SOLID 원칙과 객체지향 설계2

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

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)**

개방-폐쇄 원칙(Open-Closed Principle)

"소프트웨어 개체는 확장에는 열려있어야 하지만,
수정에는 닫혀있어야 한다"

개방-폐쇄 원칙(Open-Closed Principle)

문제 상황 - 게임 개발 스토리

당신은 인디 게임 개발자입니다.

간단한 슈팅 게임
- 적 캐릭터: 좀비 하나

첫 번째 업데이트
- **새로운 적 추가 요청...**

개방-폐쇄 원칙(Open-Closed Principle)

순진한 접근

```
public class Game
{
    public void UpdateEnemies(Enemy enemy)
    {
        if (enemy.Type == "Zombie")
        {
            // 좀비는 느리게 플레이어를 향해 움직임
            enemy.MoveToPlayer(speed: 1);
            if (enemy.IsNearPlayer())
                enemy.Bite();
        }
    }
}
```

개방-폐쇄 원칙(Open-Closed Principle)

순진한 접근

```
public void UpdateEnemies(Enemy enemy)
{
    if (enemy.Type == "Zombie")
    {
        enemy.MoveToPlayer(speed: 1);
        if (enemy.IsNearPlayer())
            enemy.Bite();
    }
    else if (enemy.Type == "SkeletonArcher")  // 새로 추가!
    {
        enemy.KeepDistance(range: 5);
        if (enemy.CanShoot())
            enemy.FireArrow();
    }
}
```

개방-폐쇄 원칙(Open-Closed Principle)

스파게티 코드의 탄생

```
public void UpdateEnemies(Enemy enemy)
{
    if (enemy.Type == "Zombie") { ... }
    else if (enemy.Type == "SkeletonArcher") { ... }
    else if (enemy.Type == "Wizard") { ... }
    else if (enemy.Type == "Dragon") { ... }
    else if (enemy.Type == "Goblin") { ... }
    else if (enemy.Type == "DarkKnight") { ... }
    else if (enemy.Type == "Necromancer") { ... }
    // ... 그리고 20개 더...
}

// 다른 메서드들도 마찬가지...
public void DrawEnemies(Enemy enemy) { /* 똑같은 if-else 지옥 */ }
public void PlayEnemySound(Enemy enemy) { /* 또 if-else... */ }
```

개방-폐쇄 원칙(Open-Closed Principle)

스파게티 코드의 탄생

```
public void UpdateEnemies(Enemy enemy)
{
    if (enemy.Type == "Zombie") { ... }
    else if (enemy.Type == "SkeletonArcher") { ... }
    else if (enemy.Type == "Wizard") { ... }
    else if (enemy.Type == "Dragon") { ... }
    else if (enemy.Type == "Goblin") { ... }
    else if (enemy.Type == "DarkKnight") { ... }
    else if (enemy.Type == "Necromancer") { ... }
    // ... 그리고 20개 더...
}

// 다른 메서드들도 마찬가지...
public void DrawEnemies(Enemy enemy) { /* 똑같은 if-else 지옥 */ }
public void PlayEnemySound(Enemy enemy) { /* 또 if-else... */ }
```

개방-폐쇄 원칙(Open-Closed Principle)

OCP 위반의 실제 문제점

Bertrand Meyer - OCP의 탄생

1988년, Bertrand Meyer

"Software entities should be open for extension,
but closed for modification"

상속을 통한 확장을 고안한 **최초의 아이디어**

Bertrand Meyer - OCP의 탄생

Meyer의 상속 기반 OCP

문제점

1. 깊은 상속 계층
2. 취약한 기반 클래스 문제
3. 다중 상속의 복잡성

Robert C. Martin의 혁신

2000년대, Robert C. Martin (Uncle Bob)

"OCP는 객체지향 설계의 심장이다"

핵심 - **추상화(Abstraction)를 통한 확장**

```
public interface IEnemy
{
    void Update();
    void Attack();
    void TakeDamage(int damage);
}
```

OCP의 마법

"수정에는 닫혀있고, 확장에는 열려있다."

Before

```
public class Game
{
    public void UpdateEnemies(List<Enemy> enemies)
    {
public class Slime : IEnemy
{
    public void Update()
    {
        // 슬라임만의 끈적끈적한 움직임
    }
}
            }
        }
    }
}
```

After

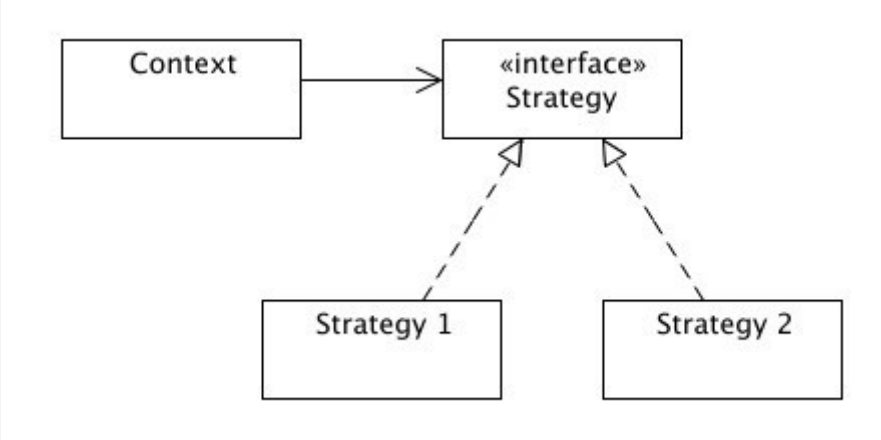
```
// ✅ OCP 준수 - 확장에 열리고 수정에 닫힌 코드
public interface IEnemy
{
    void Update();
    void Draw();
    void PlaySound();
}

public class Game
{
    public void UpdateEnemies(List<IEnemy> enemies)
    {
        foreach (var enemy in enemies)
        {
            enemy.Update();  // 다형성의 마법!
        }
    }
}
```

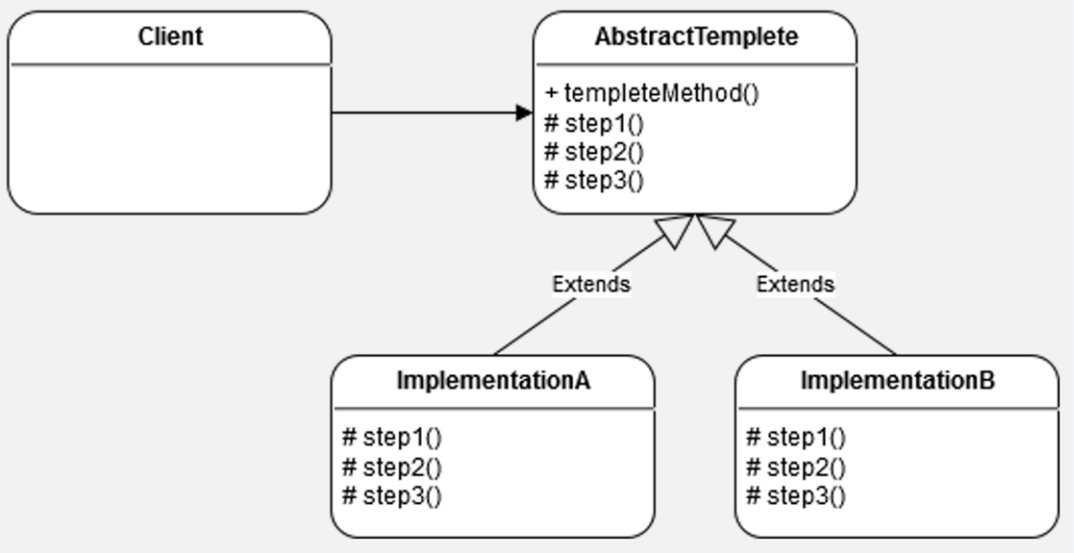
OCP 위반 vs OCP 준수

OCP를 구현한 대표적인 디자인 패턴

전략패턴
(Strategy Pattern)



템플릿메서드 패턴
(Template Method Pattern)



OCP 적용의 균형점

"모든 것을 추상화하면 안 된다"

과도한 추상화의 위험

- 복잡성 증가
- 성능 오버헤드
- 개발 시간 증가

YAGNI 원칙을 기억하세요.

"You Aren't Gonna Need It"

OCP 적용의 균형점

이런 경우는 추상화하지 마세요!

- 도메인의 핵심 로직이 아닌 부분
- 표준 라이브러리나 프레임워크의 기능
- 성능이 극도로 중요한 부분
- 팀원들이 이해하기 어려워하는 부분

YAGNI 원칙을 기억하세요.

"You Aren't Gonna Need It"

과도한 추상화: 단순한 로그인의 비극

실제로 필요했던 것

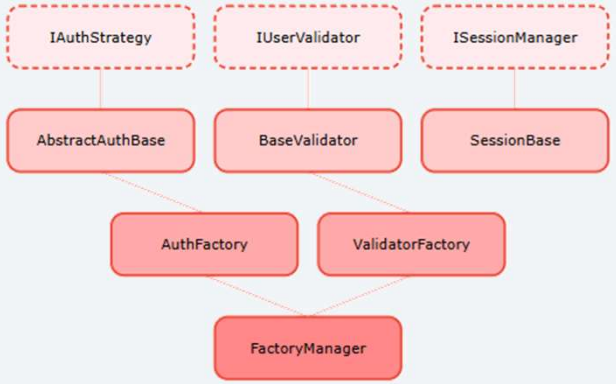
LoginService

validateUser()
createSession()

2개 클래스, 50줄

개발 시간: 3일

실제로 만든 것



... 그리고 15개 더

20개 클래스, 2000줄

개발 시간: 2개월

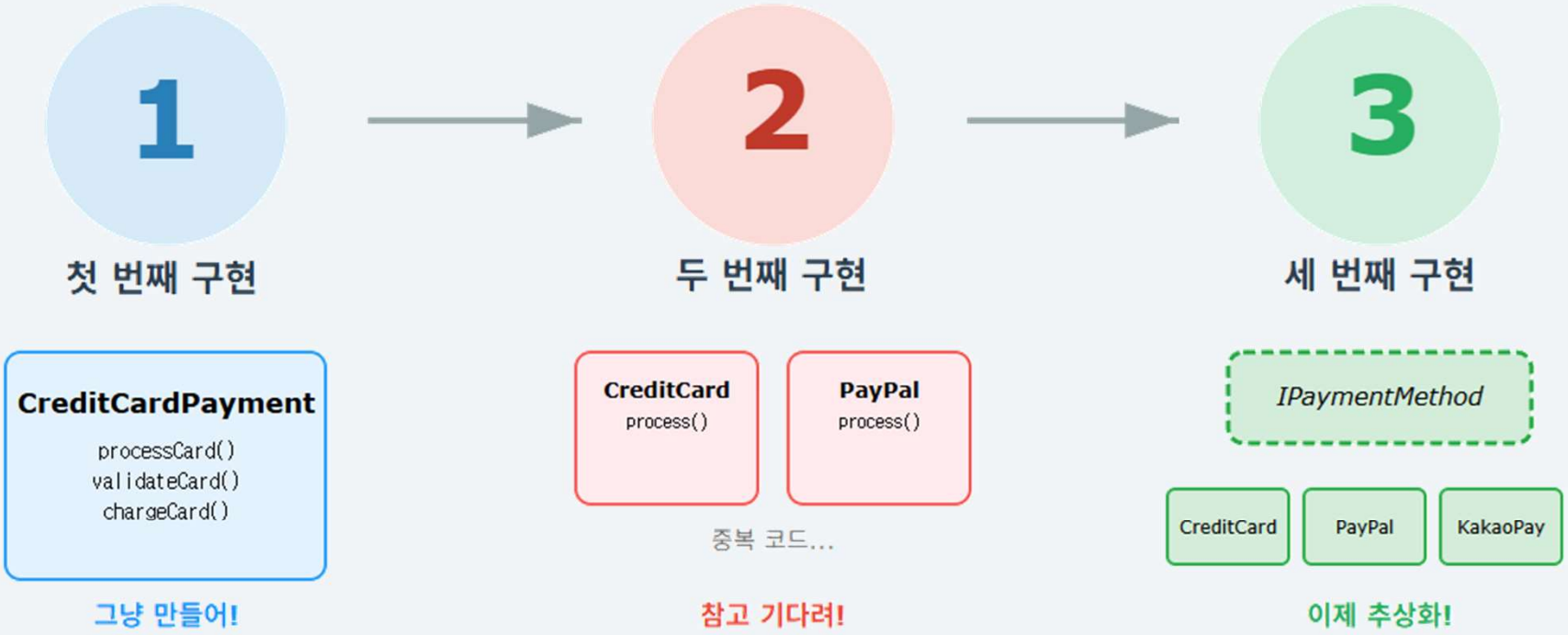
"With great power comes great responsibility"
- 스파이더맨의 벤 삼촌 (그리고 모든 시니어 개발자)

3개월 후...

간단한 버전	복잡한 버전
✅ 정상 작동 중	❌ 아직 개발 중
✅ 버그 2개 수정	❌ "완벽한 구조" 추구
✅ 기능 3개 추가	❌ OAuth 지원 (안 씀)
✅ 사용자 10만명	❌ 블록체인 준비 (왜?)
✅ 투자 유치 성공	❌ 회사 망함

💡 **미래를 예측하려 하지 마세요. 현재에 집중하세요.**

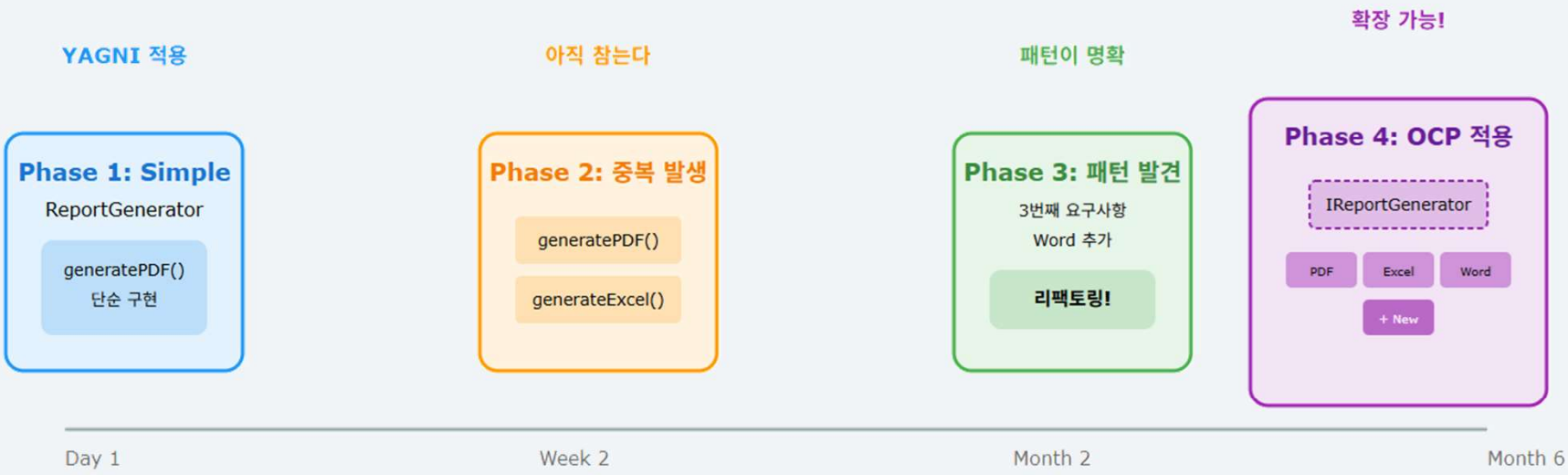
Rule of Three
추상화의 타이밍



"한 번은 우연, 두 번은 징조, 세 번은 패턴"

진짜 패턴이 보일 때까지 기다리세요

점진적 설계: OCP와 YAGNI의 균형



점진적 설계의 지혜

시작은 단순하게, 필요할 때 진화시키고, 패턴이 보일 때 추상화하라

"Make it work → Make it right → Make it fast" - Kent Beck

YAGNI 원칙을 기억하세요.

"You Aren't Gonna Need It"

XP(Extreme Programming)의 핵심 원칙

YAGNI ≠ 대충 짜기

OCP 적용 결정 체크리스트

- [x] 이미 한 번 이상 변경되었는가?
- [x] 명확한 변경 요구사항이 있는가?
- [ ] 테스트하기 어려운가?
- [x] 여러 팀/모듈이 의존하는가?
- [ ] 외부 시스템과 연동되는가?

3개 이상 ✓ = OCP 적용 고려

GO
3개 이상

WAIT
2개

STOP
1개 이하

지혜로운 개발자가 되는 법

"The best code is no code at all"

- Jeff Atwood

"Make it work, make it right, make it fast"

- Kent Beck

END.