# Clean Code

## 11. 시스템

알파서클 연구원
Luna

# 시스템의 제작과 사용을 분리하라

- 시작단계(init) 마저도 관심사로 생각하고 분리하며 SRP를 지켜야한다.
- 생성을 위한 모듈(main)로 모든 생성과 관련된 코드를 옮기고 그 외의 시스템(애플리케이션)은 모든 의존성이 연결되었다고 가정한다.
  - 애플리케이션은 생성에 대해 모르며 의존성도 단방향으로 흐른다.
- 생성되는 시점(조건)을 제어해야할땐 (추상)팩토리 패턴을 사용한다.

```
class SharpFactory {
    sharp: Sharp
    sharpCore: SharpCore

    constructor(specialSomething: string) {
        this.sharp = new Sharp(specialSomething)
        this.sharpCore = new SharpCore(specialSomething)
    }
}

const sharp4B = new SharpFactory("4B")
const sharpHB = new SharpFactory("HB")
```

# 시스템의 제작과 사용을 분리하라

- DI, IoC를 이용하면 생성과 사용의 책임이 완전히 분리될 수 있어 효과적이다.
  - Java의 JNDI - 의존성 풀, 룩업 방식
    - bean을 개발자가 직접 컨테이너에서 제공하는 API를 이용해 lookup시킴. 컨테이너와 종속성 존재
  - Spring, Nest.js - 의존성 주입 방식
    - 프레임워크가 알아서 자동으로 의존성 관리를 해주며 다른 컨테이너에 의존되지 않는 완전한 의존성 주입

# 확장

- 처음부터 완벽한 확장에 대해 고려하며 만들기는 매우 어렵다.
- 지금 당장의 주어진 스토리(요구사항)에 맞춰 시스템을 설계하고 확장하면 된다.
  - 애자일 방식의 핵심이자 TDD에 부합하다
- 하지만 미래를 위해 최소한의 관심사(생성, 사용)는 분리되어야 한다.

# 횡단 관심사

- 영속성(DB), 로깅, 보안(JWT 요구) 같은 공통 로직은 여러 곳에서 반복된다.
- 이러한 핵심 로직 외의 부가 기능들을 횡단 관심사(cross-cutting)라고 한다.
- 횡단 관심사를 분리해 따로 관리하는 것을 관점 지향 프로그래밍 (AOP, Aspect-Oriented Programming)이라고 한다.

```
getCategory('/category') {
  // 로그인 확인
  // 서버에 로그 출력
  // ... 비즈니스 로직
}

getContents('/contents') {
  // 로그인 확인
  // 서버에 로그 출력
  // ... 비즈니스 로직
}
```

```
@AuthGuard
@Logging
getCategory('/category') {
  // ... 비즈니스 로직
}

@AuthGuard
@Logging
getContents('/contents') {
  // ... 비즈니스 로직
}
```

# 프록시, 어노테이션, 데코레이터

- 프록시 패턴
  - 어떤 로직의 전/후처리를 위해 실제 로직 전에 실행되는 로직
  - 원래의 로직에 영향 X
  - ex) 클라이언트에서 요청이 오면 로그 띄우기
- 데코레이터 패턴
  - 어떤 로직이 시작하기 전에 특정 기능을 확장(추가)
  - 원래의 로직에 영향 O
  - ex) 클라이언트에서 온 요청 값을 수정 (쓸모없는 헤더 제거)
- 어노테이션
  - 일종의 메타데이터로 컴파일러나 IDE에게 정보를 제공해주는 기능
  - Java의 @Override (런타임에 영향 X)