

클린 코드

오정임(purum01@naver.com)

목차

- section01 Clean Code 개요
- section02 객체지향 프로그램
- section03 클래스와 상속
- section04 소프트웨어의 변화와 유지보수성
- section05 소프트웨어 설계 원칙
- section06 객체지향 설계 원칙

section01 클린 코드 개요

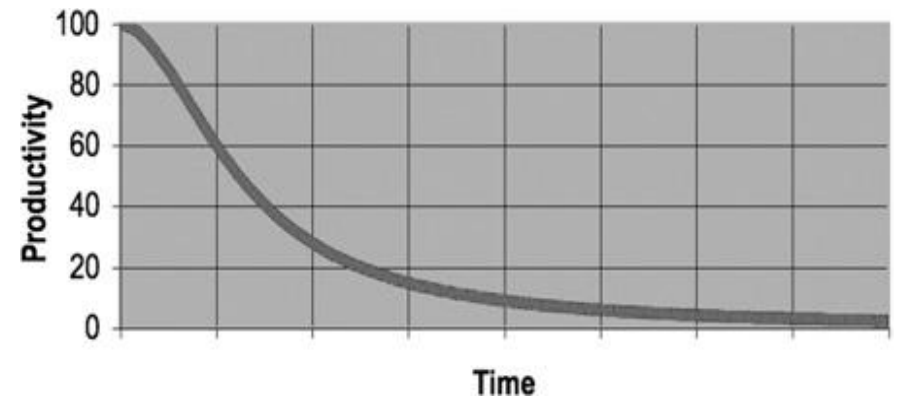
1. 클린 코드 정의
2. 명명 규칙
3. 괄호 및 주석

1. 클린 코드 개요

- 프로그래밍은 요구사항을 명확히 표현하는 작업이며, 자동화 기술이 발전해도 그 중요성은 사라지지 않는다. 따라서 **코드 품질을 지속적으로 유지하는 것은 모든 개발자의 기본 자질**이다.

나쁜 코드(Bad Code)란?

- **코드 품질:** "코드 품질은 분당 외치는 WTF 횟수로 측정된다."
- **나쁜 코드의 원인:** 급함, 시간 부족, 후속 수정 미루기 등
- **방치 결과:** 버그 발생, 시스템 성능 저하
- **유사 이론:** '깨진 유리창 이론'처럼 나쁜 코드는 더 많은 나쁜 코드를 만든다



클린 코드(Clean Code)란?

- 클린 코드란?
 - 가독성이 높은 코드로, 나쁜 코드의 반대 개념
- 나쁜 코드 개선
 - 나쁜 코드를 식별하고 개선하는 기법을 익혀야 함
- 지속적인 연습과 자세
 - 클린 코드를 위해선 꾸준한 연습과 '보이스카우트 규칙' 같은 자기관리 필요
- 가독성이 최우선 기준
 - 코드가 쉽게 읽히고 이해될 수 있어야 클린 코드라 할 수 있음
- 관리자의 압박보다 프로그래머의 책임감
 - 외부 압력보다도 개발자의 코드 책임 의식이 중요

비아네 스트롭(C++ 창시자)

“나는 우아하고 효율적인 코드를 좋아한다.

논리가 간단해야 버그가 숨어들지 못한다.

의존성을 최대한 줄여야 유지보수가 쉬워진다. 오류는 명백한 전략에 의거해 철저히 처리한다.

성능을 최적으로 유지해야 사람들이 원칙 없는 최적화로 코드를 망치려는 유혹에 빠지지 않는다.”

“깨끗한 코드는 한 가지를 제대로 한다.”

그레디 부치(객체지향 대가, UML 개발)

“깨끗한 코드는 단순하고 직접적이다.

깨끗한 코드는 잘 쓴 문장처럼 읽힌다.

깨끗한 코드는 결코 설계자의 의도를 숨기지 않는다.

오히려 명쾌한 추상화와 단순한 제어문으로 가득하다.”

마이클 페더즈 (“Working Effectively with Legacy Code” 저자)

깨끗한 코드의 특징은 많지만 그 중에서도 모두를 아우르는 특징이 하나 있다.

“깨끗한 코드는 언제나 누군가 주의 깊게 짰다는 느낌을 준다.”

고치려고 살펴봐도 딱히 손 댈 곳이 없다.

작성자가 이미 모든 사항을 고려했으므로, 고칠 궁리를 하다 보면 언제나 제자리로 돌아온다.

그리고는 누군가 남겨준 코드 누군가 주의 깊게 짜놓은 작품에 감사를 느낀다.

론 제프리 (“Extreme Programming Installed” 저자, 애자일 선언)

최근 들어 나는 [켄트 벡]이 제안한 간단한 코드 규칙으로 구현을 시작한다. 중요한 순으로 나열하자면 간단한 코드는 모든 테스트를 통과한다.

중복이 없다.

시스템 내 모든 설계 아이디어를 표현한다.

클래스, 메소드, 함수 등을 최대한 줄인다.

물론 나는 주로 중복에 집중한다. 같은 작업을 여러 차례 반복한다면 코드가 아이디어를 제대로 표현하지 못한다는 증거다.

나는 문제의 아이디어를 찾아내 좀 더 명확하게 표현하려 애쓴다.

“중복 줄이기, 표현력 높이기, 초반부터 간단한 추상화 고려하기”

내게는 이 세가지가 깨끗한 코드를 만드는 비결이다.

2.명명 규칙(Naming Rule)

- 좋은 이름은 코드의 이해와 역할 파악을 쉽게 도와준다.
- 변수, 함수, 클래스, 패키지 등 모든 구성 요소에서 이름 짓기는 가독성과 유지보수성의 핵심 요소이다.
- 이름을 지을때는 의도가 분명하게 해야 한다.

`int td; // 하루에 경과한 시간(단위 : 시간) → int elapsedTimeInDays`

명명 규칙 위반

어떤 기능의 메소드인가?

```
public List<int[]> getThem() {
```

```
List<int[]> list = new ArrayList<int[]>();
```

```
for (int[] x : theList)
```

어떤 데이터들이 들어있는가?

```
if(x[0] == 4)
```

0번 인덱스 값은 왜 특별한가?

값 4가 갖는 의미는 무엇인가?

```
list.add(x);
```

```
return list;
```

```
}
```

명명 규칙 준수

플래그된 셀을 가져오는 함수

```
public List<int[]> getFlaggedCells() {
```

```
List<int[]> flaggedCells = new ArrayList<int[]>();
```

```
for (int[] cell : gameBoard)
```

루프 변수로서 각 배열
이 셀 하나를 의미함을
드러냄

```
if (cell[STATUS_VALUE] == FLAGGED)
```

리스트의 내용이 무엇인지 정확하게 표현

```
flaggedCells.add(cell);
```

의미 있는 상수 이름으로
매직 넘버 제거

```
return flaggedCells;
```

```
}
```

비슷한 이름, 다른 기능 – 메소드 네이밍의 함정

- `getActiveAccount()`, `getActiveAccounts()`, `getActiveAccountInfo()`는 이름이 비슷해 용도 구분이 어렵다.
- 메소드 사용자(클래스 사용자)는 각 메소드의 정확한 기능을 파악하기 어려움.
- 의도가 동일하다면 통합

`AccountInfo getActiveAccountInfo();` //단일 메소드로 통합

- 기능이 다르다면 구분 가능한 이름 부여

`Account getPrimaryActiveAccount();`

`List<Account> getAllActiveAccounts();`

`AccountInfo getActiveAccountDetails();`

변수명은 가독성이 핵심이다

- 로컬 변수에 대충 짓는 경향이 있다.
- 예: status → st, stt, 또는 temp, a, b처럼 의미 없는 이름 사용
- 결과적으로 코드의 가독성 저하 및 이해 시간 증가를 초래

변수명, 길이와 축약어에 주의하라

- 변수명 내의 축약어는 모두 대문자로 표현한다.
- 의미 없는 3글자 이하 변수명은 피한다 (반복문의 i, j 같은 임시 변수는 예외)
- 너무 긴 변수명도 피한다(가독성과 간결함 사이의 균형이 중요)

변수명	설명
String Value;	대문자로 시작
String VALUE	모두 대문자로 구성
String productname;	새로운 단어의 첫 글자를 소문자로 작성
String n;	의미가 불분명한 한 글자의 변수명
String numValue;	의미가 부정확한 변수명
boolean status\$;	달러(\$) 기호 사용
String product_name;	밑줄(_) 기호 사용

상수는 대문자와 밑줄로 작성하라

- 상수는 소프트웨어 전반에서 사용되며, 변경 가능성이 거의 없는 중요한 값을 저장한다.
- 상수명 규칙은 일반 변수명 규칙과 동일하며, 대문자와 밑줄(_)로 구성한다.

상수명	설명
<code>static final int maxValue = 100000;</code>	소문자 사용
<code>static final int MINVALUE = 1;</code>	잘못된 단어 간 구분

메소드명 규칙

- 메소드명에는 동사 또는 동사와 명사의 조합을 사용한다.
- 명사를 사용할 때 너무 긴 경우 축약해서 사용할 수 있지만 의미가 불분명한 너무 짧은 이름은 자제해야한다.
- 메소드명 내의 축약어는 모두 대문자로 표현한다.
- 의미가 불분명한 세 글자 이하의 메소드명은 쓰지 않는다.

메소드명	설명
public void ParseInt() { }	대문자로 시작
public void parseInt() { }	새로운 단어의 첫 글자를 소문자로 시작
public void parse_int() { }	밑줄(_) 기호 사용
public void parse\$int() { }	달러(\$) 기호 사용
public void pi() { }	지나친 약어 사용

패키지명 규칙

- 패키지명은 소문자로 구성한다.
- 패키지명은 패키지의 기능을 정확히 전달할 수 있는 한 단어의 명사로 한다.
- 패키지의 최상위 부분은 소문자의 도메인 이름이어야 한다.
- 하위 컴포넌트명은 내부 명명 규칙을 따를 수 있다.
- 달러 기호(\$)를 패키지 명으로 사용하지 않는다.

패키지 명	설명
<code>package TEST;</code>	대문자로 구성된 패키지명 사용
<code>package wikibook.co.kr;</code>	순서가 잘못된 패키지 도메인명 사용
<code>package kr.co.wikiBook;</code>	패키지명에 대문자를 사용
<code>package kr.co.testUnit;;</code>	한 단어 이상의 패키지명 사용

클래스명과 인터페이스명 규칙

- 클래스명은 파스칼 표기법을 바탕으로 명명해야한다.
- 클래스명에는 명사만 사용할 수 있다.
- 명사를 사용할 때 너무 긴 경우 축약해서 사용할 수 있지만 의미가 불분명한 너무 짧은 이름은 자제한다.
- 클래스명 내의 축약단어는 모두 대문자로 표현한다.

잘못된 클래스명	수정된 클래스명
public class content { }	public class Content { }
public class CONTENT { }	public class Content { }
public class Stringbuilder { }	public class StringBuilder { }
public class HtmlUtil { }	public class HTMLUtil { }
public class Doc { }	public class Document { }
public class Order_Util { }	public class OrderUtil { }
public class \$ServiceBuiler { }	public class ServiceBuiler { }

괄호 규칙

올맨 스타일

```
public void test() {  
    for (int num = 0; num < 10; num++)  
    {  
        sum = sum + num;  
        if (sum > 10)  
        {  
            //...  
        }  
        else  
        {  
            //...  
        }  
    }  
}
```

K&R 스타일

```
public void test() {  
    for (int num = 0; num < 10; num++) {  
        sum = sum + num;  
        if (sum > 10) {  
            // .....  
        } else {  
            // ...  
        }  
    }  
}
```

괄호 규칙

- **K&R 스타일:** 중괄호를 생략할 수 있는 경우에는 생략할 수 있다

```
if (sum > 10)
    // ...
else
    // ...
```

- **1TBS 스타일:** 모든 제어문에 중괄호를 사용하는 것을 권장한다

```
if (sum > 10) {
    // ...
} else {
    // ...
}
```

주석의 함정: 설명보다 이해하기 쉬운 코드

- **주석의 필요성** : 코드만으로 의미를 전달할 수 없을 때 사용하며, 이해하기 쉬운 코드를 작성하면 주석이 불필요하다.
- **나쁜 코드의 방증** : 주석은 복잡한 코드를 설명하는 대신, 코드를 간결하게 만들어 주석을 줄여야 한다.
- **코드로 의도 표현** : 주석 대신 함수로 설명을 처리하고, 코드 자체로 의도를 표현하려고 노력해야 한다.
- **주석 유지보수 문제** : 시간이 지나면서 주석은 코드와 불일치하게 되어 혼란을 초래할 수 있다.
- **부정확한 주석의 위험** : 부정확한 주석은 오히려 해가 되므로, 주석을 제거하고 코드로 설명을 대체해야 한다.

좋은 주석

- 법적인 주석
- 정보를 제공하는 주석
- 의도를 설명하는 주석
- 의미를 명료하게 밝히는 주석
- 결과를 경고하는 주석
- TODO 주석
- 중요성을 강조하는 주석

나쁜 주석

- 주절거리는 주석
- 같은 이야기를 중복하는 주석
- 의무적으로 다는 주석
- 이력을 기록하는 주석
- 있으나 마나 한 주석
- 위치를 표시하는 주석
- 닫는 괄호에 다는 주석
- 공로를 돌리거나 저자를 표시하는 주석
- 주석으로 처리된 코드

section02 객체지향 프로그램

절차 지향 프로그램 VS 객체 지향 프로그램 특징

객체지향 기본 원리(캡슐화와 정보은닉, 추상화, 상속과 다형성)

객체지향스러운 자바 코드

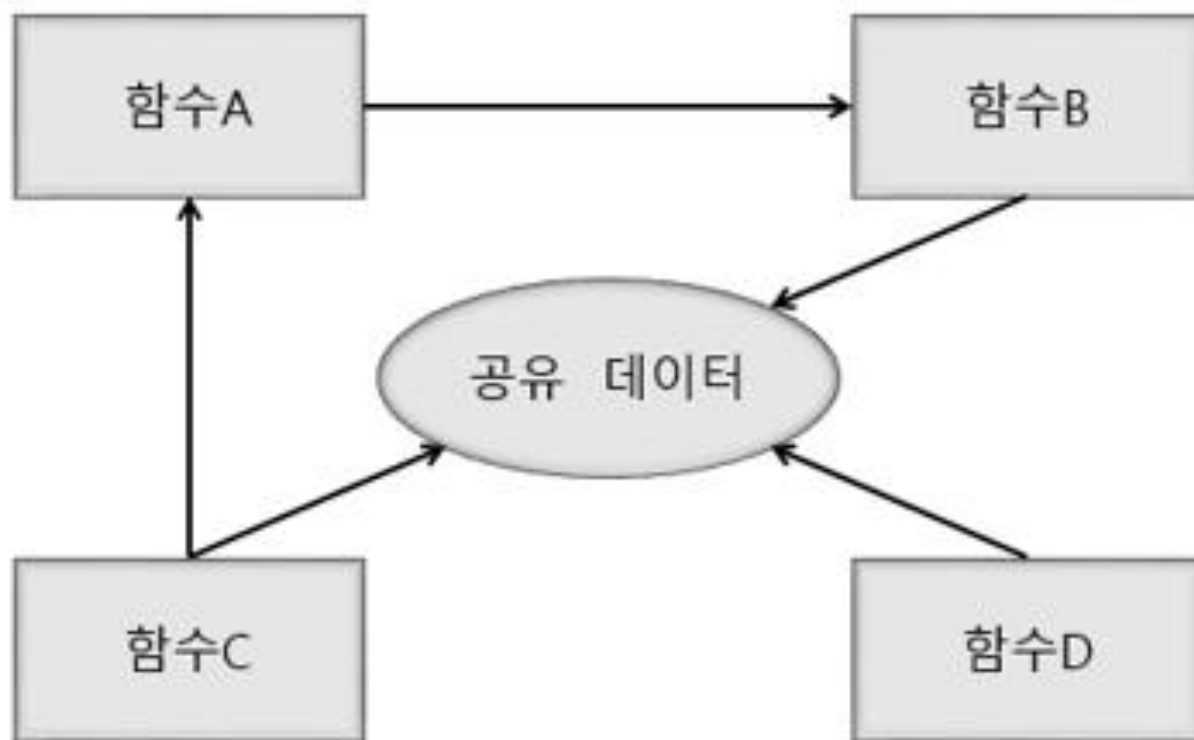
객체지향으로 다듬는 클린 코드

- 클린 코드의 목적
 - 가독성과 유지보수성 향상 : 누구나 쉽게 읽고 해석할 수 있는 코드 작성
- 유지보수의 중요성
 - 지저분한 코드는 이해에 시간 소모 : 유지보수가 어려워지면 비용 증가
- 객체지향 언어의 장점
 - 가독성, 유지보수성 우수
 - 유연성 & 확장성 → 새로운 기능 추가/변경 용이
- 객체지향 코딩은 훈련이 필요
 - 좋은 코드는 오랜 훈련과 노력의 결과물

절차지향 프로그램

- 절차지향 언어의 구조
 - 데이터 중심이며, 데이터를 처리하는 여러 함수로 구성된다.
- 함수 호출 방식
 - 필요한 데이터를 인자로 지속적으로 함수에 공급해야 한다.
- 전역 변수 사용
 - 함수 실행 후 상태 정보를 저장하기 위해 전역 변수를 사용한다.
- 전역 변수의 문제점
 - 전역 변수를 여러 함수가 공유하면 의도하지 않은 결과가 나올 수 있다.

절차지향 프로그램



```
public class Stack {  
    int size;  
    Object[] items;  
    int top = -1;  
    public Stack() {  
        size = 10;  
        items = new Object[size];  
    }  
    public Stack(int size) {  
        this.size = size;  
        items = new Object[size];  
    }  
    public void setItem(int index, Object item) {  
        items[index] = item;  
    }  
    public Object getItem(int index) {  
        return items[index];  
    }  
}
```

```
class StackFunction {  
    public boolean isEmpty(Stack stack) {  
        return (stack.top < 0);  
    }  
    public boolean isFull(Stack stack) {  
        return (stack.top == stack.size);  
    }  
    public boolean push(Stack stack, Object newItem) {  
        if (isFull(stack))  
            return false;  
        else {  
            int newTop = stack.top + 1;  
            stack.top = newTop;  
            stack.setItem(newTop, newItem);  
            return true;  
        }  
    }  
}
```

```
public Object pop(Stack stack) {  
    if (isEmpty(stack))  
        return null;  
    else {  
        int currentTop = stack.top;  
        Object item = stack.getItem(currentTop);  
        stack.top = currentTop - 1;  
        return item;  
    }  
}
```

```
public Object top(Stack stack) {  
    if (isEmpty(stack))  
        return null;  
    else  
        return stack.getItem(stack.top);  
}
```

```
}
```

객체지향 프로그램

- 객체지향 프로그래밍의 구조
 - 데이터와 함수가 객체 안에 함께 캡슐화된다.
- 속성과 메소드
 - 데이터는 속성이 되고, 이를 다루는 함수는 메소드가 된다.
- 데이터 캡슐화
 - 속성은 외부에 직접 노출되지 않으며, 필요한 데이터는 메소드 매개변수로 받아들인다.

객체지향 프로그램

🔑 절차지향 vs 객체지향

절차지향	객체지향
데이터와 함수가 분리됨	데이터와 함수가 객체 내부에 함께 존재
함수는 항상 외부 데이터를 매개변수로 받아야 함	객체가 자신의 상태(속성)을 유지
프로그램 전체가 순서대로 동작	책임을 가진 객체들이 서로 메시지를 주고 받음

📌 객체지향의 핵심 구조

속성(Attribute): 객체 내부의 데이터

메소드(Method): 속성을 다루는 함수

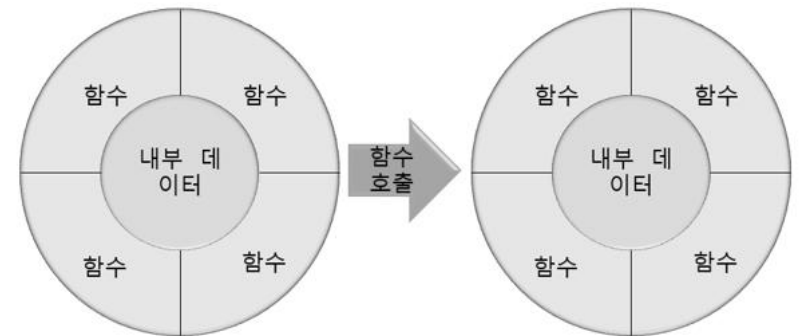
캡슐화(Encapsulation): 내부 데이터는 외부에 직접 노출되지 않음

💡 주요 특징

객체는 자기 상태를 내부에 저장

매번 데이터를 넘기지 않아도 메소드가 상태 기반 처리 가능

역할과 책임이 분산된 객체들이 협력하여 로직 수행



```
public class Stack {  
    private int size;  
    private Object[] items;  
    private int top = -1;  
  
    public Stack() {  
        size = 10;  
        items = new Object[size];  
    }  
    public Stack(int size) {  
        this.size = size;  
        items = new Object[size];  
    }  
    public boolean isEmpty() {  
        return (top < 0);  
    }  
    public boolean isFull() {  
        return (top >= size);  
    }  
}
```

```
    public boolean push(Object newItem) {  
        if (isFull())  
            return false;  
        else {  
            top++;  
            items[top] = newItem;  
            return true;  
        }  
    }  
    public Object pop() {  
        if (isEmpty())  
            return null;  
        else  
            return items[top--];  
    }  
    public Object top() {  
        if (isEmpty())  
            return null;  
        else  
            return items[top];  
    }  
}
```

```
public boolean push(Object newItem) {
    if (isFull())
        return false;
    else {
        top++;
        items[top] = newItem;
        return true;
    }
}

public Object pop() {
    if (isEmpty())
        return null;
    else
        return items[top--];
}

public Object top() {
    if (isEmpty())
        return null;
    else
        return items[top];
}
```

```
Stack stack1 = new Stack();
Stack stack2 = new Stack();
stack1.push("1");
stack2.push("2");
```

객체지향 기본 원리 – 캡슐화(Encapsulation)

- 캡슐화는 관련된 데이터(속성)와 해당 데이터를 처리하는 메소드(행위)를 하나의 단위(클래스)로 묶고, 외부에서 직접 접근하지 못하도록 제한(정보 은닉)하여 객체의 내부 구현을 보호하는 객체지향의 핵심 원리이다.
- 단순히 묶는 것(aggregation) 이상으로, 접근 제어와 정보 보호가 핵심이다.
- 유지보수성 향상, 오류 방지, 명확한 인터페이스 설계로 이어진다.

객체지향 기본 원리 - 정보 은닉

- 정보 은닉은 객체지향의 핵심 원리 중 하나로, 객체 내부의 구현 세부사항을 외부로부터 숨기고, 필요한 정보나 기능만을 공개하는 것이다. 이를 위해 접근 제어자 `private`을 사용하며, 이는 데이터 보호와 코드의 안정성을 높이는 데 기여한다.

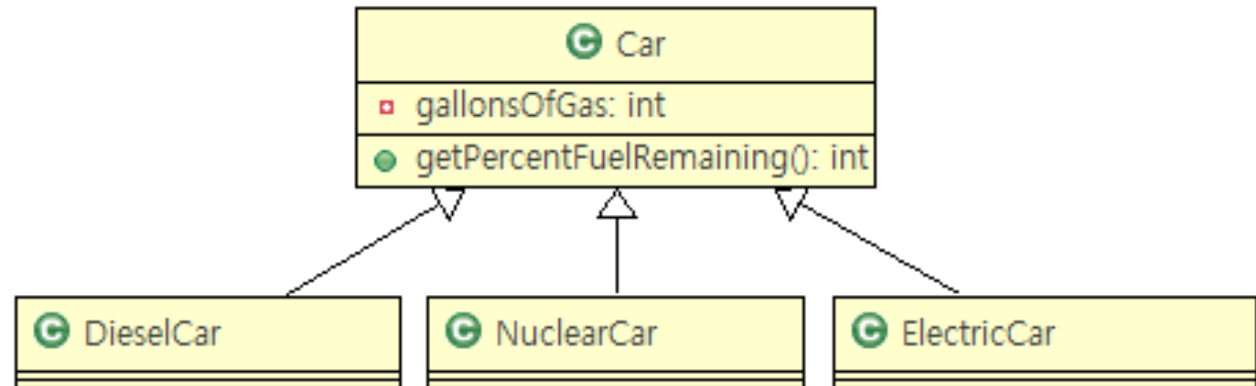
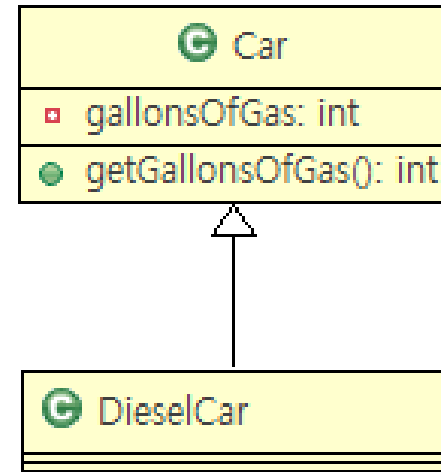
객체지향 기본 원리 – 추상화(Abstraction)

- 추상화는 객체의 복잡한 내부 구현을 감추고, 핵심적인 기능이나 공통적인 특성만을 외부에 제공하는 객체지향의 핵심 원리이다. 자바에서는 abstract class, abstract method, 그리고 interface 등을 통해 추상화를 구현할 수 있다.

요소	추상화에 사용 가능	설명
abstract class	✓	일부 구현이 가능, 공통 속성과 동작 정의
abstract method	✓	하위 클래스에서 반드시 구현해야 함
interface	✓	완전한 추상화 – 구현 없이 구조만 정의

추상화(Abstraction)

- Car 클래스의 getGallonsOfGas() 메소드는 내부 변수 이름을 드러내어 캡슐화와 추상화 원칙을 위반한다.
- Car를 상속받는 DieselCar처럼 연료 방식이 다른 클래스에도 부적절한 메소드가 상속되는 문제가 발생한다.
- getPercentFuelRemaining()처럼 구체적인 구현을 감춘 추상적인 메소드 이름이 더 적절하며, 메소드에서도 추상화를 적용하는 것이 중요하다.



상속(inheritance)과 다형성(polymorphism)

▶ 상속(inheritance)의 기본 의미

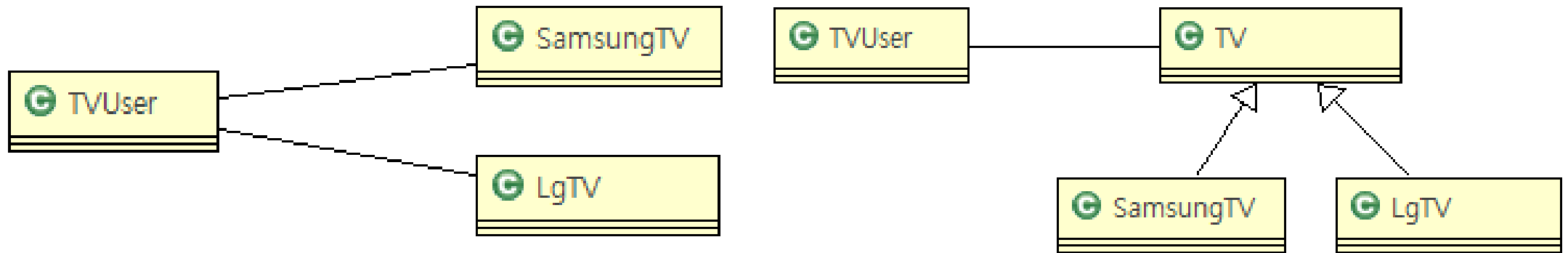
- 하위 클래스가 부모 클래스의 속성과 메소드를 다운 클래스에서 재사용 가능
- 복잡한 코드의 반복을 줄이고, 개발 효율성과 코드의 이해도를 높이며, 유지보수와 확장성을 향상시킬 수 있다.

▶ 다형성(polymorphism)

- 부모 클래스 타입으로 다양한 하위 클래스 객체를 다룰 수 있는 것
- 부모 클래스에서 선정된 메소드를 호출하지만, 실제적으로는 하위 클래스의 구현이 실행되는 방식을 가능하게 한다.

상속(inheritance)과 다형성(polymorphism)

- 다형성은 같은 타입(예: 리모컨)으로 다양한 객체(SamsungTV, LgTV 등)를 제어할 수 있게 해주는 객체지향의 핵심 개념이다
- .TV가 바뀌더라도 TVUser와 같은 클라이언트 코드를 수정하지 않고 재사용 가능하므로, 코드의 유지보수성과 확장성을 크게 높일 수 있다.



```
abstract class TV {  
    public abstract void powerOn();  
    public abstract void powerOff();  
    public abstract void volumeUp();  
    public abstract void volumeDown();  
}
```

```
class SamsungTV extends TV {  
    public void powerOn() {  
        System.out.println("SamsungTV---전원 켜다.");  
    }  
    public void powerOff() {  
        System.out.println("SamsungTV---전원 끈다.");  
    }  
    public void volumeUp() {  
        System.out.println("SamsungTV---소리 올린다.");  
    }  
    public void volumeDown() {  
        System.out.println("SamsungTV---소리 내린다.");  
    }  
}
```

```
class LgTV extends TV {  
    public void powerOn() {  
        System.out.println("LgTV---전원 켜다.");  
    }  
    public void powerOff() {  
        System.out.println("LgTV---전원 끈다.");  
    }  
    public void volumeUp() {  
        System.out.println("LgTV---소리 올린다."); }  
    }  
    public void volumeDown() {  
        System.out.println("LgTV---소리 내린다.");  
    }  
}
```

```
public class TVUser {  
    public static void main(String[] args) {  
        TV tv = new SamsungTV();  
        tv.powerOn();  
        tv.volumeUp();  
        tv.volumeDown();  
        tv.powerOff();  
    }  
}
```

```
class BeanFactory {  
    public TV getBean(String name) {  
        if(name.equals("lg")) {  
            return new LgTV();  
        } else if(name.equals("samsung")) {  
            return new SamsungTV();  
        }  
        return null;  
    }  
}
```

```
public class TVUser {  
    public static void main(String[] args) {  
        BeanFactory factory = new BeanFactory();  
  
        TV tv = (TV)factory.getBean(args[0]);  
        tv.powerOn();  
        tv.volumeUp();  
        tv.volumeDown();  
        tv.powerOff();  
    }  
}
```

객체지향적이지 않은 자바 코드

- getGender()는 내부 속성값에 따라 동작을 결정하는 방식은 비객체지향적이다.
- 객체의 행위가 상황에 따라 달라질 경우 상속을 통해 자식 클래스에 그 행위를 구체화하는 것이 객체지향 방식이다.

```
public class Person {  
    private boolean genderFlag;  
    public Person(boolean genderFlag) {  
        this.genderFlag = genderFlag;  
    }  
    public String getGender() {  
        return genderFlag ? "male" : "female";  
    }  
}
```

```
public static void main(String[] args) {  
    Person kim = new Person(true);  
    if(kim.getGender().equals("male"))  
        System.out.println("남성입니다. ");  
    else  
        System.out.println("여성입니다. ");  
}
```

객체지향적인 자바 코드

```
abstract class Person {  
    public abstract String getGender();  
}  
  
class Man extends Person {  
    public String getGender() {  
        return "male";  
    }  
}  
  
class Woman extends Person {  
    public String getGender() {  
        return "female";  
    }  
}
```

getGender()는 Person에서 추상 메소드로 선언하고, 각 자식 클래스가 자신에 맞게 구현함으로써 내부 속성값에 의존하지 않고, 객체 스스로 자신의 성별을 책임 있게 말하게 할 수 있다. 이는 객체지향의 핵심인 책임 중심 설계를 잘 반영한 구조이다.

객체지향적인 자바 코드

```
abstract class Person {  
    public abstract boolean isMale();  
}  
  
class Man extends Person {  
    public boolean isMale() {  
        return true;  
    }  
}  
  
class Woman extends Person {  
    public boolean isMale() {  
        return false;  
    }  
}
```

```
if(kim.isMale()) {  
    System.out.println("남성입니다. ");  
} else {  
    System.out.println("여성입니다. ");  
}
```

getGender().equals("male")처럼 값을 비교해 판단하는 방식은, 내부 로직이 바뀌면 클라이언트 코드 전체에 영향을 미친다.

이보다 나은 방법은 성별을 판단하는 책임 자체를 Person 객체에 위임하는 것이다.

즉, "당신은 남자입니까?"라고 객체에게 직접 묻고 객체가 판단하여 응답하도록 하면, 더 객체지향적이고 변경에 유연한 코드가 된다.

section03 클래스와 상속

클래스와 인터페이스

다형성에서 인터페이스 역할

상속과 합성

객체와 인터페이스

- 객체지향의 본질은 클래스가 아니라 객체와 그 기능
 - 객체는 외부에 기능(서비스) 을 제공하는 것이 핵심
 - 내부 데이터는 외부에 공개되지 않으며 중요하지 않음
- 인터페이스란?
 - 객체가 외부에 제공하는 기능의 명세
 - 메소드의 시그니처(리턴타입, 이름, 매개변수) 로 구성
 - 실제 내부 구현은 알 필요 없이 기능만으로 사용 가능(TV, 계산기)

인터페이스 지정하기

- 자바에서는 객체를 선언할 때 어떤 기능을 쓸 수 있을지가 타입에 따라 결정된다.
- 이때 타입은 클래스일 수도 있고, 인터페이스일 수도 있다.
- 객체가 어떤 클래스로 만들어졌는지가 중요한 게 아니라, "어떤 타입으로 선언되었는가"가 중요하다
- 다음 객체는 ArrayList로 만들어졌지만, List라는 인터페이스 타입으로 선언되었기 때문에 List가 제공하는 기능만 사용할 수 있다.

```
List<String> names = new ArrayList<>();
```

다형성과 다운캐스팅

```
class A {  
    public void methodA1() { System.out.println("A클래스 methodA1()"); }  
    public void methodA2() { System.out.println("A클래스 methodA2()"); }  
    public void methodA3() { System.out.println("A클래스 methodA3()"); }  
}  
  
class B extends A {  
    public void methodA1() {System.out.println("B클래스 methodA1()"); }  
    public void methodB1() {System.out.println("B클래스 methodB1()"); }  
    public void methodB2() {System.out.println("B클래스 methodB1()"); }  
}  
  
public class CastingTest {  
    public static void main(String[] argv) {  
        A a = new A();    a.methodA1();  
        A b = new B();    b.methodA1();  
        ((B) b).methodB1();  
    }  
}
```

✓ 실행 결과:

java

```
A클래스 methodA1()  
B클래스 methodA1()  
B클래스 methodB1()
```

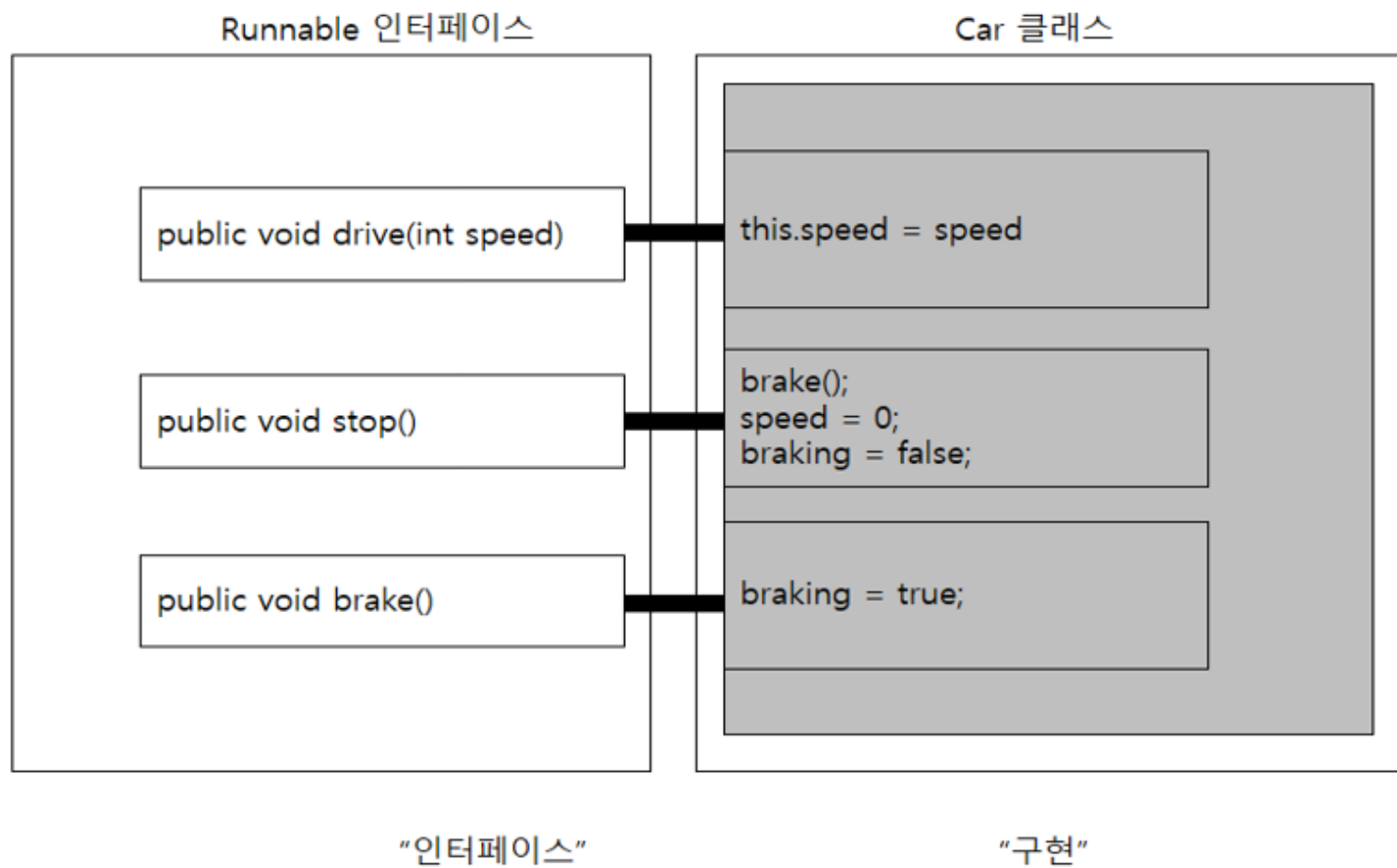
인터페이스 사용

```
interface Runnable {  
    public void drive(int speed);  
    public void stop();  
    public void brake();  
}
```

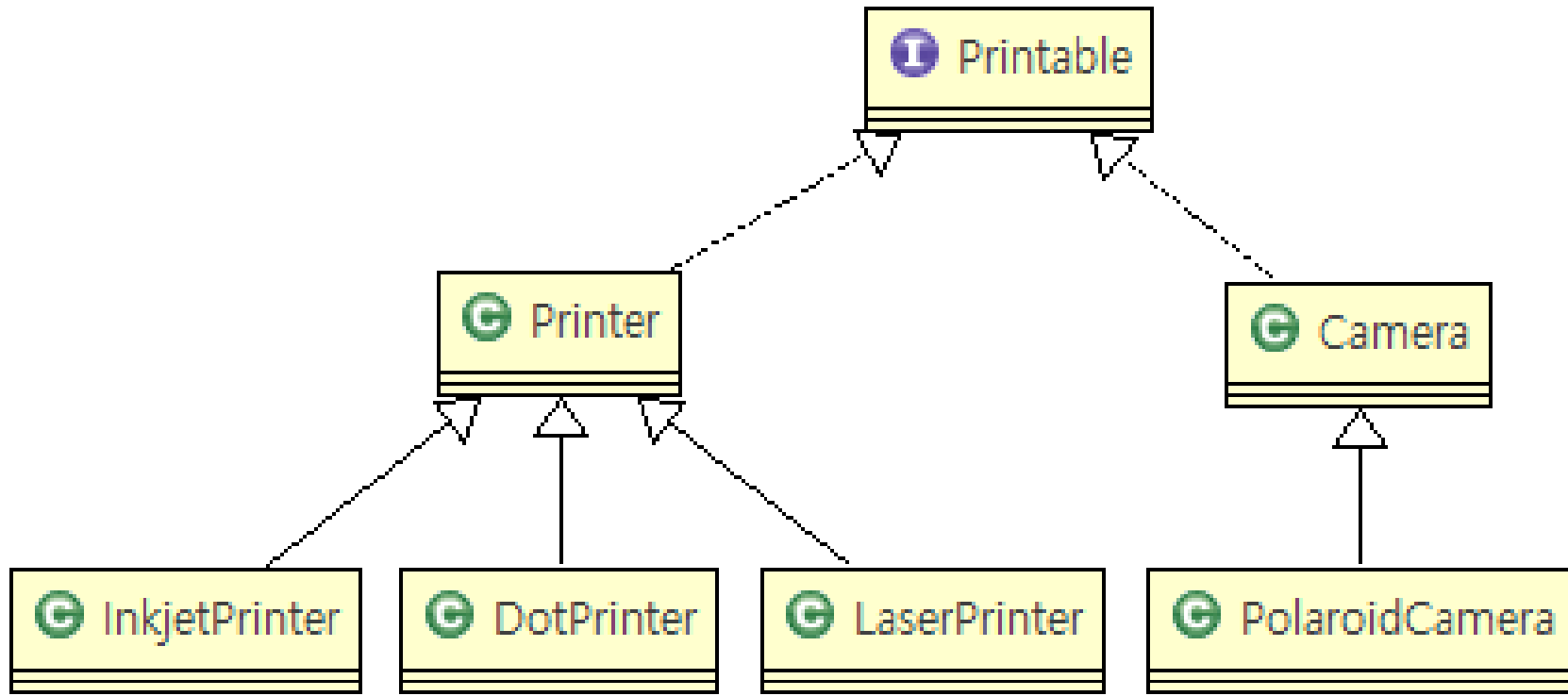
인터페이스를 사용하면 클래스에서 인터페이스와 구현을 분리할 수 있다.

자바의 인터페이스는 메소드의 시그니처만을 선언하며 메소드의 구현 코드는 빠져있다.

```
class Car implements Runnable {  
    private int speed;  
    private boolean braking;  
  
    public void drive(int speed) {  
        this.speed = speed;  
    }  
    public void stop() {  
        brake();  
        speed = 0;  
        braking = false;  
    }  
    public void brake() {  
        braking = true;  
    }  
}
```



자바의 인터페이스는 객체의 인터페이스와 구현을 분리하는 역할을 한다. Runnable 인터페이스를 구현한 클래스(Car)가 어떤 클래스인지 몰라도, 인터페이스만으로 객체의 기능을 사용할 수 있어 구체적인 클래스에 의존하지 않는 유연한 프로그램을 만들 수 있다.



인터페이스는 상속과는 다른 개념으로, 서로 다른 타입의 객체들을 동일한 타입처럼 다루고 싶을 때 사용한다. 예를 들어 Printer와 Camera는 전혀 다른 클래스지만, 같은 인터페이스를 구현하면 공통된 메시지를 보내는 다형성을 실현할 수 있다. 반면, 클래스 상속은 'IS-A' 관계가 성립될 때 사용하는 것이다.

100% 순수 추상 클래스 VS 인터페이스

- 추상 클래스와 인터페이스는 구조는 비슷하지만, 쓰임새와 의미는 다르다.
 - 추상 클래스는 'IS-A' 관계를 전제로 한 공통 기능을 모은 상속 계열을 구성할 때 사용한다.
 - 인터페이스는 서로 다른 계열의 클래스라도 공통된 동작이 필요할 때 사용한다.
- Printer는 모든 프린터의 상위 클래스(패밀리), Printable은 프린터 외에도 인쇄 가능한 다양한 객체를 묶기 위한 행동 기반 타입이다. 따라서 개발자의 의도와 문맥에 따라 적절히 선택해야 한다.

```
abstract class Printer {  
    public abstract void print(String message);  
}
```

```
interface Printable {  
    public void print(String message);  
}
```


상속(Inheritance)과 합성(Composition)

- 상속은 기능을 물려받아 확장하는 것이고, 합성은 다른 객체에게 역할을 위임하는 것이다. 상황에 따라 합성을 사용하면 더 유연하고 유지보수하기 좋은 코드를 만들 수 있다.

상속(Inheritance)

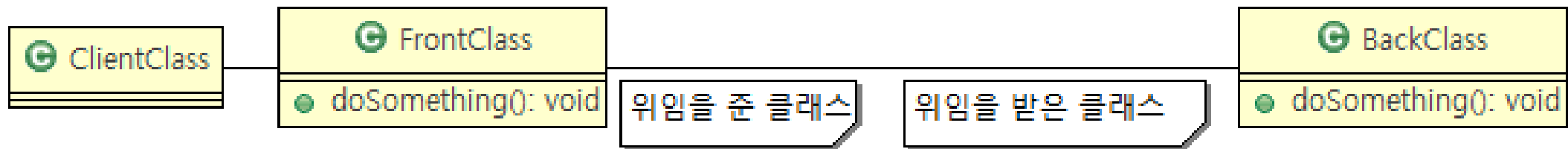
- 상속의 재사용
 - 부모 클래스의 변수와 메소드를 자식 클래스가 재사용하며, 오버라이딩과 메소드 추가로 기능을 확장할 수 있다.
- 다형성의 구현
 - 상속을 통해 프로그램에서 다형성을 구현할 수 있으며, 이는 소스 재사용 이상의 의미를 가진다.
- 유연한 프로그램 설계
 - 다형성을 사용하면 요구사항을 쉽게 추가, 반영할 수 있는 유연한 프로그램을 설계할 수 있다.
- 동적 바인딩
 - 하위 클래스의 메소드를 오버라이딩하고, 상위 클래스 타입으로 객체를 할당한 후 메소드를 호출하면, 컴파일 시간이 아닌 실행 시점에 메소드가 결정된다. 이를 동적 바인딩이라고 한다.

상속 관계 규칙

- 상속은 클래스들끼리 영구적으로 'IS-A' 관계가 성립될 때 사용한다.
- 상속이라는 의미가 통하지 않음에도 단순히 기존에 있는 클래스의 속성과 메소드를 재사용하기 위해 상속을 사용하는 것은 바람직한 일이 아니다. 이때는 합성을 사용하는 것이 좋다.
- 다형성을 구현하기 위해 상속을 강제로 사용하는 것 역시 바람직하지 않다. 이때는 합성과 인터페이스를 같이 사용하면 해결할 수 있다.

합성(Composition)

- 합성은 객체가 자신의 기능을 직접 수행하지 않고, 다른 객체의 메소드를 사용해 작업을 위임(Delegation)하는 설계 방식이다. 서로 대등한 관계에서 기능을 결합한다는 점이 특징이다.
- 클라이언트는 FrontClass만 알고, 실제 작업은 BackClass가 수행하지만 그 존재는 모른다. 즉, FrontClass가 중간 역할을 하며 내부 구현을 숨긴 구조다. 이는 캡슐화와 의존성 숨기기(추상화)의 예이다.



```
class BookShelf extends Vector<Book> {  
    public void addBook(Book book) {  
        add(book);  
    }  
    public Book getBook(int index) {  
        return get(index);  
    }  
    public void removeBook(Book book) {  
        remove(book);  
    }  
}
```

❌ 왜 Vector를 상속하면 안 될까?

1. 불필요한 기능까지 상속됨

- BookShelf는 책을 보관하는 용도로만 사용되기를 원하지만, Vector를 상속함으로써 Vector의 모든 메소드가 BookShelf에 그대로 노출된다.
- 예: addAll(), removeElementAt(), insertElementAt() 같은 의도하지 않은 메소드도 사용 가능해져서 내부 구조가 무너질 수 있다.
- 즉, 클래스가 원하지 않는 API까지 외부에 공개된다 → 캡슐화 실패

❌ 왜 Vector를 상속하면 안 될까?

2. is-a 관계가 성립하지 않음

- 상속은 "is-a" 관계일 때 사용해야 한다.
- 질문: BookShelf는 Vector<Book>인가?
→ 아니다. BookShelf는 책을 저장하는 "기능"을 가진 도메인 객체일 뿐, 일반적인 Vector가 아니다.
- BookShelf는 "Vector를 사용하여 구현된 책 저장소"이지, "Vector 그 자체"는 아님.

3. 부모 클래스 변경에 취약

- Vector 클래스의 내부 구현이나 메소드 시그니처가 바뀌면 BookShelf도 직접 영향을 받는다.
- 즉, 상속 구조는 결합도가 높아진다 → 변경에 유연하지 않음

4. 책임이 모호해짐

- BookShelf가 Vector를 상속받음으로써 책 관리 외에도 Vector로서의 책임(자료구조 처리)까지 떠안게 됨 → 단일 책임 원칙(SRP) 위배

✅ 그럼 어떻게 해야 할까?

상속이 아닌 "합성(composition)"을 사용해야 한다.

cleancode.section03.inheritanceORdelegation0.InheritanceToDelegation2

```
class BookShelf {  
    private Vector<Book> bookShelf;  
    public BookShelf() {  
        bookShelf = new Vector<Book>();  
    }  
  
    public void addBook(Book book) {  
        bookShelf.add(book);  
    }  
    public Book getBook(int index) {  
        return bookShelf.get(index);  
    }  
    public void removeBook(Book book) {  
        bookShelf.remove(book);  
    }  
}
```

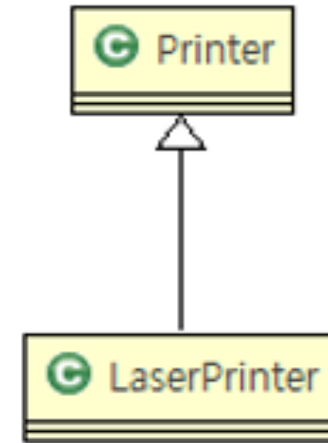
이렇게 하면 BookShelf는 Vector를 내부 구현에만 사용하고, 외부에 자신만의 인터페이스만 노출하게 된다.

→ 유지보수, 변경, 테스트가 더 쉬워지고 의미 있는 추상화를 할 수 있다.


상속(Inheritance)

```
public abstract class Printer {  
    private String id;  
    public String getId() {  
        return id;  
    }  
    abstract void print(Object message);  
}
```

```
public class LaserPrinter extends Printer {  
    public void print(Object message) {  
        System.out.println(message);  
    }  
}
```



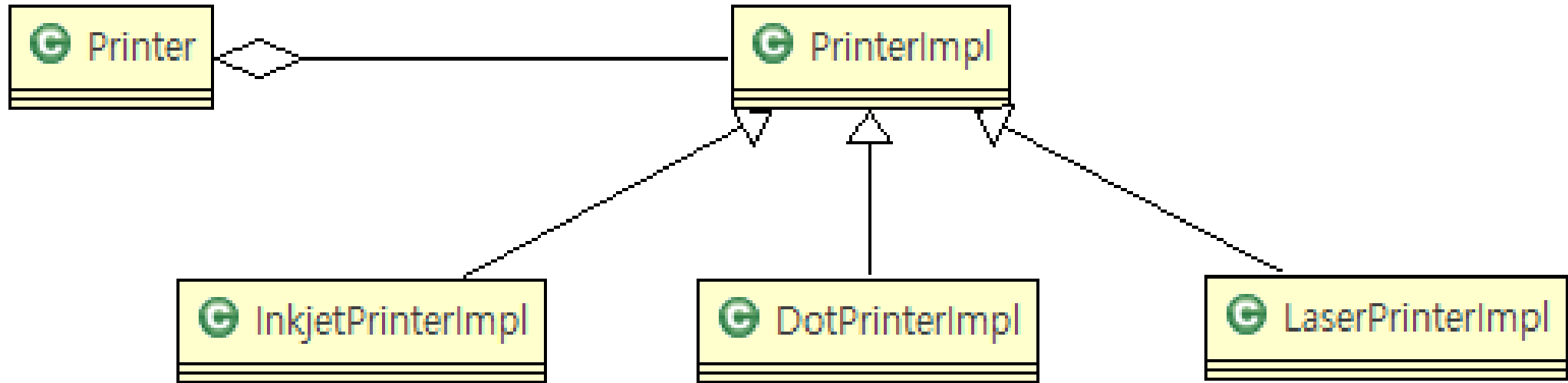
Person의 getId() 리턴타입이 String에서 int로 변경된다면




```
if ( printers[i].getId().equals("101") ) {  
    if ( printers[i].getId() == 101 ) {
```

상속은 부모와 자식 클래스 간에 강한 결합을 만들어, 부모 클래스의 메소드 시그니처가 변경되면 자식 클래스를 사용하는 모든 클라이언트 코드에 영향을 미친다.

합성(Composition)



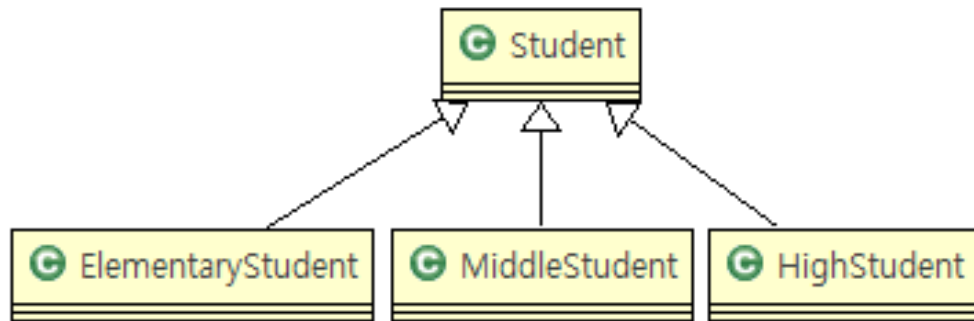
```
public class Printer {  
    private String id;  
    private PrinterImpl printerImpl;  
  
    public Printer(PrinterImpl printerImpl) {  
        this.printerImpl = printerImpl;  
    }  
  
    public String getId() {  
        return printerImpl.getId();  
    }  
  
    public void print(Object message) {  
        printerImpl.print(message);  
    }  
}
```

```
abstract public class PrinterImpl {  
    private String id;  
  
    public String getId() {  
        return id;  
    }  
     return Integer.toString(printerImpl.getId());  
  
    abstract public void print(Object message);  
}  
  
public class LaserPrinterImpl extends PrinterImpl {  
    public void print(Object message) {  
        System.out.println(msg.toString());  
    }  
}
```

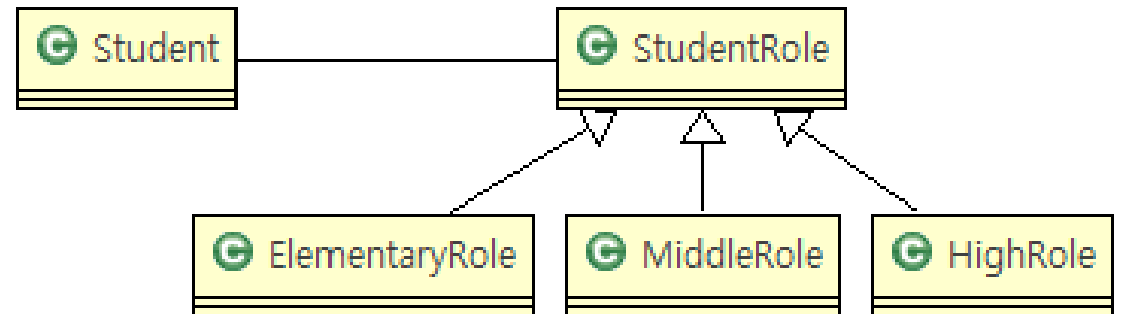
상속보다 합성이 좋은 이유

- 합성은 시간이 지나며 변할 수 있는 역할(Role)을 표현할 때 적합하다.
- 상속은 본질적으로 변하지 않는 항구적인 IS-A 관계를 표현할 때 사용한다.
 - ✓ 남학생/여학생 → 학생: 변하지 않는 본질 → 상속 사용
 - ✓ 초등학생 → 중학생 → 고등학생: 시간이 지나며 변함 → 합성 사용
- 합성을 쓰면 역할이 바뀔 때마다 객체 내부의 역할만 교체하면 되므로 유지보수와 확장성이 높아진다.

상속보다 합성이 좋은 이유



상속(✖)



합성(○)

CaseStudy01 상속과 위임 사용 실습

section04 소프트웨어의 변화와 유지보수성

소프트웨어 품질 기준

유지보수성의 하위 특성

소프트웨어 품질 평가 기준

- 기능성(Functionality) – 사용자가 원하는 기능이 오류 없이 잘 작동하는 정도
- 안정성(Reliability) – 성능을 유지하며 문제 없이 작동하는 정도
- 사용 편의성(Usability) – 사용자가 쉽게 사용할 수 있는 정도
- 효율성(Efficiency) – 자원을 적게 쓰면서 기능을 수행하는 정도
- **유지보수성(Maintainability)** – 수정이나 기능 추가가 얼마나 쉬운지
- 이식성(Portability) – 다른 환경에서도 쉽게 사용할 수 있는지

유지보수성과 디자인 패턴, 리팩토링

- 유지보수성(Maintainability)이란?
 - 소프트웨어 품질 기준에서 가장 중요
 - 요구사항이 변경될 때, 기존 코드를 수정, 개선하는 데 필요한 노력의 정도를 의미
 - 노력이 적게 드는 소프트웨어일수록 유지보수성이 높다
 - 고객의 요청에 빠르게 대응할 수 있는 소프트웨어 = 좋은 소프트웨어
- 디자인 패턴
 - 유지보수성이 좋은 소프트웨어의 내부 구조를 설계할 때 사용
- 리팩토링
 - 이미 만든 코드의 유지보수성을 향상시키기 위한 코드 구조 개선 활동

코드를 수정해야 하는 이유

- 기존 코드의 개선

정상적으로 동작하는 코드들을 수정함으로 소프트웨어의 유지보수성을 높이거나 성능을 개선한다.

- 오류 수정

소프트웨어가 의도한 기능 대로 동작하지 않으면 그 원인을 찾아내어 제거한다.

- 고객의 새로운 요구사항 반영

소프트웨어를 주문한 고객, 또는 시장에서 원하는 요구사항들을 반영하도록 코드를 변경한다. 주로 새로운 기능을 추가하거나 기존 기능을 삭제, 변경한다.

유지보수성의 5가지 하위 특성

특성명	설명
분석 용이성 (Analyzability)	문제 원인과 수정 위치를 쉽게 파악할 수 있는 정도
변경 용이성 (Changeability)	코드를 수정하는 데 필요한 노력의 정도
안정성 (Stability)	수정 시 예기치 못한 오류 발생 가능성
테스트 검증성 (Testability)	수정 후 정상 작동을 테스트할 필요성
규칙 준수성 (Compliance)	표준과 규칙을 얼마나 잘 준수하는가

디자인패턴과 리팩토링은 변화에 유연하게 대응할 수 있는 소프트웨어를 만들기 위한 핵심 도구

객체 지향 코드의 수정

- 코드를 추가하는 방식
 - 기존 코드를 변경하며 추가한다.
 - 기존 코드를 그대로 유지하면 새롭게 추가한다.
- 코드를 추가함으로써 변경해야 하는 코드의 위치
 - 추가된 기능을 사용하는 클라이언트 코드(Client Code)
 - 코드 추가로 인해 영향을 받는 곳(Side Effect)
- 상속을 통한 확장의 장점
 - 기존 코드 변경보다 상속을 통해 새로운 클래스를 정의하는 것이 오류 발생 가능성을 줄임.
 - 변경할 코드의 양이 적을수록 작업이 수월함.

객체 지향 코드의 수정

- 유지보수성 높은 소프트웨어
 - 기존 클래스를 유지하고 확장을 통해 새로운 클래스를 정의하는 방식이 바람직함.
 - 변경 작업을 수행할 때, 변경된 클래스의 객체를 사용하는 클래스(Client Code)는 일관성이 있어야 함.
 - 변경해야 하는 코드(Side Effect)의 분량은 적을수록 좋음.
 - 어쩔 수 없이 변경해야 하는 부분은 최소화되어야 함.

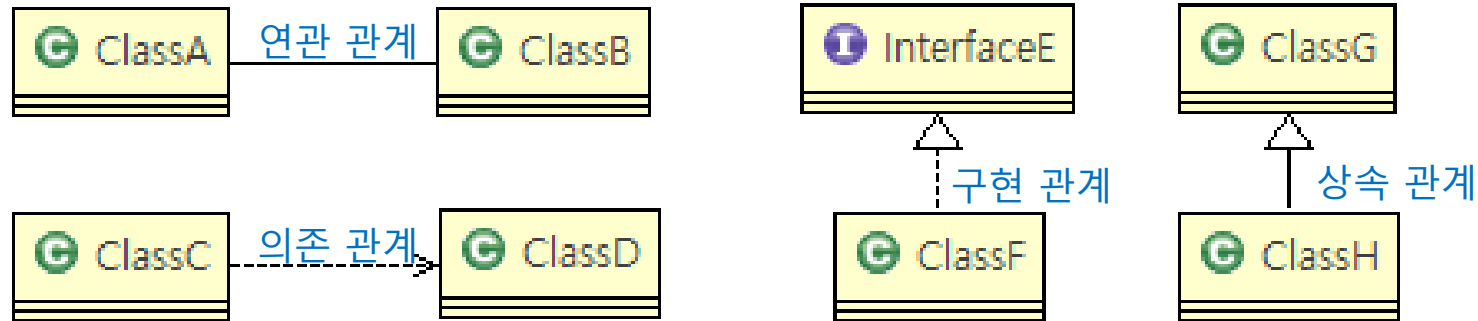
인터페이스와 메소드 내부 변경

- 기존 인터페이스 변경
 - 메소드 시그니처가 변경되는 경우
- 기존 클래스의 메소드 내부 변경
 - 정상적으로 잘 동작하는 클래스의 메소드 내부 코드를 수정해야 하는 경우
- 변화를 포용하는 객체지향 프로그램은 기존 인터페이스 변경의 영향을 최소화한다
- 기존 클래스를 수정하지 않고 확장하여 새로운 기능을 추가하는 방식을 지향한다.

클래스나 인터페이스 변경 시 주의점

- 프로그램 개발 후 변경이 필요한 상황
 - 성능 개선
 - 새로운 기능 추가
- 메소드 시그니처 변경의 위험
 - 클래스나 인터페이스의 메소드 시그니처(이름, 매개변수 등)를 변경하면 관련된 클라이언트 코드 전체에 영향을 미침
- 필요한 작업
 - 변경된 메소드와 연관된 모든 코드를 찾아서 수정
 - 누락 시 컴파일 오류 또는 런타임 오류 발생 가능
- 한 곳의 변경이 여러 곳에 영향을 미친다면, 그 코드는 유지보수에 취약하다.
→ 따라서 변경을 최소화할 수 있는 설계가 중요하다.

기존 인터페이스 변경



- 문제점
 - 클래스간의 강한 의존 관계 → 변경 전파 범위를 넓힘
 - 유지보수가 어려워지고, 오류 가능성 증가
- 설계 시 유의할 점
 - 클래스는 기능 수행에 필요한 범위 내에서만 다른 클래스와 의존하거나 연관되어야 함
- 핵심 설계 원칙
 - 클래스 간의 관계는 최소화해야 변경의 영향을 줄일 수 있다.(LoD(Law of Demeter, 디메터 법칙))

기존 메소드 내부의 변경

- 기능 추가 시마다 기존 클래스의 메소드를 반복해서 수정해야 하는 경우, 유지보수가 어렵다.
- 특히 분기문(if/switch)으로 프린터 종류를 구분하는 방식은, 새로운 프린터 추가 시마다 print() 메소드를 수정해야 한다.

```
public class PrinterTest {  
    public static void main(String[] args) {  
        Printer printer = new Printer("DOT");  
        printer.print("샘플 출력...");  
    }  
}
```

```
class Printer {  
    private String type;  
    public Printer(String type) {  
        this.type = type;  
    }  
    public void print(String message) {  
        if(type.equals("DOT")) {  
            dotPrint(message);  
        } else if(type.equals("INKJET")) {  
            inkjetPrint(message);  
        }  
    }  
    public void dotPrint(String content) {  
        System.out.println("도트 프린터 : " + message);  
    }  
    public void inkjetPrint(String content) {  
        System.out.println("잉크젯 프린터 : " + message);  
    }  
}
```

```
abstract class Printer {  
    public abstract void print(String content);  
}  
  
class DotPrinter extends Printer {  
    public void print(String content) {  
        System.out.println("도트 프린터 : " + message);  
    }  
}  
  
class InkjetPrinter extends Printer {  
    public void print(String content) {  
        System.out.println("잉크젯 프린터 : " + content);  
    }  
}
```

기존 메소드 내부의 변경

- 새로운 프린터 종류를 추가할 때 상속만 하면 되므로 확장에 유리하다.
- 이전 코드는 기존 print() 메소드를 계속 수정해야 하므로 실수로 다른 로직에 영향을 줄 수 있어 에러 발생 가능성이 높다.
- 상속의 방식은 변화에 더 유연하고 안전한 설계이다.

```
public class PrinterTest {  
    public static void main(String[] args) {  
        Printer printer = new DotPrinter();  
        printer.print("샘플 메시지");  
    }  
}
```

section05 소프트웨어 설계 원칙

결합도

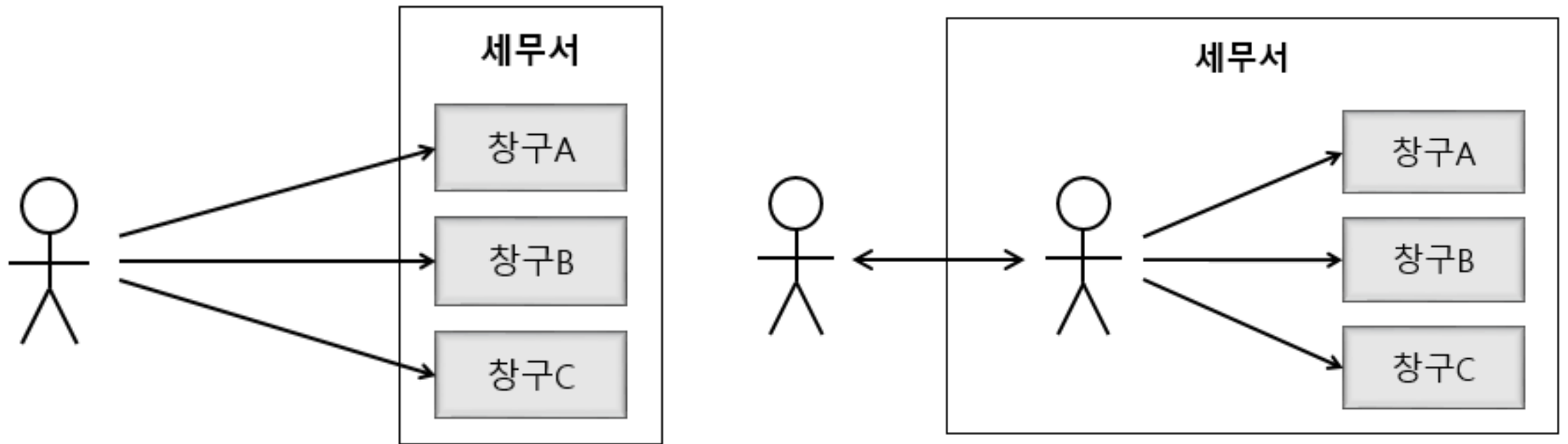
응집도

OAOO(Once And Only Once)

결합도(Coupling)

- 유지보수성이 높은 소프트웨어의 가장 중요한 특징은 프로그램의 각 요소들이 결합도는 낮게, 응집도는 높게 구성해야 한다는 것이다.
- **결합도**란 소프트웨어의 한 요소가 다른 것과 얼마나 강력하게 연결되어 있는지, 또한 얼마나 의존적인지를 나타내는 정도다.
- 프로그램의 요소가 결합도가 낮다는 것은 그것이 다른 요소들과 관계를 그다지 맺지 않는 상태를 의미한다.
- 결합도가 높으면 과도하게 많은 다른 요소들과 얽히게 된다.

결합도(Coupling)



사용자에게 복잡한 절차를 감추고, 직원이 대신 처리함으로써 세무 업무를 쉽게 만든 구조이다.

결합도(Coupling)가 높을 경우

- 변경에 취약함→ 다른 클래스에 의존하고 있어, 해당 클래스가 변경되면 같이 수정해야 한다.
- 재사용 어려움→ 다른 클래스와 강하게 연결되어 있어, 독립적으로 재사용하기 힘들다.
- 이해하기 어려움→ 다양한 클래스와 복잡하게 얽혀 있어서 전체 구조를 파악하기 어렵다.
- 테스트 어려움→ 의존성이 많아 단위 테스트를 하려면 관련 클래스들도 함께 준비해야 한다.
- 확장에 제약→ 구조가 단단히 엮여 있어 새로운 기능을 추가하거나 수정하기 어렵다.
- 결합도가 높을수록 유지보수성과 유연성이 떨어진다.

낮은 결합도(Coupling)를 위해

- 결합도는 낮을수록 좋다→ 클래스 간 의존성이 적을수록 바람직하다.
- 불필요한 연관은 피해야 한다→ 꼭 필요한 클래스와만 연결되어야 한다.
- 개발자는 결합도를 낮추도록 노력해야 한다→ 구조 설계 시 낮은 결합도를 목표로 해야 한다.
- 낮은 결합도를 위해 클래스는 꼭 필요한 대상과만 연결되고, 불필요한 의존은 제거해야 한다.

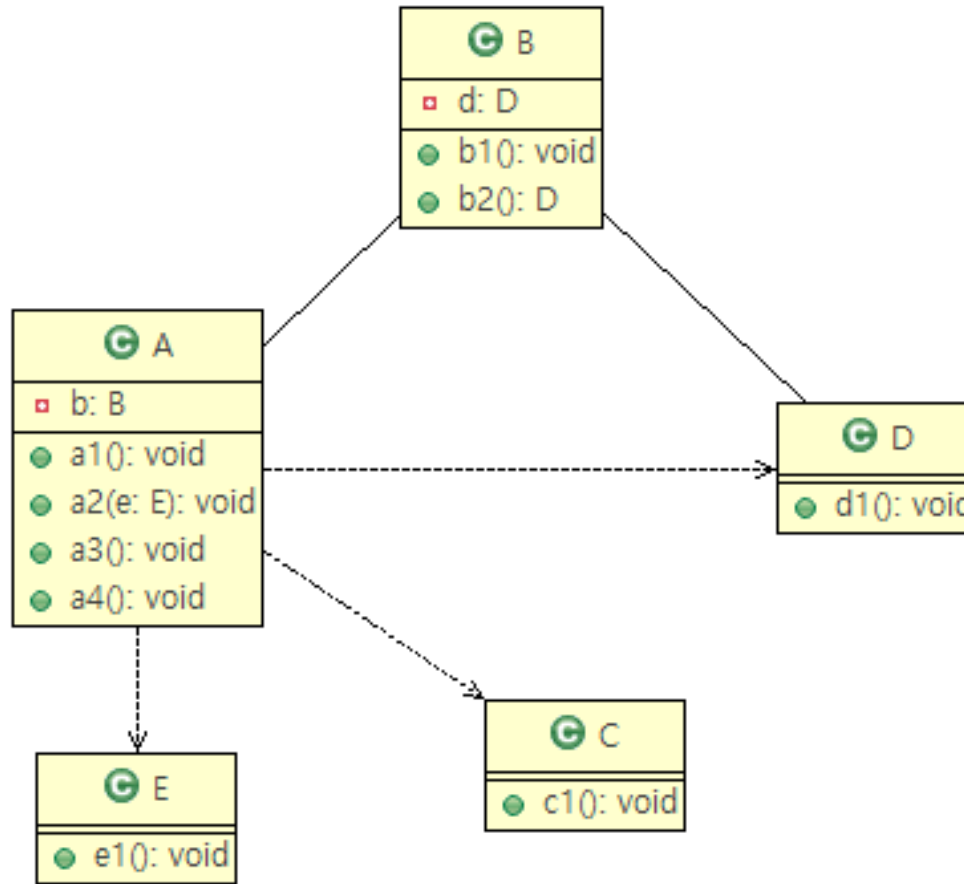
결합 형태

- 속성 객체 결합 (연관 관계) → 다른 객체를 멤버 변수로 갖고 그 객체의 메소드를 사용함.
- 로컬 객체 결합 (의존 관계) → 메소드 안에서 다른 객체를 직접 생성해 사용함.
- 파라미터 객체 결합 (의존 관계) → 메소드가 파라미터로 받은 객체의 메소드를 호출함.
- 반환 객체 결합 (의존 관계) → 다른 객체로부터 리턴받은 객체를 통해 작업함.
- 상속 결합 (상속 관계) → 부모 클래스를 상속받아 기능을 확장함.
- 인터페이스 결합 (구현 관계) → 인터페이스를 구현하여 기능을 정의함.
- 결합은 객체를 어떻게 참조하고 사용하는지에 따라 여러 형태로 나뉘며, 의존성의 방향과 강도에 따라 구조 설계에 영향을 미친다.

```
package cleancode.section05.coupling;
```

```
class A {  
    private B b;  
    // 속성 객체 결합  
    public void a1() {  
        b.b1();  
    }  
    // 파라미터 객체 결합  
    public void a2(E e) {  
        e.e1();  
    }  
    // 로컬 객체 결합  
    public void a3() {  
        C c = new C();  
        c.c1();  
    }  
    // 반환 객체 결합  
    public void a4() {  
        b.b2().d1();  
    }  
}
```

```
class B {  
    private D d;  
    public void b1() {  
    }  
    public D b2() {  
        return d;  
    }  
}  
  
class C {  
    public void c1() {}  
}  
  
class D {  
    public void d1() {}  
}  
  
class E {  
    public void e1() {}  
}
```



- 클래스 A가 여러 클래스(B, C, D, E)와 다양한 방식으로 결합되어 있다.
- 결합도(Coupling) 측면에서 바라볼 때, 클래스 A는 지나치게 많은 클래스에 의존하고 있어 유지보수성과 확장성이 떨어지는 문제가 있다.

반환 객체 결합

```
class A {  
    ...  
    // 반환 객체 결합  
    public void a4() {  
        b.b2().d1();  
    }  
}  
class B {  
    private D d;  
    public void b1() {  
    }  
    public D b2() {  
        return d;  
    }  
}  
class D {  
    public void d1() {}  
}
```

- 반환 객체란?
 - 어떤 객체가 다른 객체의 내부에 있는 객체에 직접 접근하는 것.
- 반환 객체 결합이 위험한 이유
 - A가 D를 사용하는 경우 D 내부 구조를 알아야 함 → 캡슐화 위반
 - D 내부 구조가 바뀌면 A 코드도 변경 해야함 →, 변경 전파
 - 한곳을 바꾸면 전체 시스템에 영향을 줌 → 유지보수 어려움
- 해결 방향
 - 반환 객체 결합은 가능한 제거하는 것이 좋음
 - LoD (Law of Demeter, 디메터 법칙) 적용

```
package cleancode.section05.coupling;
```

```
class A {  
    ...  
    public void a4() {  
        b.b3();  
    }  
}  
  
class B {  
    private D d;  
    public void b1() {  
    }  
    public D b2() {  
        return d;  
    }  
    public void b3() {  
        d.d1();  
    }  
}  
  
class D {  
    public void d1() {}  
}
```

- 수정 전 코드

- A는 D 객체를 리턴받아 직접 사용
- A는 B와 D 동시 결합 → 높은 결합도, 낮은 캡슐화

- 수정 후 코드

- A는 B에게 행위를 요청
- D는 B 내부에서만 사용
- A는 D를 직접 사용하지 않음 → 결합도 낮아지고 캡슐화 향상

응집도(Cohesion)

- 응집도란?

- 프로그램 요소가 기능 수행에 꼭 필요한 책임만 갖는 정도

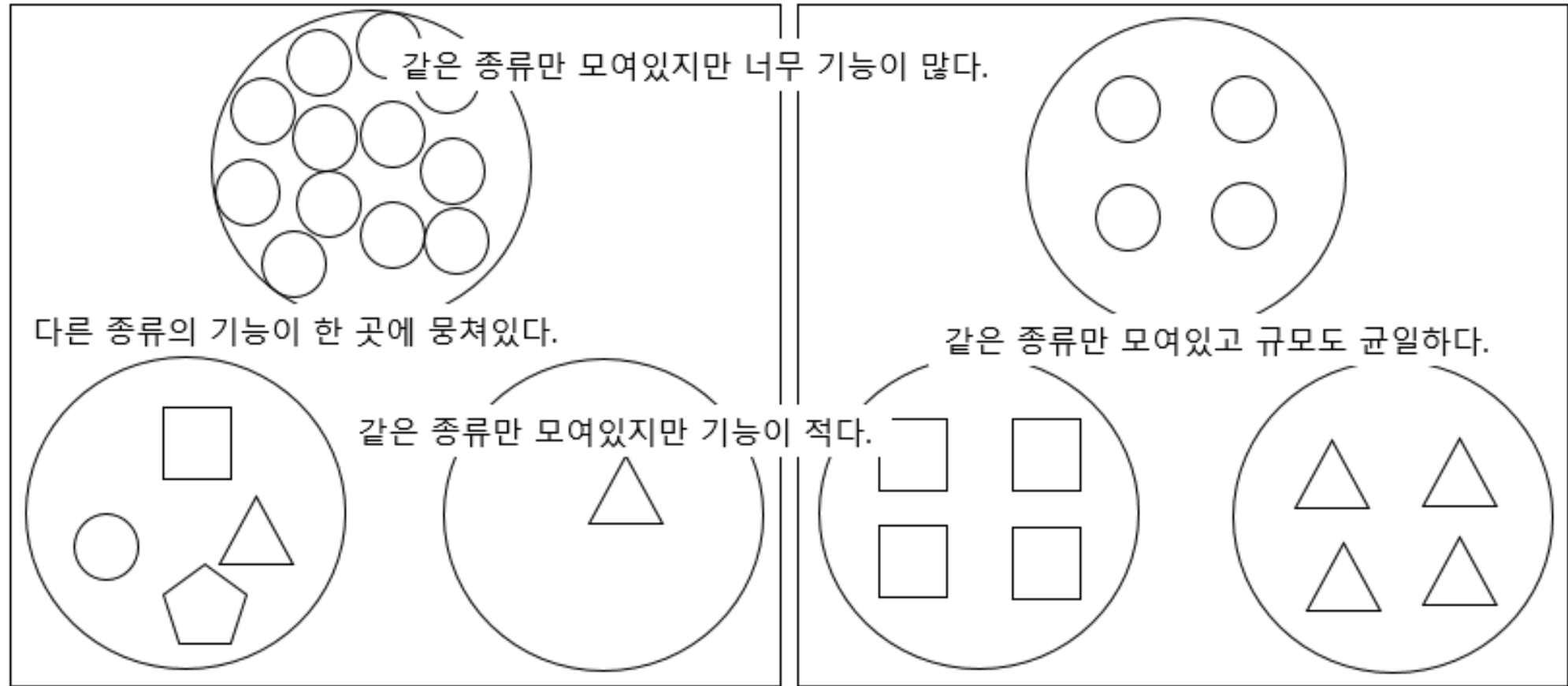
- 응집도가 높다는 의미

- 특정 목적에 맞는 기능들만 포함되고, 불필요한 작업이 없음

- 응집도가 높을 때 장점

- 이해하기 쉽고, 유지보수가 쉬운 코드가 된다

응집도(Cohesion)



<응집도가 낮은 경우>

<응집도가 높은 경우>

응집도의 중요성과 설계 원칙

- 응집도가 높으면 기능 추가, 변경, 삭제가 쉬워지고 구조가 직관적이다.
- 클래스의 메소드들이 하나의 목적에 집중되어 있어야 높은 응집도를 가진다.
- 응집도가 낮으면 유지보수가 어려워지고, 관리가 복잡해진다.
- 클래스는 책임에 맞는 기능만을 포함하고, 과도한 역할을 피해야 한다.

응집도 높은 클래스의 조건

- 클래스는 같은 목적의 기능을 가진 메소드들로 구성되어야 한다.
- 각 메소드는 단일 기능만 수행하고, 개수가 적당해야 한다.
- 너무 많은 일을 혼자 하지 말고, 다른 클래스와 역할을 나눠야 한다.
- 메소드가 너무 크다면 별도의 클래스로 분리하는 것도 고려해야 한다.

응집도가 낮은 클래스의 문제와 개선 방안

- 어떤 클래스가 응집도가 낮을까?
 - 여러 기능 영역의 메소드를 함께 포함함
 - 클래스가 하나의 책임에 집중하지 못함
- 예시 : 잘못 설계된 클래스
 - ImageAndSoundProcessing 클래스
 - 이미지와 소리 처리가 뒤섞여 있음
 - 조건에 따라 이미지 or 소리를 처리
- 결과적으로
 - 가독성 저하
 - 변경 시 오류 가능성 증가
 - 유지 보수 어려움
- 응집도 높이는 방법
 - 기능별로 클래스 나눔
 - ImageProcessing
 - SoundProcessing
- 메소드 정리
 - 기능에 따라 메소드도 역할에 맞게 정리
- 핵심
 - 클래스가 하나의 책임에 집중할수록 응집도가 높아지고, 설계는 더 깔끔하며 유지보수가 쉬워진다.

응집도가 낮은 클래스의 문제와 개선 방안

- 문제 상황

- Payment 클래스가 현금, 수표, 신용카드 결제 기능을 모두 직접 처리하도록 구현됨
- 다양한 결제 방식이 뒤섞여 있어 응집도 낮음

- 개선 방향

- 공통된 결제 기능은 Payment 클래스에 정의
- 각 결제 방식은 자식 클래스로 분리→ Cash, Check, CreditCard 등
- 역할 분담을 통해 응집도 향상

응집도 vs 결합도 – 객체지향 설계의 핵심 원리

- 응집도: 클래스 내부의 책임이 하나로 집중
- 결합도: 클래스 간의 관계가 느슨할수록 좋음
- 응집도와 결합도는 서로 다른 개념이지만 설계 품질에 모두 중요함
- 상호 보완적이기도 하고 서로 충돌할 때도 있음
- 좋은 설계란, 응집도와 결합도의 균형을 잘 맞추는 것
- 개발자에게 필요한 사고
 - 한 가지 원칙만 고집하지 말고
 - 두 개념을 함께 고려하여 유연하게 설계해야 한다

OAAO(Once And Only Once) 원칙

- OAAO 원칙이란
 - OAAO(Once And Only Once) = 한 번만 작성하라
 - 중복된 코드나 기능이 여러 곳에 존재하면 유지보수성 급격히 저하
- OAAO 원칙의 핵심 개념
 - 소프트웨어의 모든 요소는 중복 없이 한 군데에서만 정의되어야 함
- 중복
 - 물리적 중복: 완전히 같은 코드가 여러 곳에 존재
 - 논리적 중복: 다르지만 같은 기능을 수행하는 코드가 여러 곳에 흩어짐
- 중복의 결과:
 - 코드 관리 어려움
 - 책임 분산 → 이해도 저하
 - 변화에 대한 저항 증가

OAOO(Once And Only Once) 원칙

- OAOO 원칙의 효과
 - 중복 제거 → 책임 명확화
 - 한 곳에 집중 → 변경 용이
 - 가독성 ↑ 유지보수성 ↑
 - 프로그램의 품질과 일관성 향상
- OAOO 실현의 예 - AOP
 - AOP (Aspect Oriented Programming): OAOO를 극단적으로 실현
 - 반복되는 기능을 외부의 독립 클래스로 분리
 - 예외 처리, 보안, 트랜잭션 처리등
 - 핵심 로직과 분리하여 유지보수성 향상
 - Spring Framework는 AOP를 가장 널리 지원하는 프레임워크

결합도 ↓ 응집도 ↑ OAOO = 유지보수성 ↑

- 변경해야 하는 코드의 위치를 전체 소스코드에서 잘 찾을 수 있을까?
 - 관련 기능이 모여 있고 중복이 없어, 수정할 부분이 명확하다.
- 코드를 쉽고 빠르게 수정할 수 있을까?
 - 중복이 없어 한 곳만 수정하면 되고, 클래스 책임이 나뉘어 있어 확장도 쉽다.
- 코드를 수정한 이후에도 시스템이 안정적으로 동작할 확률이 높아질까?
 - 결합도가 낮아 변경이 다른 부분에 미치는 영향이 적다.
- 코드를 수정한 이후에 그 부분과 전체 시스템이 제대로 동작하는지 테스트를 쉽게 할 수 있을까?
 - 결합도가 낮아 수정하지 않은 부분은 독립적으로 테스트하기 쉽다.

section06 객체지향 설계 원칙

ORR(One Response Rule) 원칙

OCP(Open Closed Principle) 원칙

LoD(Law of Demeter) 원칙

ORR(One Responsibility Rule) 원칙

- ORR은 클래스나 메소드가 자신에게 부여된 한 가지 고유한 책임만을 수행해야 한다는 원칙이다.
- 이 원칙은 높은 응집도와 밀접하게 관련되어 있기때문에 이 원칙을 따르면 자연스럽게 프로그램의 응집도가 높아진다.
- ORR 원칙을 풀어서 설명하면 다음과 같다.
 - 그 일을 자신만이 유일하게 해야 하며,
 - 그 일을 다른 클래스의 메소드보다 더 잘 할 수 있어야 한다.
 - 그리고 그 책임에 해당하는 일을 빠짐없이 모두 해야 한다.

ORR(One Responsibility Rule) 원칙 위배

- 과도한 기능 집중
 - 하나의 클래스에 너무 많은 기능이 몰림
 - 불필요한 코드 증가 → 기능 추가/수정 어려움
- 협업 및 형상 관리 어려움
 - 여러 개발자가 동시에 수정 → 버전 충돌 발생
 - CVS/SVN 같은 형상 관리 도구 사용이 복잡해짐
- 성능 저하
 - 객체가 무거워져 메모리 낭비
 - 공통 기능의 반복 호출로 시스템 성능 저하
- 해결 방안
 - 기능을 관리 가능한 단위로 분리
 - 역할별로 클래스를 나누고 위임
 - 코드의 일관성 유지

ORR(One Responsibility Rule) 원칙

G BoardAndUserDAO
<ul style="list-style-type: none">● insertBoard(vo: BoardVO): void● updateBoard(vo: BoardVO): void● deleteBoard(vo: BoardVO): void● getBoard(vo: BoardVO): BoardVO● getBoardList(vo: BoardVO): List<BoardVO>● insertUser(vo: UserVO): void● update(vo: UserVO): void● deleteUser(vo: UserVO): void● getUser(vo: UserVO): UserVO● getUserList(vo: UserVO): List<UserVO>

- 문제 상황

- 게시글 처리 + 회원 정보 처리 기능 모두 포함
- 두 가지 책임을 동시에 가짐

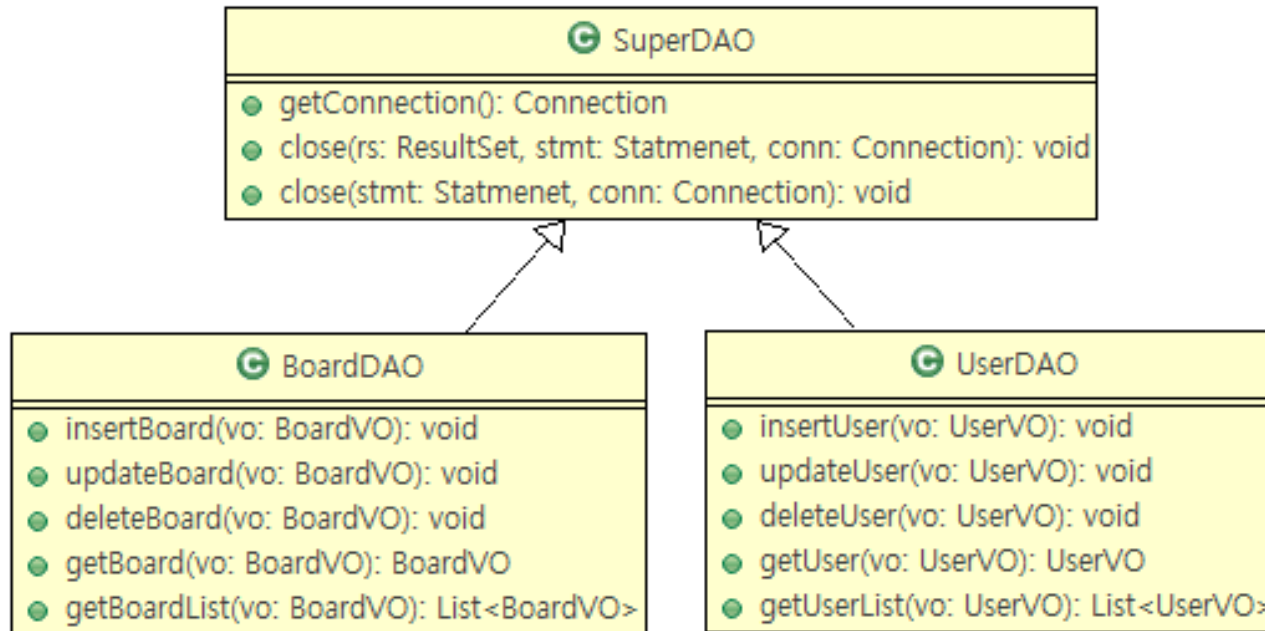
- 위반된 설계 원칙

- ORR 원칙(One Responsibility Rule, 단일 책임 원칙) 위배됨
- 하나의 클래스는 오직 하나의 책임만 가져야 함

- 점검 방법

- 클래스 내 메소드가 서로 다른 종류의 작업을 하고 있다면
→ 책임이 중복된 것
- 클래스 이름에 AND / OR가 들어간다면
→ 이미 책임이 여러 개라는 신호

ORR(One Responsibility Rule) 원칙



메서드도 한 가지 일만 해야 한다

- 문제 상황
 - 메서드에 여러 기능이 혼재되어 있음
 - 클래스 내부에서 기능 분리 필요
 - 또는 다른 클래스로 책임 이전 필요
- 복잡한 메서드의 위험
 - 개발자가 기능을 명확히 이해하지 못하는 상태에서 작성
 - 스파게티 코드로 발전 가능성 높음
 - 유지보수성, 확장성 저해
- 원인 : 잘못된 개발 습관
 - 요구사항 분석 부족
 - 잦은 요구 변경
 - 프레임워크/컴포넌트 이해 부족
 - 설계없이 먼저 코딩하는 습관
- 악순환의 구조
 - 허점 많은 코드
 - 보완하려다 또 코드 추가
 - 복잡도 증가 → 스파게티 코드
- 현실의 개발 방식
 - 분석·설계 없이 코딩부터 시작하는 방식 선호
 - 실행 결과가 바로 보이기 때문에 즉각적인 성취감 발생
→ 실제로는 문제를 미루고 복잡도를 누적시키는 셈
- 해결 방향
 - 설계를 먼저하고, 메서드는 하나의 책임만 수행하도록 구현 → 유지보수성과 확장성이 뛰어난 구조로 이어짐

복잡한 메서드 해결

- 1단계: 중복된 기능이 있는지 확인
 - 기존 컴포넌트나 프레임워크에 동일한 기능이 있는지 점검
 - 프로젝트 내에 유사한 기능이 이미 구현되어 있는지 확인
 - 불필요한 중복 개발 방지, 업무 효율성 향상, 소스코드의 일관성 유지
 - 당면한 업무에만 몰두하여 비슷한 기능이 중복 개발되는 사례가 많음
 - 협업 마인드가 해법 → 옆 사람이 무엇을 하고 있는지 관심을 갖는 것
- 2단계: 응집도 관점에서 리팩토링
 - 메서드에 직접 관련이 없는 기능은 다른 객체나 메서드로 위임
 - 불필요한 코드 제거로 메서드를 간결하고 목적에 맞게 정리
- 중복 확인 → 협업 인식 → 책임 분리 → 복잡한 메서드를 효율적이고 깔끔하게 개선

<pre>package cleancode.section06.orr;</pre>		
<pre>public class NumberSortingTest1 { public static void main(String[] args) { int[] array = { 0, 3, 5, 2, 6, 7, 8, 9, 1, 4 }; int[] sortedArray = new int[array.length]; int temp; try { if (array.length > 0) { if (sortedArray.length > 0) { if (sortedArray.length == array.length) { for (int i = 0; i < array.length; i++) { for (int k = i + 1; k < array.length; k++) { if (array[i] > array[k]) { temp = array[i]; array[i] = array[k]; array[k] = temp; } } } for (int v : array) { System.out.print(v + " "); } System.out.print("\n"); } for (int i = 0; i < array.length; i++) { sortedArray[i] = array[i]; } } } } System.out.println("최종 결과"); for (int v : sortedArray) { System.out.print(v + " "); } } catch (Exception e) { e.printStackTrace(); } }</pre>	<p>항목</p> <p>중복 로직</p> <p>중첩 조건문</p> <p>혼합된 책임</p> <p>비효율적인 출력 위치</p> <p>유지보수 어려움</p>	<p>설명</p> <p>수동 정렬, 배열 복사 직접 구현 → 불필요한 반복 코드</p> <p>if 문이 과도하게 중첩됨 → 가독성 저하</p> <p>정렬, 출력, 복사 등 여러 기능이 뒤섞여 있음</p> <p>정렬 중간마다 매번 출력 → 의미 없는 반복 출력</p> <p>수정 시 로직 추적 어려움, 버그 발생 가능성 증가</p>


```

package cleancode.section06.orr;

import java.util.Arrays;

public class NumberSortingTest {

    public static void main(String[] args) {

        int[] array = { 0, 3, 5, 2, 6, 7, 8, 9, 1, 4 };
        int[] sortedArray = new int[array.length];
        Arrays.sort(array);
        System.arraycopy(array, 0, sortedArray, 0, array.length);
        System.out.println("최종 결과");
        for (int num : sortedArray) {
            System.out.print(num + " ");
        }

    }

}

```

항목

기능 재사용

책임 분리

가독성

예외 처리

단일 책임 원칙 준수

설명

Arrays.sort()와 System.arraycopy() 사용 → 검증된 라이브러리 활용

정렬과 복사가 명확히 분리됨

전체 코드 길이 짧고 한눈에 로직 파악 가능

복잡한 조건문 없이 필요한 로직에 집중

메서드 안에서 하나의 명확한 책임 수행

OCP 원칙

- OCP (Open Closed Principle) 란?
 - 새로운 기능에는 열려 있고, 기존 코드 수정에는 닫혀 있음
 - 디자인 패턴과 다형성을 활용하여 구현
- OCP 구현 원리
 - 변화 가능성이 있는 부분은 추상 클래스나 인터페이스로 정의
 - 하위 클래스에서 기능을 구체화(override) 하여 확장
- OCP 핵심
 - 기존 코드 수정 없이 새로운 기능을 유연하게 추가할 수 있는 구조가 OCP를 따르는 설계

OCP 원칙

- 로봇의 팔이 고장난 경우

비유	설명
비모듈형 로봇	팔을 교체하려면 로봇 전체를 부숴야 함
모듈형 로봇	팔만 분리하여 간단히 교체 가능

- 직원의 급여 산출

- Employee 클래스: 급여 계산 메서드를 추상화
- FullTimeEmployee, PartTimeEmployee 등 하위 클래스에서 각기 다른 방식으로 급여 계산 메서드 구현

OCP(Open Closed Principle) 원칙

```
class Employee {  
    private int empSalary;  
    private String empType; // mere, manager 등  
    public Employee(int empSalary, String empType) {  
        this.empSalary = empSalary;  
        this.empType = empType;  
    }  
    public int calculateSalary() {  
        if (empType.equals("Mere"))  
            empSalary += 90;  
        else if (empType.equals("Manager"))  
            empSalary += 120;  
        return empSalary;  
    }  
}
```



새로운 직원 추가시 코드 수정

OCP(Open Closed Principle) 원칙

```
abstract class Employee {  
    private String empName;  
    private int empSalary;  
    public Employee(String empName, int empSalary)  
{  
        this.empName = empName;  
        this.empSalary = empSalary;  
    }  
    public abstract int calculateSalary();  
}
```

Employee를 상속받는 클래스
추가로 새로운 종류의 직원 추가

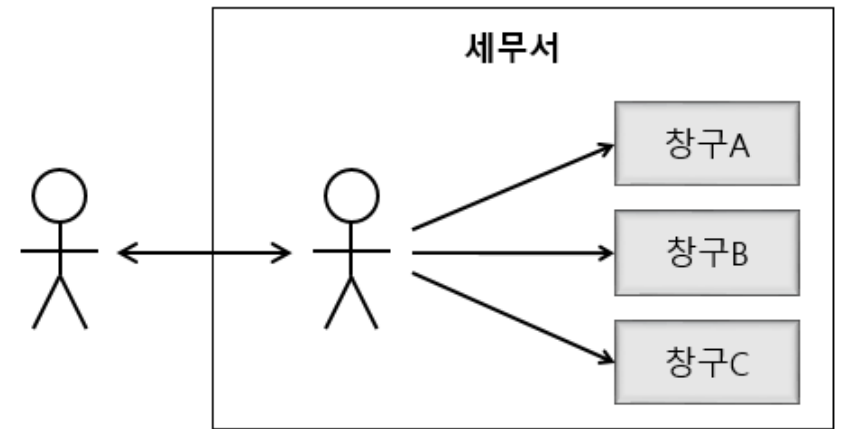
```
class MereEmployee extends Employee {  
    public MereEmployee(String empName, int empSalary) {  
        super(empName, empSalary);  
    }  
    public int calculateSalary() {  
        int salary = getEmpSalary();  
        return salary += 90;  
    }  
}  
  
class ManagerEmployee extends Employee {  
    public ManagerEmployee(String empName, int empSalary) {  
        super(empName, empSalary);  
    }  
    public int calculateSalary() {  
        int salary = getEmpSalary();  
        return salary += 120;  
    }  
}
```

LoD원칙

- 디미터 법칙(Law of Demeter)이란?
"낮은 결합도를 위한 설계 원칙"
- 핵심 개념
클래스는 자신과 밀접한 객체에만 메시지를 보내야 한다
→ 이 원칙을 지키면 결합도가 자연스럽게 낮아짐
→ 클래스 수정 시 다른 클래스에 미치는 영향 최소화
- 메소드가 호출할 수 있는 대상
 - 자신의 메소드 또는 상위 클래스의 메소드
 - 자신의 속성 객체의 메소드
 - 파라미터로 전달된 객체의 메소드
 - 내부에서 생성한 객체의 메소드
 - 다른 클래스 메소드에서 생성되어 리턴된 객체의 메소드
- "너무 멀리 있는 객체에게 직접 명령하지 마라" → 객체 간의 불필요한 의존을 줄이는 핵심 원칙

LoD 원칙

- 고객(클래스)은 내부 처리 과정을 알 필요 없이, 직접 참조된 직원(객체)에게만 요청해야 한다.
- 이는 직원이 아닌 사람과는 이야기하지 마라"는 원칙이며, LoD(Law of Demeter)를 의미한다.
- 클래스는 자신의 멤버 변수나 메소드 인자로 받은 객체만 사용해야 하며, 다른 객체의 속성 객체를 반환받아 사용하는 것은 LoD 위반이다.
- 단, 다른 클래스 내부에서 생성된 로컬 객체를 반환받는 것은 예외적으로 허용된다.



```
package cleancode.section07.lod;

public class Bank {
    public Map<String, Integer> safe;
    public Bank() {
        safe = new HashMap<String, Integer>();
    }
    public void makeAccount(String customerCode, int currentMoney) {
        safe.put(customerCode, currentMoney);
    }
}

public class BankTeller {
    public Bank bank;
    public BankTeller() {
        bank = new Bank();
    }
    public void makeAccount(String customerCode, int currentMoney) {
        bank.makeAccount(customerCode, currentMoney);
    }
}
```



```
public class Customer {  
    public String customerCode;  
    public int currentMoney;  
    public BankTeller teller;  
    public Customer(String customerCode, int currentMoney) {  
        teller = new BankTeller();  
        this.currentMoney = currentMoney;  
        this.customerCode = customerCode;  
    }  
    public int withdrawal(int money) {  
        int deposit = teller.bank.safe.get(customerCode);  
        teller.bank.safe.put(customerCode, deposit - money);  
        return deposit - money;  
    }  
    public int deposit(int money) {  
        int deposit = teller.bank.safe.get(customerCode);  
        teller.bank.safe.put(customerCode, deposit + money);  
        return deposit + money;  
    }  
    public void makeAccount() {  
        teller.makeAccount(customerCode, currentMoney);  
    }  
}
```

```
public class BankingTest1 {  
    public static void main(String[] args) {  
        Customer kim = new Customer("CUST-A0001", 1000);  
        kim.makeAccount();  
        System.out.println("입금 후 잔액 : " + kim.deposit(10));  
        System.out.println("출금 후 잔액 : " + kim.withdrawal(20));  
    }  
}
```

- Customer가 예금과 인출 작업을 하기 위해서는 창구직원(BankTeller)과 은행(Bank)과 금고 정보를 모두 알아야 한다.
- 만약 이들 중 하나라도 변경이 발생되면 Customer를 무조건 수정되어야 한다.
- 이렇게 LoD를 어긴 프로그램은 결합도가 높아지기 때문에 유지보수가 어려울 수밖에 없다.

리팩토링

```
public class Bank {  
    private Map<String, Integer> safe;  
    public Bank() {  
        safe = new HashMap<String, Integer>();  
    }  
    public int widtrawal(String customerCode, int money) {  
        int balance = safe.get(customerCode) - money;  
        safe.put(customerCode, balance);  
        return balance;  
    }  
    public int deposit(String customerCode, int money) {  
        int balance = safe.get(customerCode) + money;  
        safe.put(customerCode, balance);  
        return balance;  
    }  
    public void makeAccount(String customerCode, int currentMoney) {  
        safe.put(customerCode, currentMoney);  
    }  
}
```

리팩토링

```
public class BankTeller {  
    private Bank bank;  
    public BankTeller() {  
        bank = new Bank();  
    }  
    public int withdrawal(String customerCode, int money) {  
        return bank.widtrawal(customerCode, money);  
    }  
    public int deposit(String customerCode, int money) {  
        return bank.deposit(customerCode, money);  
    }  
    public void makeAccount(String customerCode, int currentMoney) {  
        bank.makeAccount(customerCode, currentMoney);  
    }  
}
```

리팩토링

```
public class Customer {  
    private String customerCode;  
    private int currentMoney;  
    private BankTeller teller;  
  
    public Customer(String customerCode, int money) {  
        teller = new BankTeller();  
        this.currentMoney = money;  
        this.customerCode = customerCode;  
    }  
    public int withdrawal(int money) {  
        teller.withdrawal(customerCode, money);  
        return currentMoney -= money;  
    }  
    public int deposit(int money) {  
        teller.deposit(customerCode, money);  
        return currentMoney += money;  
    }  
    public void makeAccount() {  
        teller.makeAccount(customerCode, currentMoney);  
    }  
}
```

LoD(Law of Demeter) 준수 여부

항목	BankingTest1	BankingTest2
접근 방식	customer → teller → bank → safe까지 깊게 접근	customer → teller까지만 접근, teller가 내부 위임
결합도	높은 결합도 (Customer가 Bank와 Map에 직접 접근)	낮은 결합도 (중간 객체인 Teller가 모든 처리 위임)
캡슐화	캡슐화 무너짐 (Bank의 내부 구조를 외부에서 직접 조작)	캡슐화 잘 지켜짐 (Bank는 외부에 노출되지 않음)
디미터 법칙(LoD)	위반됨 (반환 객체의 객체에 접근: teller.bank.safe.get(...))	준수함 (직접 참조된 객체(teller)만 사용)
유연성 및 유지보수	Bank 구조 변경 시 Customer도 같이 수정 필요	구조 변경이 Teller까지만 영향을 줌, 확장성 ↑

CaseStudy02 객체지향 설계 원리 실습