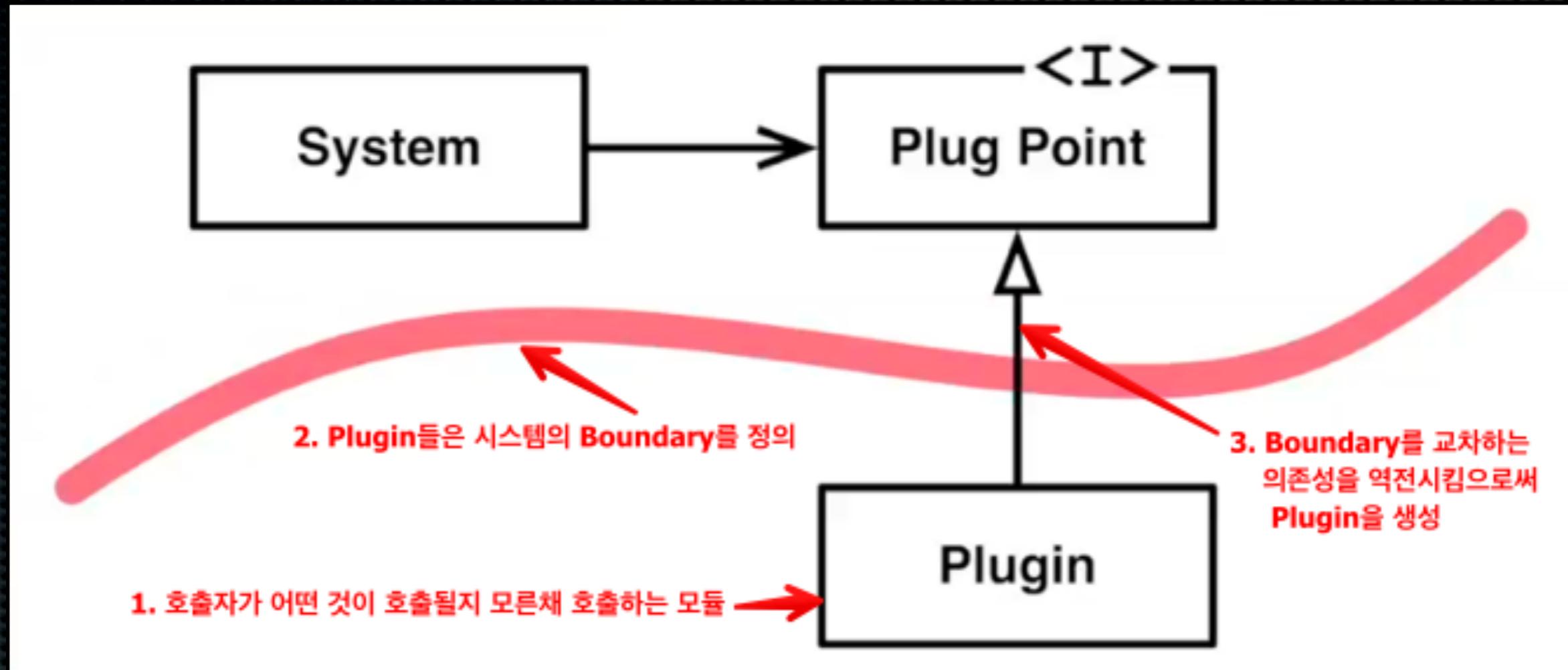


Dependency Inversion



- 절차
 - A와 B 사이에 polymorphic interface를 삽입
 - A는 uses interface
 - B는 implements the interface

Plugins



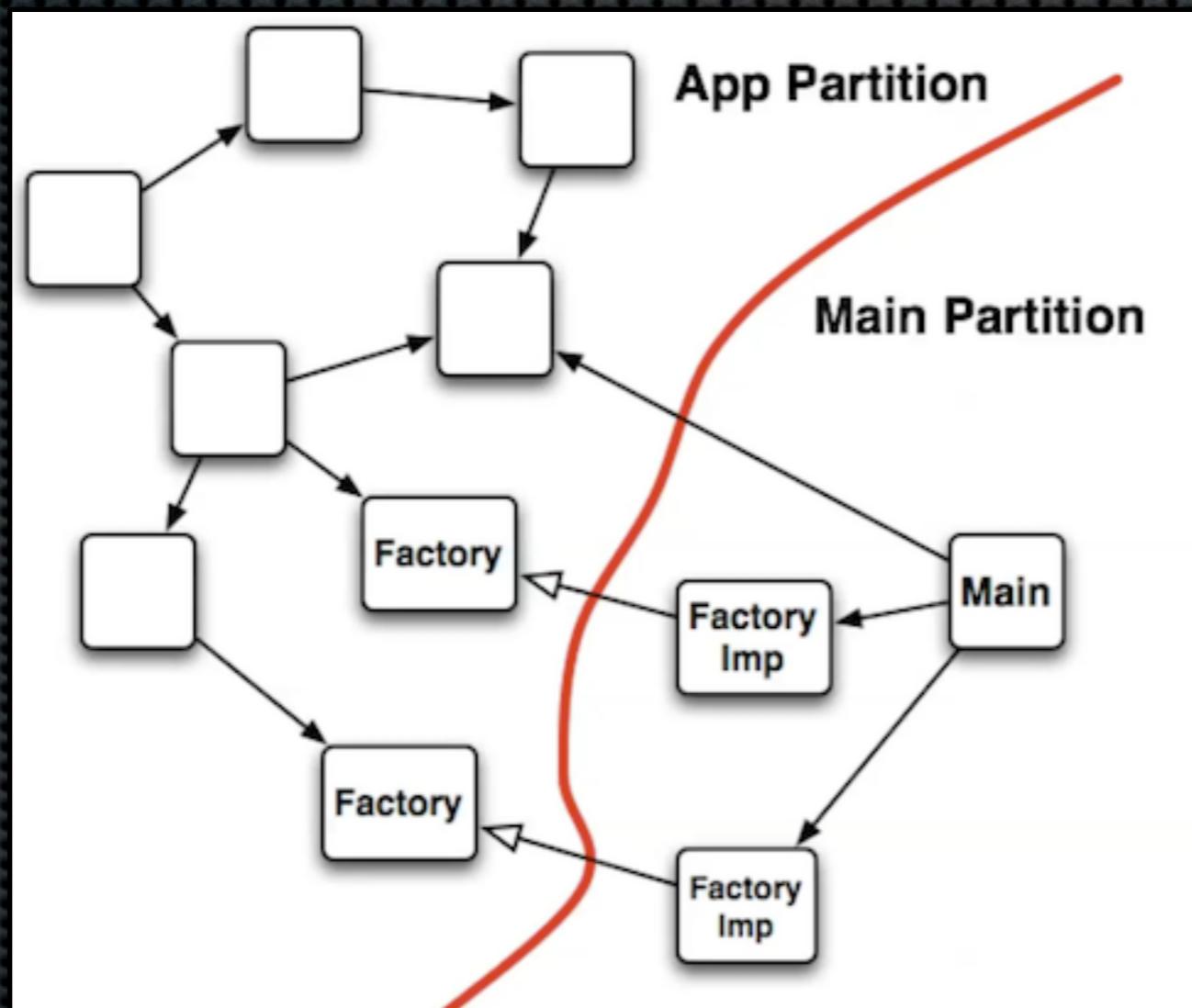
- plugin
 - Boundary를 Plugin Interface로 다룬다.
 - 의존성 역전이 SW 모듈 간의 경계를 만드는 수단

Plugins

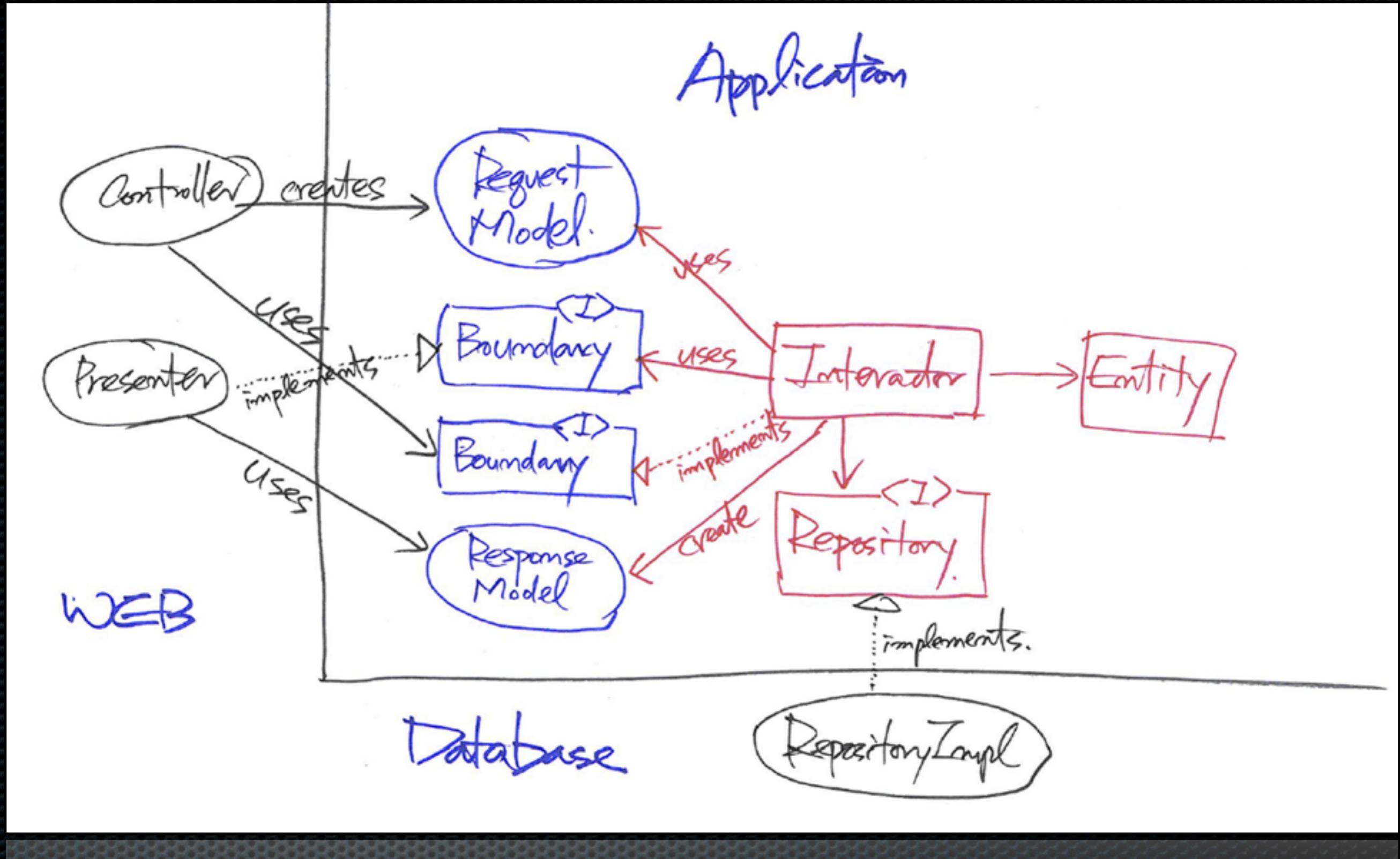
- Boundary는 우리가 plugin을 만드는 방법과 같다.
- Boundary를 만들고자할 때 마다 어떤 의존성을 역전 시킬지 고심
- Boundary를 교차하는 의존성의 방향이 같도록
- Boundary로 시스템을 나누고 Boundary를 교차하는 의존성을 역전

Plugins

- Main should be a plugin to the rest of the applications.



Architectural Implications



A Reusable Framework

- 객체지향이 과거 지향했던 재사용이 통하지 않는 이유
 - 2인이 1년간 개발한 10만 라인 중 88%의 코드가 reusable framework에 있었다.
 - 고객은 만족했고, 추후 3개의 어플리케이션을 더 계약
 - 하지만 reusable framework가 다른 3개의 어플리케이션에 적합하지 않다는 것을 알게 되었다.
 - 그래서 재사용 못하고 바닥부터 다시 만들었다.
 - 고객에게 진실을 말할 수 밖에 없었다.
 - 7만7천 라인의 코드가 reusable하지 않다고.

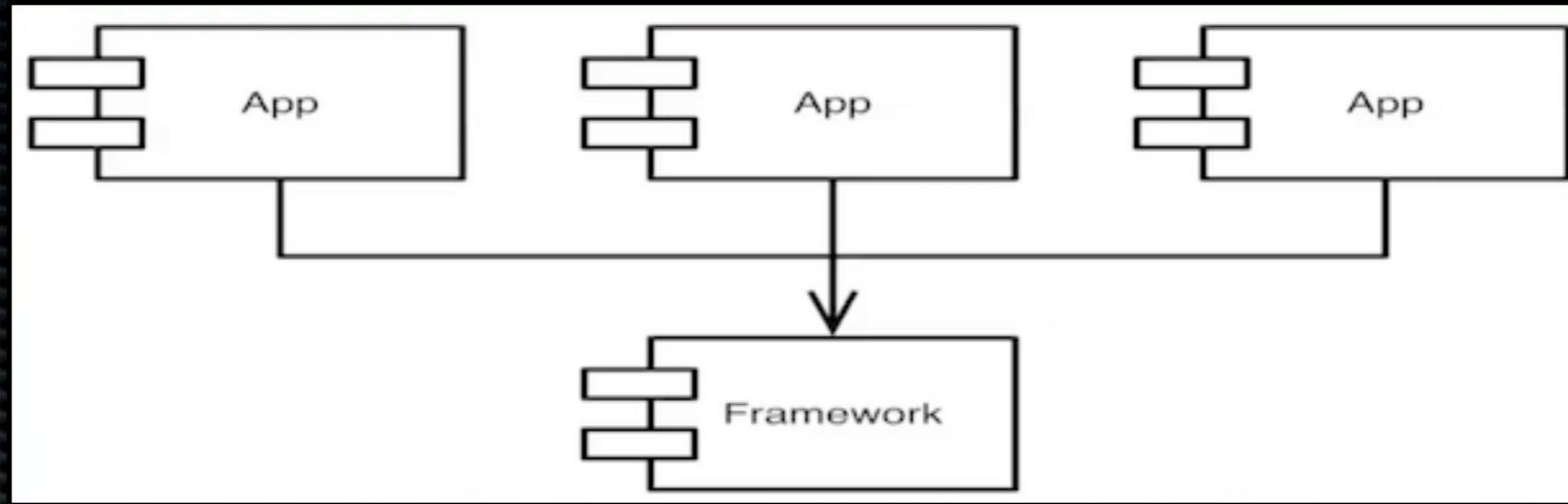
A Reusable Framework

- 객체지향이 과거 지향했던 재사용이 통하지 않는 이유
 - 하지만 시간과 비용을 고려했을 때 reusable한 코드가 있어야만 했다.
 - 그래서 고객은 첫 어플리케이션에서 만든 reusable framework을 버리고 바닥부터 만드는 대신 다시 한번 재사용 가능성을 검토하길 원했다.
 - 그래서 다시 검토했다.
 - 이번엔 3개의 어플리케이션을 병렬로 개발했다.
 - 그리고 이후 2 man years 동안 다시 7만7천 라인의 reusable framework를 만들었다.

A Reusable Framework

- 객체지향이 과거 지향했던 재사용이 통하지 않는 이유
 - 3개의 어플리케이션에서 이 framework는 재사용되었다.
 - 이후에는 계속해서 시간과 비용을 줄이면서 개발할 수 있었다.
 - reusable framework를 만드는 일은 매우 어렵다.
 - 2개 이상의 어플리케이션을 병렬로 개발하지 않는다면 거의 실패할 것이다.
 - 재사용 가능한 코드
 - 만드는 노력: 3배
 - 유지하는 노력: 10배

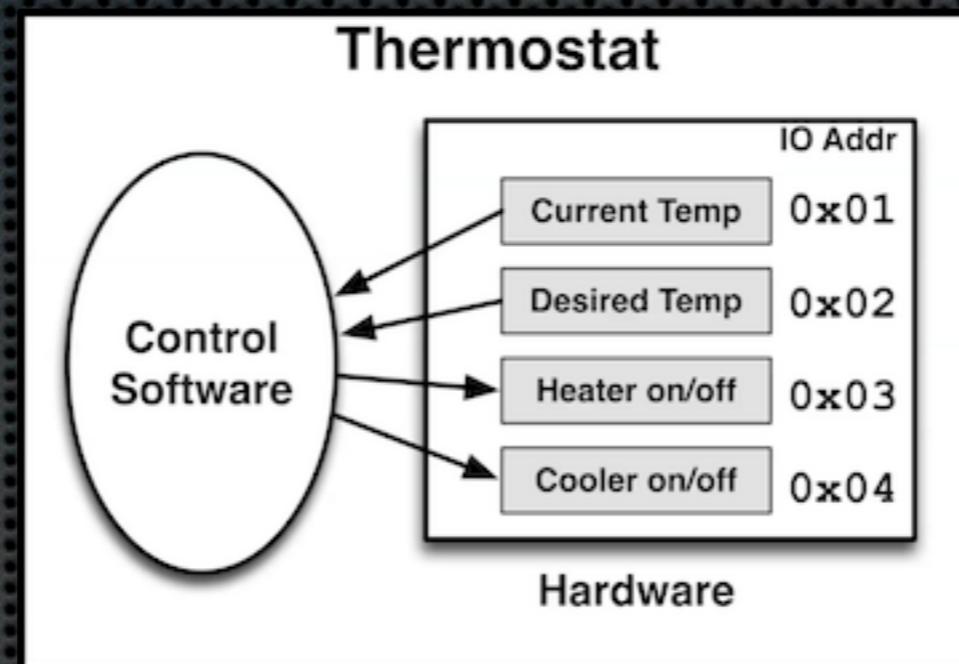
The Inversion



- 주목할 점
 - App이 FW에 의존성을 갖는다.
 - 하지만 FW는 App에 의존성을 갖지 않는다.
 - FW의 High Level Policy는 App의 Low Level Detail에 대한 의존성이 없다.
 - 일반적인 Structured Design(High level Module이 Low Level Module에 의존성을 갖는)과는 반대

The Furnace Example

- 벽에 있는 Thermostat(온도조절장치)을 제어하는 SW를 개발해야 한다고 가정



- 2개의 input과 2개의 output을 갖는 장치
 - 2개의 input: 현재 온도/희망 온도를 반환
 - 2개의 output: heater/cooler를 켜고/끄기 위한 boolean 값을 수신

The Furnace Example

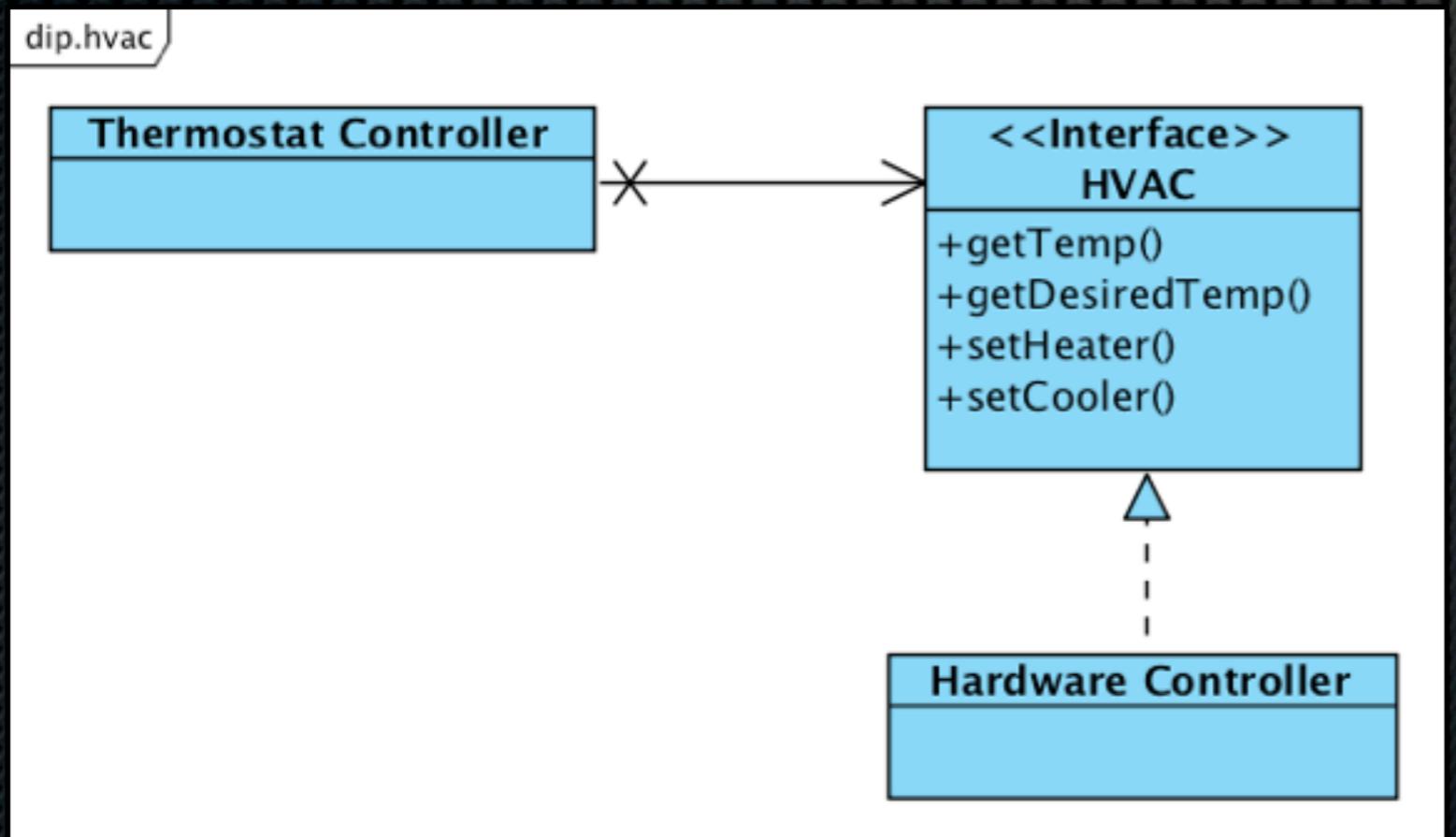
```
int t = in(0x02);
out(0x03, true);
```

```
void regulate() {
    int goal_t, t;
    while(1) {
        sleep(ONE_MINUTE);
        goal_t = in(0x02);
        t = in(0x01);
        if(t < goal_t) {
            out(0x03, true);
            out(0x04, false);
        }
        else if(t > goal_t) {
            out(0x03, false);
            out(0x04, true);
        }
        else {
            out(0x03, false);
            out(0x04, false);
        }
    }
}
```

- DIP 위반
 - 알고리즘이 low level detail에 의존
 - High level control algorithm은 다른 디바이스와는 사용될 수 없다.

The Furnace Example

- DIP에 순응하도록 변경
 - HVAC이라는 인터페이스를 생성
 - 필요한 함수들을 정의한다.



```
void regulate(Hvac hvac) {
    int goal_t, t;
    while(true) {
        Thread.sleep(ONE_MINUTE);
        goal_t = hvac.getDesiredTemp();
        t = hvac.getTemp();
        if(t < goal_t) {
            hvac.setHeater(true);
            hvac.setCooler(false);
        } else if(t > goal_t) {
            hvac.setHeater(false);
            hvac.setCooler(true);
        } else {
            hvac.setHeater(false);
            hvac.setCooler(false);
        }
    }
}
```