SOLID 원칙과 객체지향 설계1

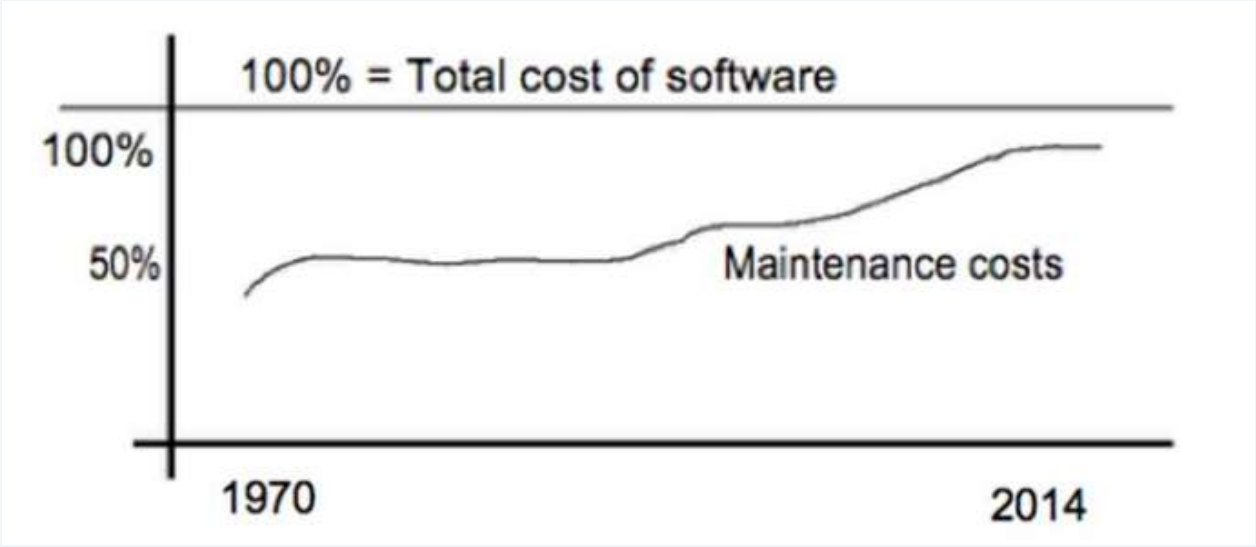
유지보수 가능한 소프트웨어 설계의 핵심 원칙

SOLID 원칙과 객체지향 설계

- 소프트웨어 설계 품질의 중요성
- SOLID 원칙을 통한 견고한 객체지향 설계
- 실무에서 적용 가능한 설계 철학

왜 설계가 중요한가?

- 소프트웨어 개발 비용의 80%는 유지보수
- **설계 품질이 프로젝트 성공을 좌우**
- **좋은 설계 = 변화에 유연하게 대응**



좋은 설계 vs 나쁜 설계

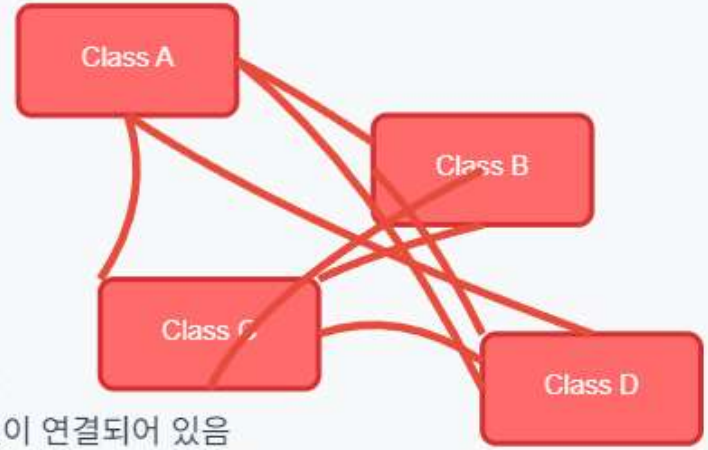
설계 품질의 특징

좋은 설계	나쁜 설계
✓ 변경에 유연함	✗ 변경이 어려움 (경직성)
✓ 이해하기 쉬움	✗ 복잡하고 이해 어려움
✓ 재사용 가능	✗ 재사용 불가능
✓ 테스트 용이	✗ 테스트 곤란

설계 품질 비교 다이어그램

좋은 설계 vs 나쁜 설계

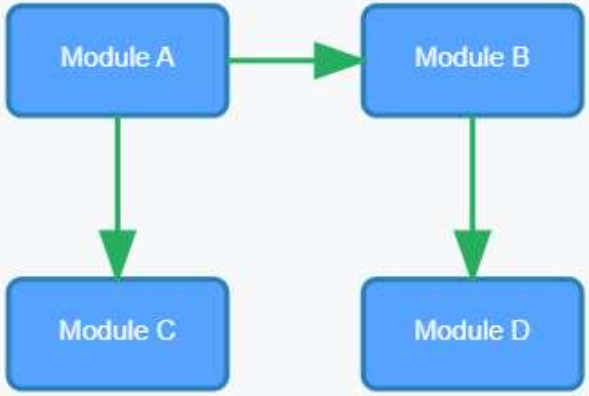
나쁜 설계 (스파게티 코드)



문제점:

- 모든 것이 연결되어 있음
- 하나 변경시 모든 곳에 영향
- 이해하기 어려움
- 테스트 불가능

좋은 설계 (모듈형)



장점:

- 명확한 책임 분리
- 독립적 변경 가능
- 이해하기 쉬움
- 테스트 용이

SOLID 원칙의 탄생

로버트 C. 마틴과 SOLID

- **1990년대 후반** 객체지향 설계 원칙 정립
- **2000년** Michael Feathers가 SOLID 약어 명명
- **Clean Code, Clean Architecture 저자**

SOLID의 목표

- 변경에 유연한 소프트웨어
- 이해하기 쉬운 코드
- 컴포넌트 재사용성

SOLID 5가지 원칙

S - Single Responsibility Principle (**단일 책임 원칙**)

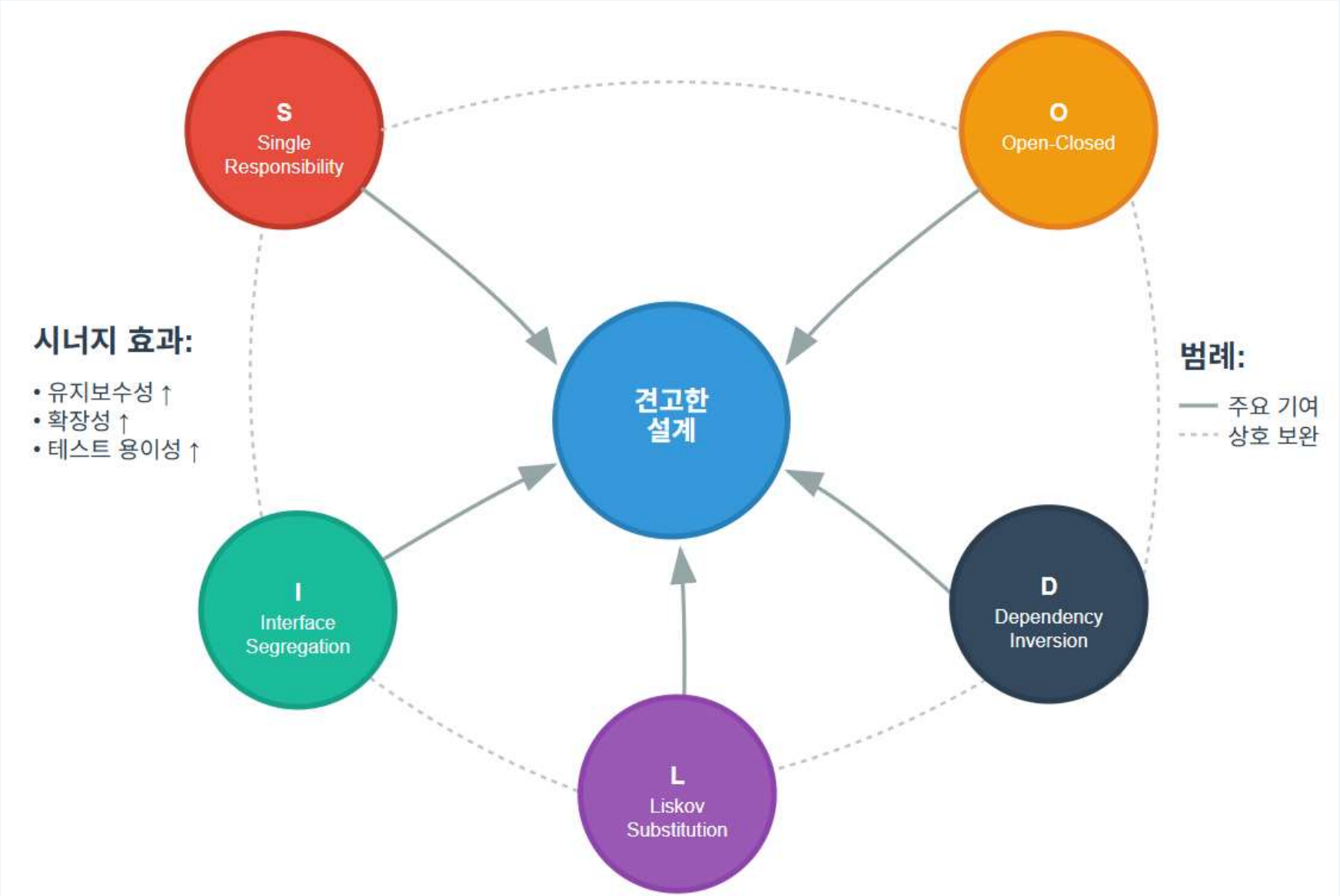
O - Open-Closed Principle (**개방-폐쇄 원칙**)

L - Liskov Substitution Principle **(리스코프 치환 원칙)**

I - Interface Segregation Principle **(인터페이스 분리 원칙)**

D - Dependency Inversion Principle **(의존성 역전 원칙)**

SOLID 원칙의 상호작용 원칙들의 시너지 효과



단일 책임 원칙 (SRP)

Single Responsibility Principle (SRP)

"클래스는 단 하나의 변경 이유만 가져야 한다"

- SOLID의 첫 번째 원칙
- 가장 이해하기 쉬우면서도 적용하기 어려운 원칙
- 좋은 객체지향 설계의 출발점

책임(Responsibility)이란?

책임이란

- 클래스가 "알고 있어야 하는 정보"
- 클래스가 "수행해야 하는 작업"
- 클래스가 "변경되는 이유"

책임 식별 질문들

- 이 클래스는 무엇을 알고 있는가?
- 이 클래스는 무엇을 하는가?
- 누가 이 클래스의 변경을 요구하는가?

Actor 관점에서의 책임

변경을 요구하는 주체 (Actor)

Actor란

- 시스템 변경을 요구하는 사용자 그룹
- 서로 다른 Actor = 서로 다른 책임

예시) 직원 관리 시스템

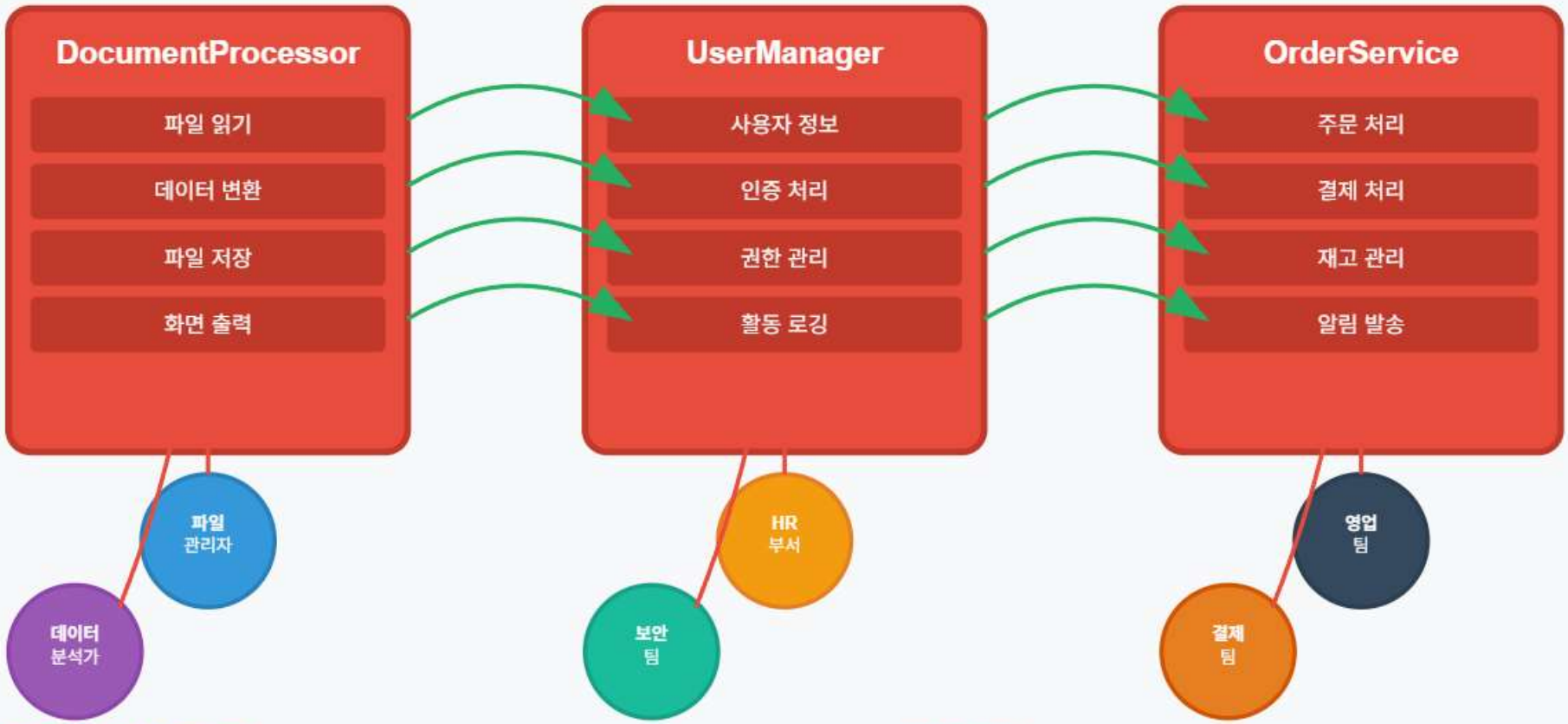
HR 부서: 직원 정보 관리

회계 부서: 급여 계산

IT 부서: 데이터 저장 방식

→ 하나의 Employee 클래스가 모든 Actor를 만족하려 하면 SRP 위반!

SRP 위반 사례 - 개념적 설명



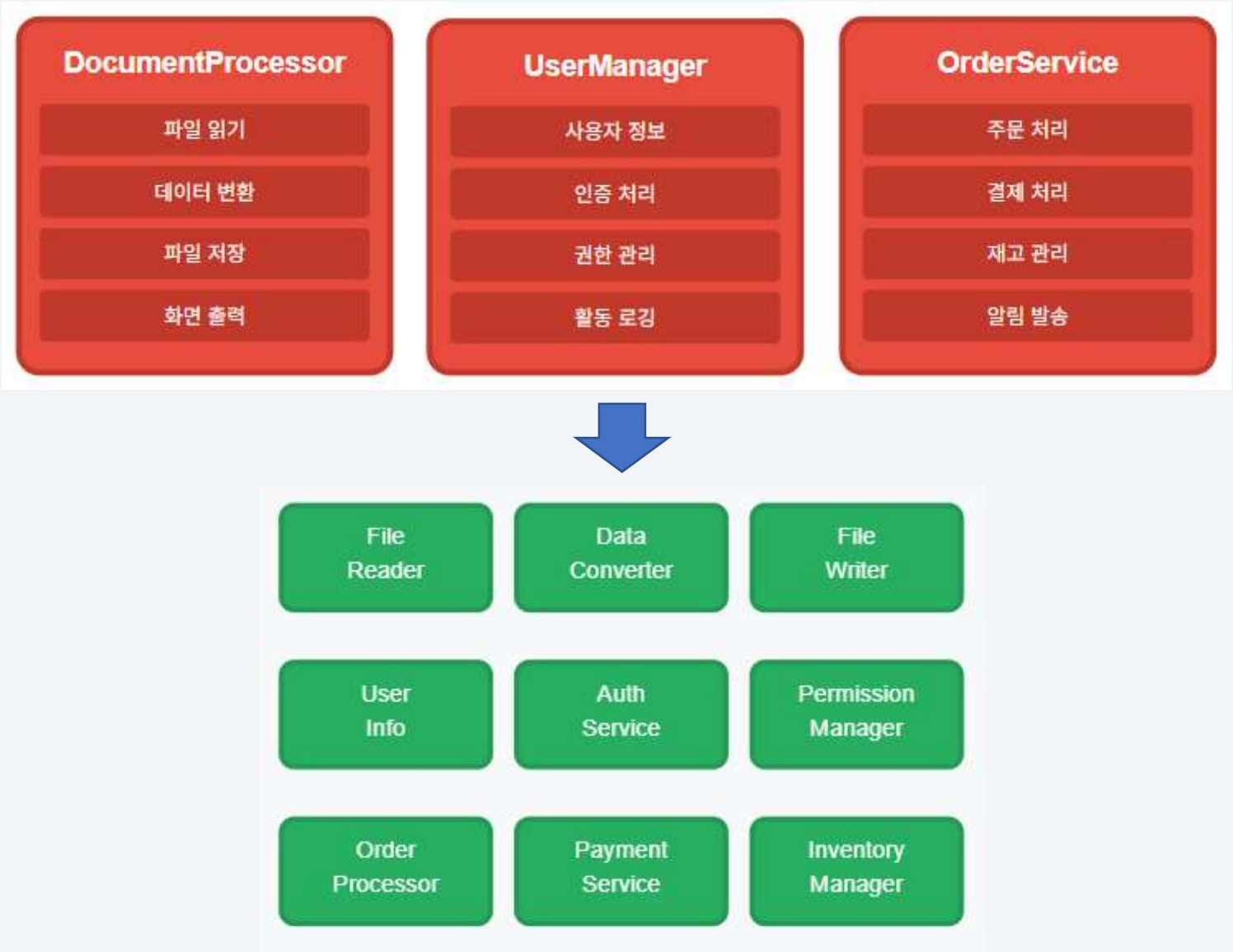
각 클래스의 문제점:

- DocumentProcessor: 4가지 다른 변경 이유
- UserManager: 4가지 다른 Actor의 요구사항
- OrderService: 비즈니스/기술적 책임 혼재

해결 방향:

- 각 책임별로 독립적인 클래스 분리
- Actor별 전용 클래스 할당
- 조합(Composition)을 통한 협력

SRP 위반 사례 - 개념적 설명



SRP 위반 코드 예시

```
public class User
{
    // 사용자 정보
    public string Name { get; set; }
    public string Email { get; set; }

    // 인증 로직
    public bool Login(string password) { /* ... */ }

    // 데이터 저장
    public void SaveToDatabase() { /* ... */ }

    // 이메일 발송
    public void SendWelcomeEmail() { /* ... */ }

    // 로깅
    public void LogActivity(string action) { /* ... */ }
}
```

변경 시나리오

- 비즈니스 팀에서 "사용자 프로필에 생년월일 필드를 추가해주세요"
- 보안 팀에서 "2단계 인증을 도입해서 로그인 로직을 바꿔주세요"
- DBA에서 "데이터베이스 스키마를 변경했으니 저장 로직을 수정해주세요"
- 마케팅 팀에서 "웰컴 이메일 템플릿을 바꿔주세요"

SRP 적용 방법론

책임 분리 전략

1. Actor별 분리
- 서로 다른 사용자 그룹의 요구사항 분리

2. 추상화 레벨별 분리
- 고수준 정책 vs 저수준 구현

3. 변경 빈도별 분리
- 자주 바뀌는 것 vs 안정적인 것

4. 데이터와 행위 분리
- 정보 저장 vs 비즈니스 로직

분리 신호들

- 클래스 이름에 "And", "Manager", "Helper" 등
- 너무 많은 import문
- 거대한 클래스 (수백 줄 이상)

SRP 위반 코드 예시

```
// 사용자 정보만 담당
public class User
{
    public string Name { get; set; }
    public string Email { get; set; }
    public DateTime CreatedAt { get; set; }
}

// 인증만 담당
public class AuthenticationService
{
    public bool Authenticate(string email, string password) { }
    public void ChangePassword(string email, string newPassword) { }
}

// 데이터 저장만 담당
public class UserRepository
{
    public void Save(User user) { }
    public User FindByEmail(string email) { }
}
```

SRP 적용의 이점

1. **높은 응집도 (High Cohesion)**
   - 관련된 기능들이 함께 모여있음

2. **낮은 결합도 (Low Coupling)**
   - 클래스 간 의존성 최소화

3. **이해하기 쉬운 코드**
   - 각 클래스의 역할이 명확함

4. **테스트 용이성**
   - 각 기능을 독립적으로 테스트 가능

5. **재사용성 향상**
   - 필요한 기능만 선택적으로 사용

6. **유지보수성 향상**
   - 변경 영향 범위가 제한적

SRP 적용 시 주의사항

과도한 분리의 위험

- 너무 많은 클래스 → 복잡성 증가
- 성능 오버헤드 가능성
- 개발 시간 증가

실용적 접근

- 변경 빈도 고려
- 팀 역량과 프로젝트 규모 고려
- 80/20 법칙 적용

END.