

# SOLID Foundation

Daum Corp.  
백명석

# 발표목차

- The Source Code is the Design
- Design Smells
  - Rigidity / Fragility / Immobility
- Needless Complexity
- Code Rot
  - Procedural
- What is OOP ?

# The Source Code is the Design

- 1992, “What is Software Design?” by Jack W. Reeves
  - Q: What do Engineers produce?
  - A: Engineers produce documents.
- 소프트웨어 공학의 결과물은?
  - 소스코드? 실행하는 바이너리 코드
  - 소스코드는 document이다.
  - The source code is the design.

# The Source Code is the Design

- 건물, 회로, 기계
  - 저렴한 설계 비용
  - 비싼 수정 비용
- 소프트웨어
  - 저렴한 구축 비용(컴파일, 빌드)
  - 비싼 설계 비용(소스 코드 작성)

# Design Smells

- Rigidity
- Fragility
- Immobility

# Design Smells - Rigidity

- 정의
  - 시스템의 의존성으로 인해 변경하기 어려워지는 것
- Rigid하게 하는 원인
  - 많은 시간이 소요되는 테스트와 빌드
  - 전체 리빌드를 유발하는 아주 작은 변화
- 테스트와 리빌드 시간을 줄이면
  - Rigidity를 줄이고
  - 수정이 용이해짐

# Design Smells - Fragility

- 한 모듈의 수정이 다른 모듈에 영향을 미칠 때
- ex. 자동차를 SW로 제어
  - 라디오버튼을 수정하는데 자동창문이 영향을 받는 경우
- 해결책
  - 모듈간의 의존성을 제거하는 것

# Design Smells - Immobility

- 모듈이 쉽게 추출 되지 않고 재사용 되지 않는 경우
- ex. 로그인 모듈이 특정 DB의 schema를 사용하고, 특정 UI skin을 사용하는 경우
  - 이 로그인 모듈은 다른 시스템에서 재사용하지 못할 것
  - 이 로그인 모듈은 immobile하다.
- 해결책
  - DB, UI, Framework등과 결합도를 낮추는 것

# Viscosity(점성)

- 빌드/테스트 같은 필수 오퍼레이션들이 오래 걸려 수행이 어렵다면 그 시스템은 역겨운(disgust) 것
  - 체크인, 체크아웃, 머지등은 비용이 크고 역겨움
  - 여러 레이어를 가로질러 의존성을 갖는 것은 역겨움
- 항상 같은 역겨움의 원인
  - Irresponsible tolerance(무책임한 용인)
- 강하게 coupling된 시스템은 테스트, 빌드, 수정을 어렵게 함.
- 해결책
  - dependency는 유지한채로 decoupling하는것

# Needless Complexity

- SW 설계의 이슈
  - 미래를 어떻게 다루나
- 현재 요구사항만 구현 vs 미래를 예측하여 구현
- 앞으로의 확장을 고려하여 설계
  - 불필요하게 시스템은 복잡해짐
  - 개발자는 현재를 제어할 수 없음
- 불필요한 복잡함: 강한 커플링
- 해결방법
  - 현재 요구 사항에 집중

# Code Rot

- 키보드 입력을 받아 프린터로 출력하는 프로젝트
  - 월요일 아침. 보스. 6개월. 3주
  - 다이어그램
  - 화요일.

```
public void copy() {  
    int c;  
    while ((c = readKeyboard()) != EOF) {  
        writePrinter(c);  
    }  
}
```

# Code Rot

- 화요일
  - 컴파일 하려는데 일이 생긴다.
- 수요일
  - 또 현장에서 큰 버그. 하루가 또 간다.
- 목요일
  - 드디어 테스트 하고, 처음으로 실행
  - 하루종일 또 미팅이다.
- 금요일
  - 회의도, 버그도, 어떠한 방해도 없다.
  - 소스코드 시스템에서 코드를 받는데 하루가 간다.
- 아직 2주가 있지만...

# Code Rot - Version 2

- 2가지 입력 도구
  - 키보드, 종이 테입 리더기
- 3주

```
private boolean GptFlag = false;
// Remember to clear

public void copy() {
    int c;
    while ((c = (GptFlag ? readPt() : readKeyboard())) != EOF) {
        writePrinter(c);
    }
}
```

# Code Rot - Version 3

- 2가지 출력 도구
  - 프린터, 종이 테입 펀처
- 3주

```
private boolean GptFlag = false;
private boolean GpunchFlag = false;
// Remember to clear

public void copy() {
    int c;
    while ((c = (GptFlag ? readPt() : readKeyboard())) != EOF) {
        if(GpunchFlag)
            writePunch(c);
        else
            writePrinter(c);
    }
}
```

# Code Rot - Summary

- 시간이 흐름에 따라
  - 점점 더 많아지는 입출력 장치
  - 점점 더 커지는 코드
  - 점점 더 지저분해지는 코드
  - 변경/분석이 어려워짐
  - 가독성/유연성 저하

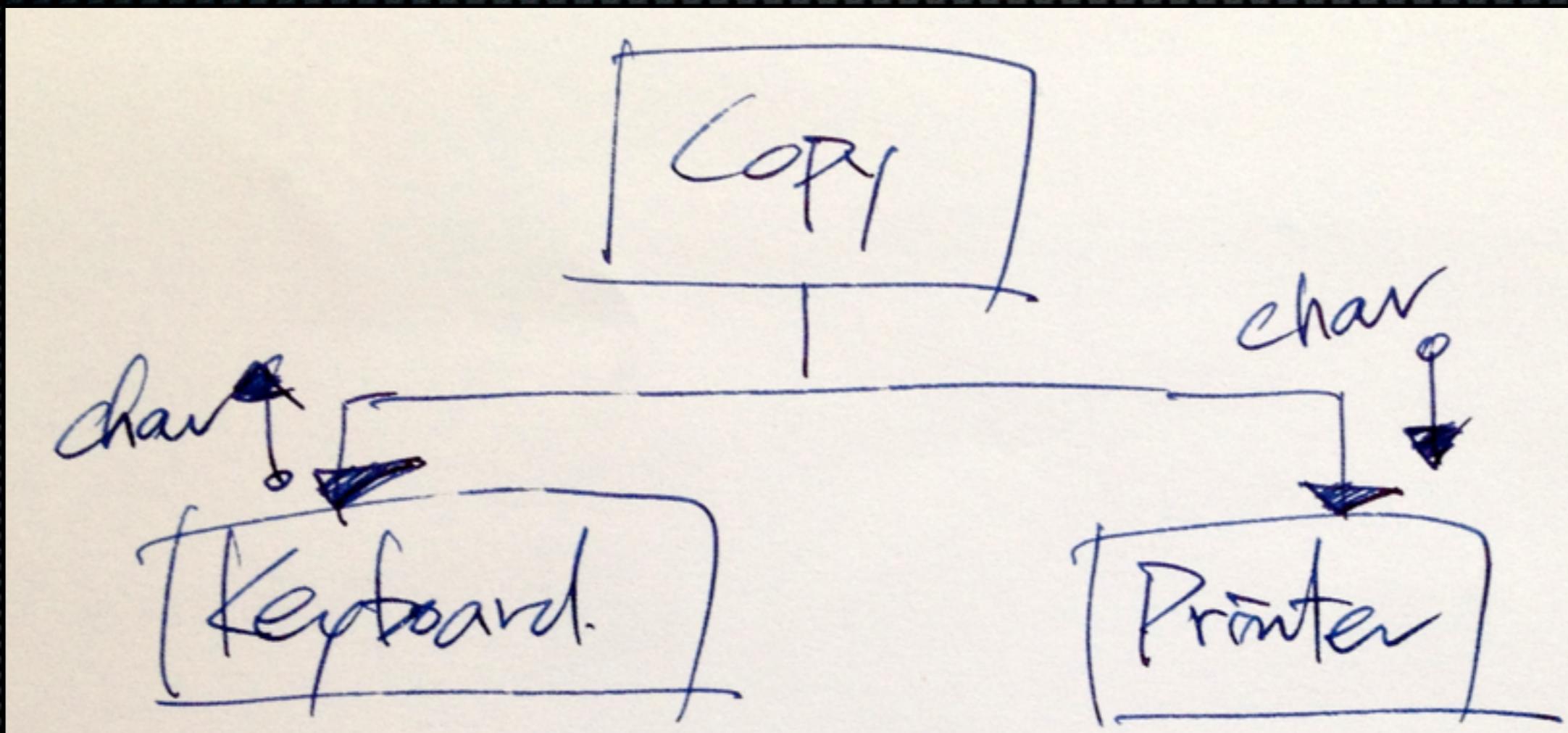
# What is OOP ?

```
public void copy() {  
    int c;  
    while ((c = getChar()) != EOF)  
        putChar(c);  
}
```

- stdio로 입출력
  - 다른 장치로 redirect 가능
- “테이프 리더로부터 입력 받을 수 있게 해야해”
  - “3주 걸립니다.” or
  - “이미 테이프 리더로부터 입력을 받을 수 있도록 되어 있습니다.”
- 2가지 효과
  - 장치 추가에 따른 코드 품질 저하 제거
  - 재컴파일 하지 않아도 된다.

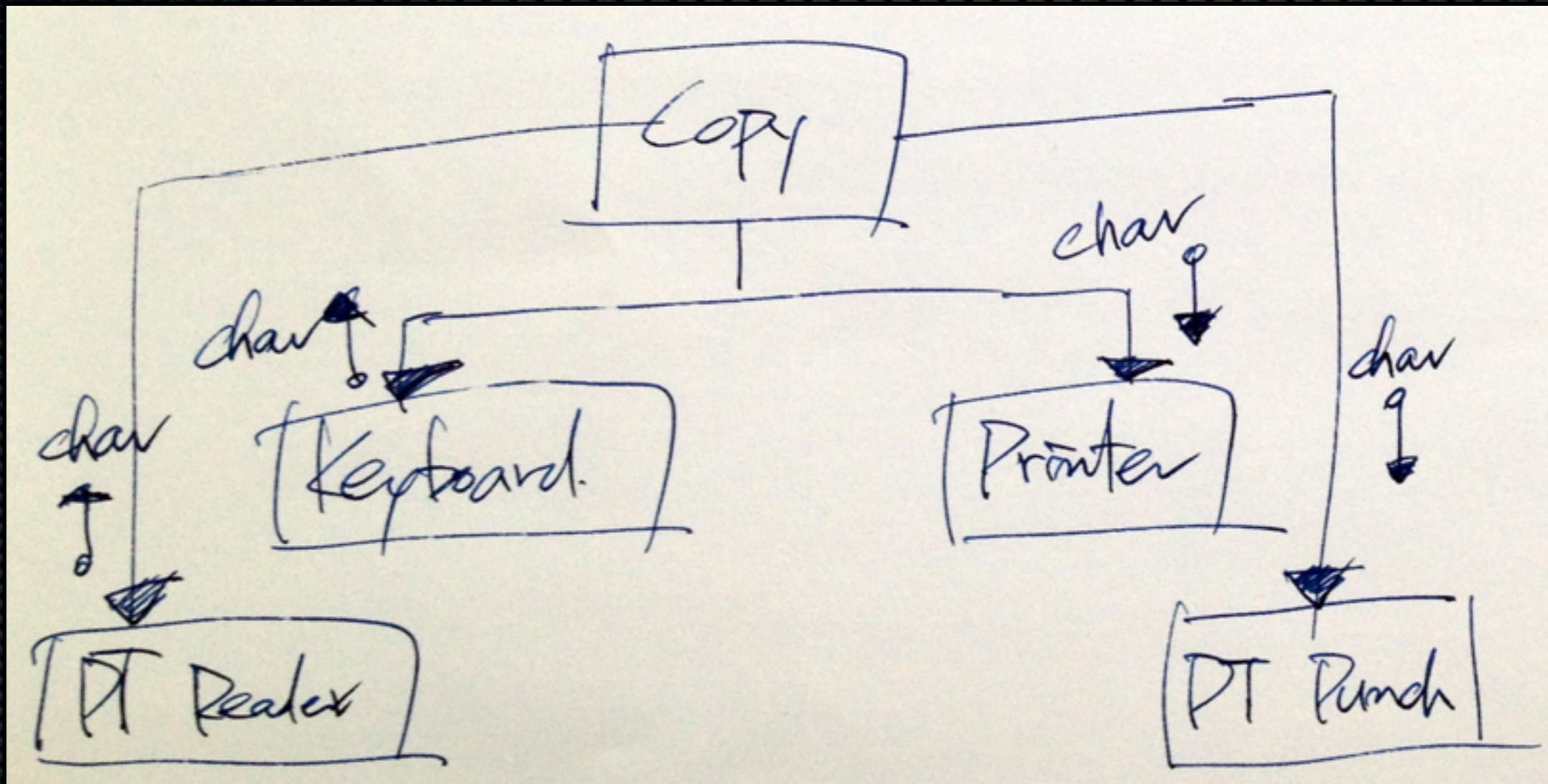
# Procedural

- High Level depends on Low Level



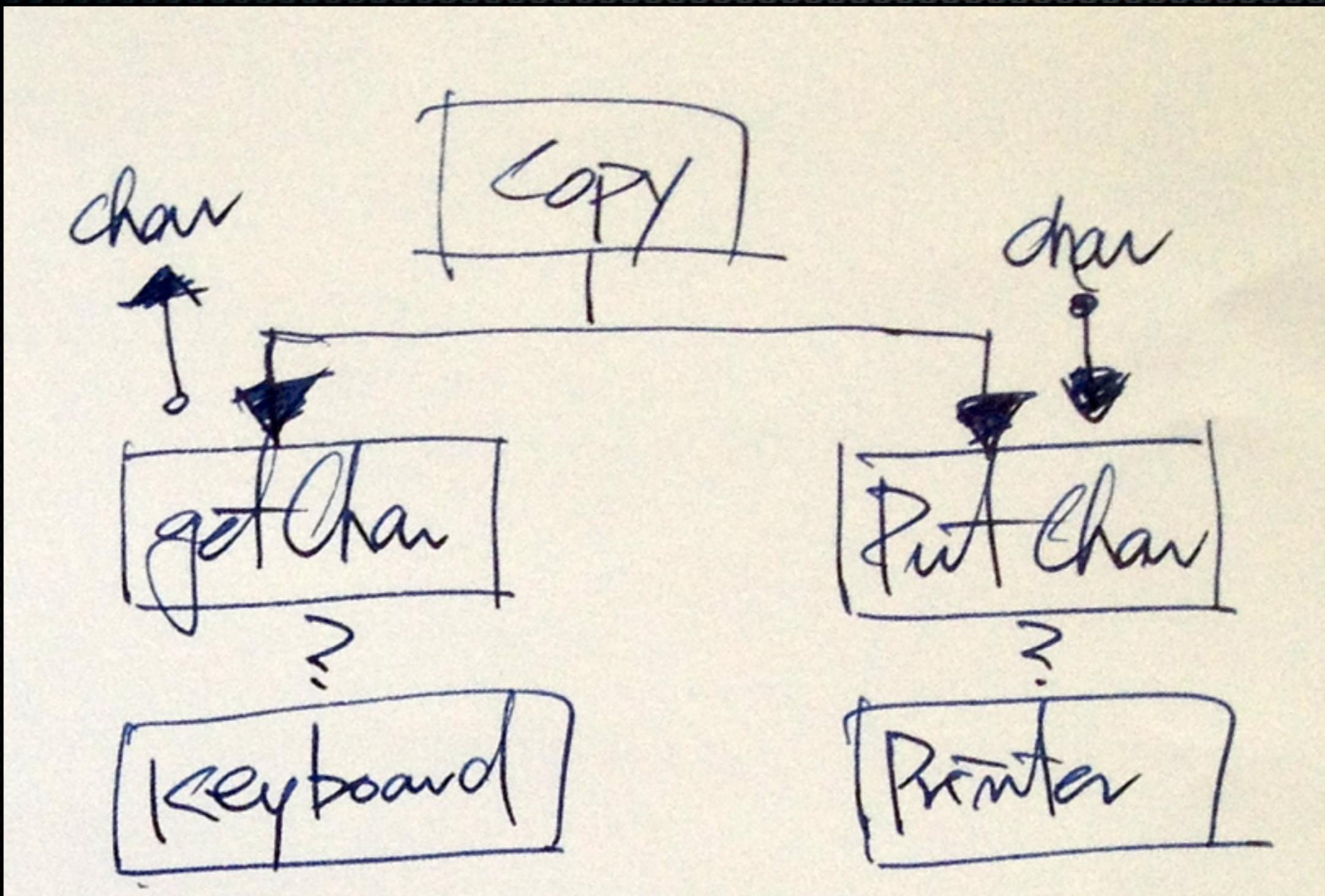
# Procedural

- 장치가 증가하면 - fan-out



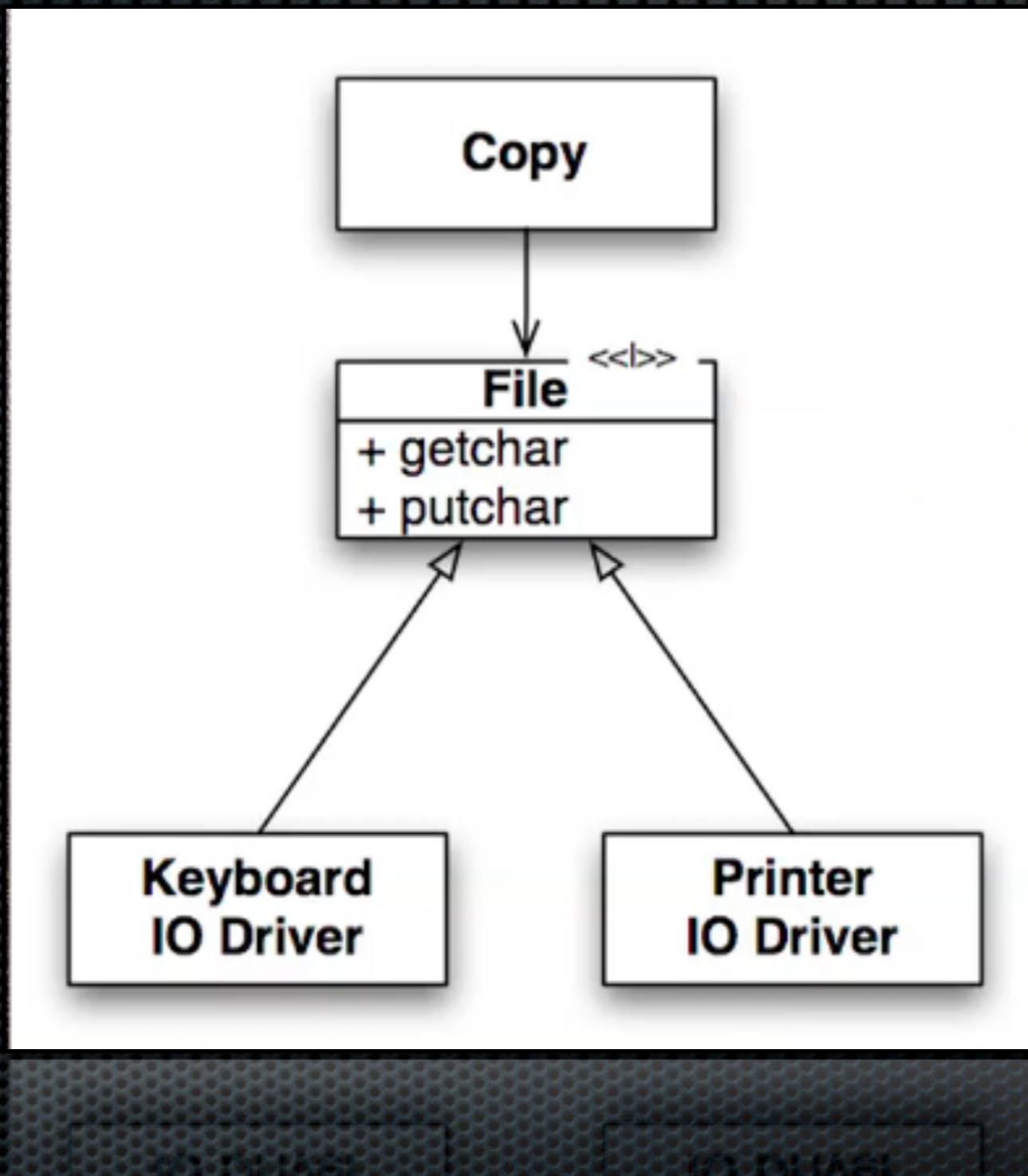
# OOP

- High Level don't depend on Low Level



# OOP

- Dependency Inverted



# OOP

- IoC via Reader/Writer Interface

```
public void copy() {  
    int c;  
    while ((c = getChar()) != EOF)  
        putChar(c);  
}
```

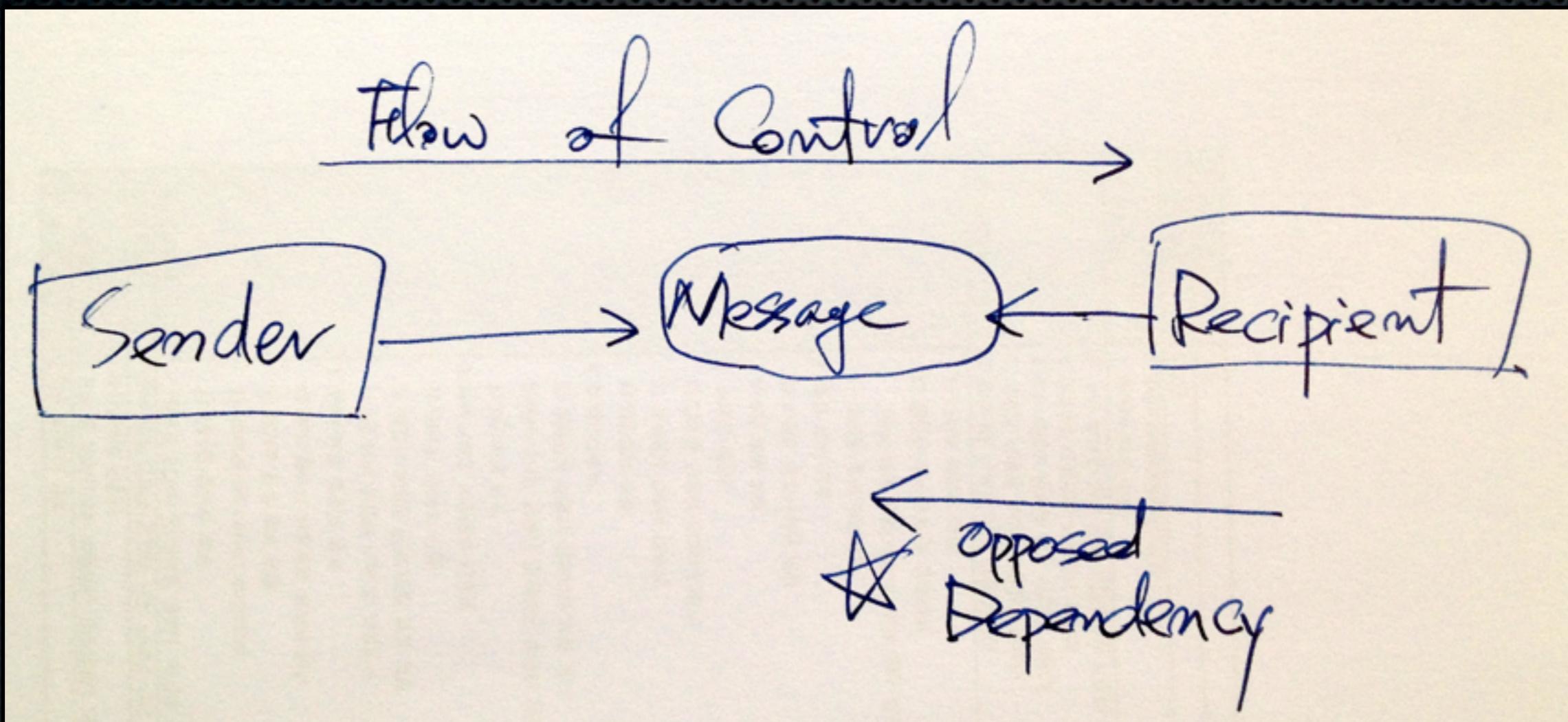
```
public interface Reader {...}  
  
public interface Writer {...}  
  
public void copy(Reader reader, Writer writer) {  
    int c;  
    while ((c = reader.getChar()) != EOF)  
        writer.putChar(c);  
}  
  
public class Keyboard implements Reader {...}  
  
public class Printer implements Writer {...}
```

# What is OO ?

- $o.f(x) \neq f(o,x)$
- Dynamic polymorphism
- OO는 메시지를 전달하는 것
  - 어떻게 동작하는지 모르고
  - 무엇을 원하는지를 전달하는 것

# What is OO ?

- Dependency Inversion - OO의 정수



# What is OO ?

- OO는 실세계를 똑같이 모델링 하는 것
- Inheritance, Encapsulation, Polymorphism
- OO의 핵심이 아니라 메커니즘
- OO의 핵심
  - IoC를 통해 상위 레벨의 모듈을 하위 레벨의 모듈로 부터 보호하는 것
  - Inverted structure tends not to rot
- OOD
  - Dependency Management
  - Inversion of key dependencies that isolate the high level policies from low level details

# Dependency Management

- 의존성 관리에 대한 중요한 규칙 - SOLID
  - SRP: Single Responsibility Principle
  - OCP: Open Closed Principle
  - LSP: Liskov Substitution Principle
  - ISP: Interface Segregation Principle
  - DIP: Dependency Inversion Principle