SOLID 원칙과 객체지향 설계4

유지보수 가능한 소프트웨어 설계의 핵심 원칙

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)**

ISP(Interface Segregation Principle),
DIP(Dependency Inversion Principle)

인터페이스 분리 원칙 (ISP) - "필요한 것만 의존하자"

의존성 역전 원칙 (DIP) - "구체적인 것보다 추상적인 것에"

ISP(Interface Segregation Principle)

Interface Segregation Principle 인터페이스 분리 원칙

"클라이언트는 사용하지 않는 인터페이스에 의존하지 않아야 한다"

- Robert C. Martin

ISP를 영화로 이해하기

어벤져스와 ISP

```
ISuperhero {
    Fly();          // 아이언맨 ✅, 헐크 ❌
    UseMagic();     // 닥터스트레인지 ✅, 캡틴 ❌
    Shoot();        // 호크아이 ✅, 토르 ❌
}

✅ 각자 필요한 능력만!
IFlyer, IMagician, IShooter...
```

역할 중심의 인터페이스

Role-based Interfaces

인터페이스는 "역할"을 표현

- 클라이언트가 필요로 하는 것
- 구현자가 제공할 수 있는 것
- 둘 사이의 계약

핵심: WHO needs WHAT?

Fat Interface Problem

뚱뚱한 인터페이스의 문제

```
public interface IEmployee {
    // 인사팀이 필요한 것
    void UpdatePersonalInfo();

    // 회계팀이 필요한 것
    void CalculateSalary();

    // IT팀이 필요한 것
    void UpdateSystemAccess();

    // 마케팅팀이 필요한 것?!
    void SendPromotionalEmail();
}
```

모든 직원이 홍보 이메일을 보내나?

ISP 적용 전략

인터페이스 분리의 3가지 접근법

1. **역할 기반 분리 (Role-based)**
   "이 인터페이스의 사용자는 누구인가?"
2. **기능 기반 분리 (Feature-based)**
   "어떤 기능들이 함께 사용되는가?"
3. **변경 빈도 기반 분리 (Change-based)**
   "무엇이 자주 바뀌는가?"

역할 기반 분리

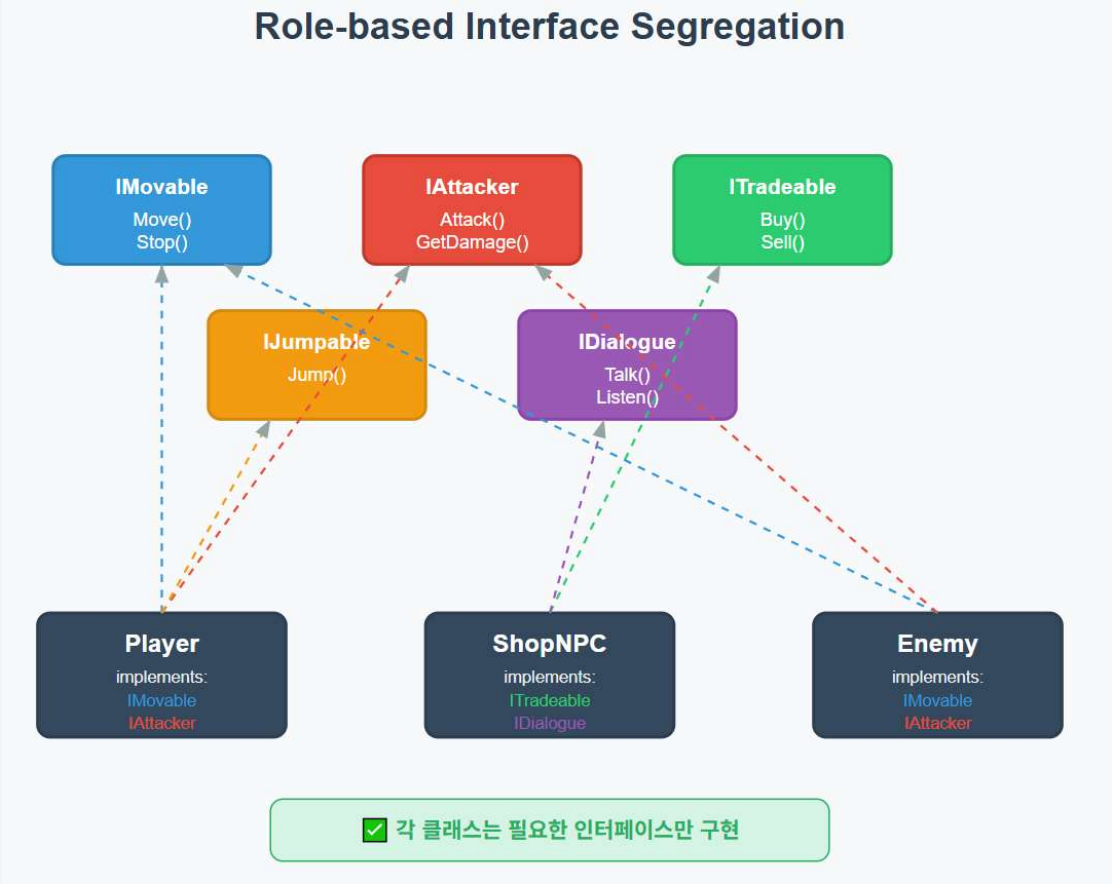
역할별로 인터페이스를 나누자
예: 게임 캐릭터 시스템

플레이어가 보는 역할

- IDisplayable (화면 표시)
- IInteractable (상호작용)
- IControllable (조작)

시스템이 보는 역할

- ISaveable (저장/로드)
- INetworkable (네트워크 동기화)
- IPoolable (오브젝트 풀링)



기능 기반 분리

관련 기능끼리 묶기
예: 전투 시스템

기능별로 분리

- IAttacker - 공격 기능
- IDefender - 방어 기능
- IDamageable - 피해 받기
- IHealable - 치유 받기

하나의 거대한 Icombat 은 NO!

변경 빈도 기반 분리

자주 바뀌는 것 vs 안정적인 것

안정적 (거의 안 바뀜)

- IIdentifiable { GetID(); }
- INamed { GetName(); }

자주 변경

- IGameRules { /* 게임 규칙 */ }
- IBalance { /* 밸런스 수치 */ }

자주 바뀌는 것은 더 작게 분리!

실제 적용 예시

Before

```
interface ICharacterAbility {
    void MeleeAttack();
    void RangedAttack();
    void CastSpell();
    void Heal();
    void Stealth();
    // 20개 더...
}
```

After

```
// 전사 = 근접전투 + 방어
class Warrior : IMeleeAttacker, IDefender {
    void MeleeAttack() { /* 검으로 공격 */ }
    void Block() { /* 방패로 막기 */ }
}

// 궁수 = 원거리 + 은신
class Archer : IRangedAttacker, IStealther {
    void RangedAttack() { /* 활쏘기 */ }
    void Hide() { /* 숨기 */ }
}

// 성직자 = 치유 + 버프
class Priest : IHealer, IBuffer {
    void Heal() { /* 치유 */ }
    void Buff() { /* 강화 */ }
}
```

ISP와 SRP의 관계

형제 같은 두 원칙

SRP (Single Responsibility)

"하나의 클래스는 하나의 책임"

ISP (Interface Segregation)

"하나의 인터페이스는 하나의 역할"

공통점: 응집도 ↑, 결합도 ↓
차이점: SRP는 구현, ISP는 계약

ISP와 LSP의 상호작용

대체 가능성과 인터페이스 설계

LSP를 지키려면 ISP가 필요하다!

큰 인터페이스 → 구현 어려움 → LSP 위반 가능성 ↑

작은 인터페이스 → 구현 쉬움 → LSP 준수 가능성 ↑

예: IFlyable
✓ Bird implements IFlyable // OK
✓ Airplane implements IFlyable // OK
✗ Penguin implements IFlyable // LSP 위반!

과도한 분리의 위험

```
극단적인 예:
interface IWalkable { void Walk(); }
interface IRunnable { void Run(); }
interface IStoppable { void Stop(); }
interface ITurnable { void Turn(); }
// ... 메서드 하나당 인터페이스 하나?!
```

문제점:

- 인터페이스 폭발 ✸
- 관리 복잡도 증가
- 코드 가독성 저하

ISP 정리

Interface Segregation Principle

핵심 메시지: "클라이언트가 필요한 것만 제공하라"

- 유연한 설계
- 쉬운 테스트
- 낮은 결합도
- 명확한 책임

DIP
(Dependency Inversion Principle)

"고수준 모듈은 저수준 모듈에 의존하지 않아야 한다.
둘 다 추상화에 의존해야 한다."
- Robert C. Martin

의존성의 방향


전통적 방식 ✗
고수준 정책
의존
중간 레벨
의존
저수준 구현
문제: 경직된 설계
하위 변경 → 상위 영향
DIP 방식 ✓
고수준 정책
의존
<<인터페이스>>
추상화
의존
저수준 구현
장점: 유연한 설계
구현 교체 자유
"추상화에 의존하라!"

영화로 이해하는 DIP

아이언맨과 자비스

토니 스타크가 자비스에게 명령할 때:

"자비스 버전 3.2.1, C:₩Program Files₩Jarvis₩voice.exe 실행해!"

"자비스, 날씨 알려줘" **(DIP 적용)**
→ 구체적인 구현이 아닌 추상적 인터페이스에 의존!

DIP 위반 사례

😮구체 클래스에 직접 의존

```
public class GameManager {
    private MySQLDatabase db = new MySQLDatabase();
    private FileLogger logger = new FileLogger();

    public void SaveGame() {
        logger.LogToFile("Saving game...");
        db.SaveToMySQL(gameData);
    }
}
```

MySQL을 MongoDB로 바꾸려면...?

DIP 적용

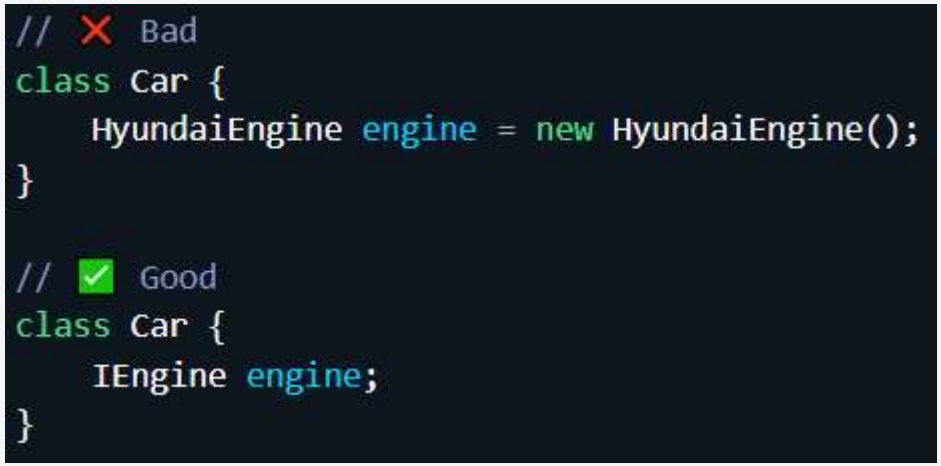
추상화에 의존

```
public class GameManager {
    private IDatabase db;
    private ILogger logger;

    public GameManager(IDatabase db, ILogger logger) {
        this.db = db;
        this.logger = logger;
    }

    public void SaveGame() {
        logger.Log("Saving game...");
        db.Save(gameData);
    }
}
```

구현체 교체 자유!

DIP - 설계 원칙

DIP: 의존성 역전 원칙은 **"추상화에 의존하라"**는 철학



토니 스타크가 자비스에게 명령할 때:

"자비스 버전 3.2.1, C:₩Program Files₩Jarvis₩voice.exe 실행해!"

"자비스, 날씨 알려줘" **(DIP 적용)**

→ 구체적인 구현이 아닌 추상적 인터페이스에 의존!

DI - 구현 기법

DI: 의존성 주입

"의존성을 외부에서 주입"하는 방법

3가지 주입 방식

- 생성자 주입
- 프로퍼티 주입
- 메서드 주입

```
// 생성자 주입 예시
public Car(IEngine engine) {
    this.engine = engine;
}
```

IoC - 제어의 역전

IoC: Inversion of Control

"프레임워크가 당신의 코드를 호출"

전통적 방식:
내 코드 → 라이브러리 호출

IoC 방식:
프레임워크 → 내 코드 호출

예: 이벤트 핸들러, 콜백 함수

세 개념의 관계

IoC: Inversion of Control

"프레임워크가 당신의 코드를 호출"

전통적 방식:

내 코드 → 라이브러리 호출

IoC 방식:

프레임워크 → 내 코드 호출

예: 이벤트 핸들러, 콜백 함수

ISP + DIP 시너지

두 원칙의 환상적인 조합

ISP: 작은 인터페이스 + DIP: 추상화 의존

= 유연하고 테스트 가능한 설계

```
interface IReader { string Read(); }
interface IWriter { void Write(string data); }

class FileHandler : IReader, IWriter {
    // 필요한 것만 구현
}
```

종합 예제

게임 저장 시스템 리팩토링

문제상황

```
class GameSaveManager {
    void SaveGame() {
        // 파일에 직접 저장
        File.WriteAllText("save.dat", data);

        // MySQL에 백업
        var mysql = new MySqlConnection();
        mysql.Execute("INSERT...");

        // 스팀 클라우드 동기화
        SteamAPI.SaveToCloud(data);
    }
}
```

종합 예제

ISP + DIP 적용

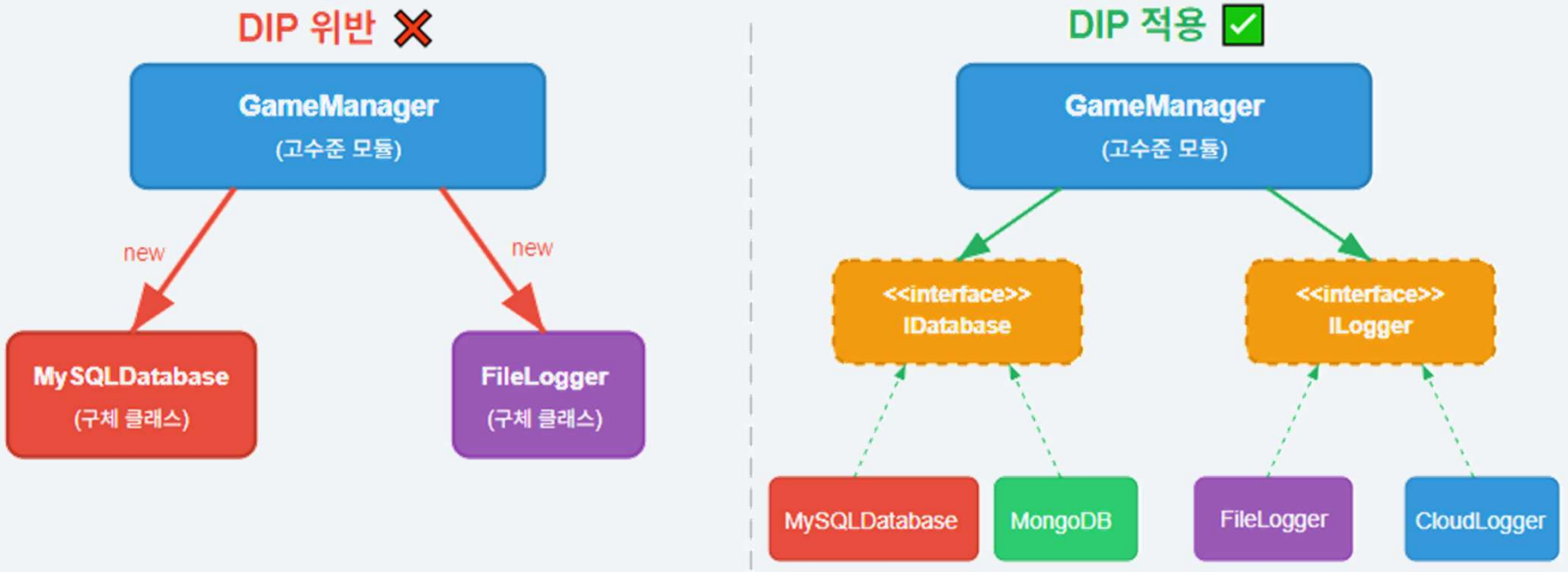
```
// ISP: 작은 인터페이스들
interface ISaveWriter {
    void Write(GameData data);
}

interface ISaveReader {
    GameData Read();
}

// DIP: 추상화에 의존
class GameSaveManager {
    private readonly IList<ISaveWriter> writers;

    public GameSaveManager(IList<ISaveWriter> writers) {
        this.writers = writers;
    }

    public void SaveGame(GameData data) {
        foreach(var writer in writers) {
            writer.Write(data);
        }
    }
}
```

종합 예제

확장 가능한 시스템으로 리팩토링 완료!

```
// 다양한 구현체
class FileSaveWriter : ISaveWriter { }
class CloudSaveWriter : ISaveWriter { }
class DatabaseSaveWriter : ISaveWriter { }

// 조합 자유!
var saveManager = new GameSaveManager(new[] {
    new FileSaveWriter(),
    new CloudSaveWriter()
});

// 새로운 저장 방식 추가도 쉽게!
class BlockchainSaveWriter : ISaveWriter { }
```

DIP 위반 vs DIP 적용




```
public class GameManager {
private MySQLDatabase db = new MySQLDatabase();
private FileLogger logger = new FileLogger();
// 구체 클래스에 강하게 결합!
```

문제: 변경이 어려움

MySQL → MongoDB 변경 시 코드 수정 필요

```
public class GameManager {
    private IDatabase db;
public GameManager(IDatabase db, ILogger log) {
this.db = db; // 추상화에 의존!
```

장점: 유연한 변경

구현체 교체가 자유로움

"구체적인 것이 아닌 추상적인 것에 의존하라"

이것이 DIP의 핵심!

SOLID 전체 정리

SOLID 원칙 총정리

S - 단일 책임: 한 가지만 잘하자

O - 개방-폐쇄: 확장엔 열려있고 수정엔 닫혀있게

L - 리스코프 치환: 자식은 부모를 대체 가능

I - 인터페이스 분리: 필요한 것만 의존

D - 의존성 역전: 추상화에 의존 함께 사용하면 더 강력한 설계!

실무 적용 팁

1. **점진적 적용**
   - 한 번에 모든 원칙 적용 X
   - 문제가 있는 부분부터 개선
2. **과도한 설계 주의**
   - YAGNI (You Ain't Gonna Need It)
   - 필요할 때 리팩토링
3. **팀과 함께**
   - 코드 리뷰에서 SOLID 체크
   - 팀 컨벤션으로 정착

"완벽한 설계보다 개선 가능한 설계"

END.