IT@COOKBOOK

쉽게 배우는 소프트웨어 공학

2판

Chapter 05 설계

목차

학습목표

- 설계 원리에 대해 알아본다.
- 모듈의 평가 기준을 자세히 살펴본다.
- 사용자 인터페이스 설계 지침을 알아본다.

01. 설계의 이해

■ 설계

- 건물 건축과 소프트웨어 설계
  - 건축: 건축주와 설계사가 의견 교환을 통해 도면을 확정하면 최종 산출물로 설계 도면이 만들어지고 이를 기반으로 건설
  - 소프트웨어: 요구분석명세서를 작성 후 이를 참조해 개발팀에서 설계서를 만든 후 이를 기반으로 구현 작업에 착수

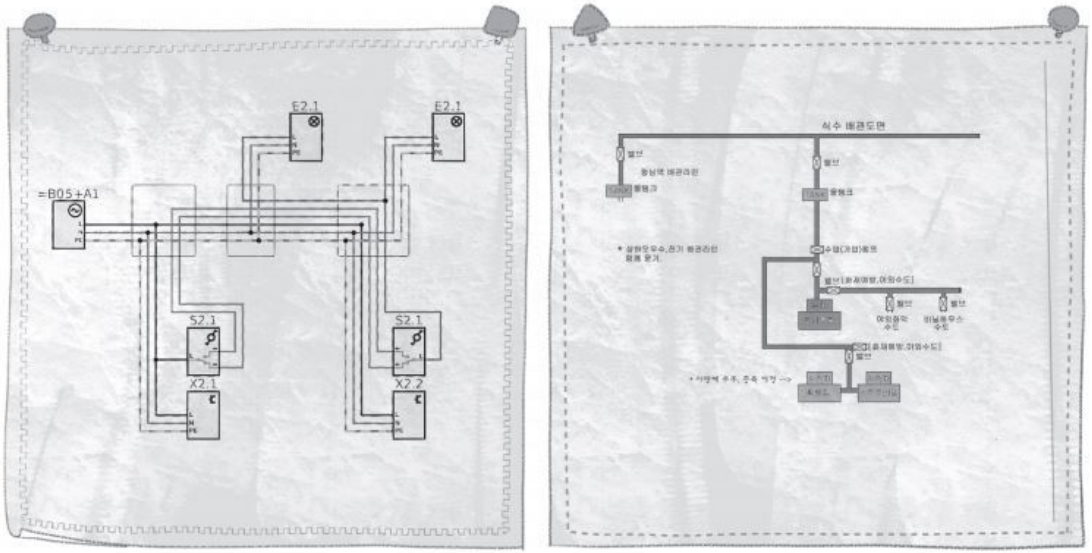


그림 5-1 전기 배선도와 상하수도 배관도

01. 설계의 이해

■ 설계

- 요구분석과 설계의 차이

표 5-1 요구분석과 설계의 차이

구분	요구분석	설계
산출물	요구분석명세서	설계서
관점	무엇(what)	어떻게(how)
특성	개념적, 추상적	사용 환경을 반영해 구체적
비고	미고려 대상: –운영체제 –DBMS(오라클, MS-SQL Server 등) –프레임워크(.NET, J2EE 등)	고려 대상: –비기능 요구사항 –제약 사항 –플랫폼: 운영체제, 미들웨어, 프레임워크

- 좋은 설계가 되기 위한 조건
  - 설계서는 요구분석명세서의 내용을 모두 포함해야 함
  - 유지보수가 용이하도록 추적이 가능해야 함
  - 변화에 쉽게 적응할 수 있어야 함
  - 시스템 변경으로 인한 영향이 최소화되도록 국지적이어야 함
  - 설계서는 읽기 쉽고 이해하기 쉽게 작성해야 함

02. 설계의 원리

■ 객체지향 중심의 설계 원리

1. 분할과 정복
2. 추상화
3. 캡슐화
4. 정보은닉
5. 상속
6. 다형성

02. 설계의 원리

1. 분할과 정복

- 분할과 정복의 개요
  - 분할: 큰 소프트웨어 하나를 개발할 때 여러 개의 서브시스템으로 세분화해 나누는 작업
  - 정복 어느 정도 수준까지 분할했다면 말단에 있는 것부터 하나씩 개발하는 작업

- 분할과 정복의 원리
  - 프로젝트를 수행할 때 먼저 작은 단위로 분할한 뒤 말단에 있는 작은 시스템부터 개발하면서 하나씩 위로 올라가며 완성

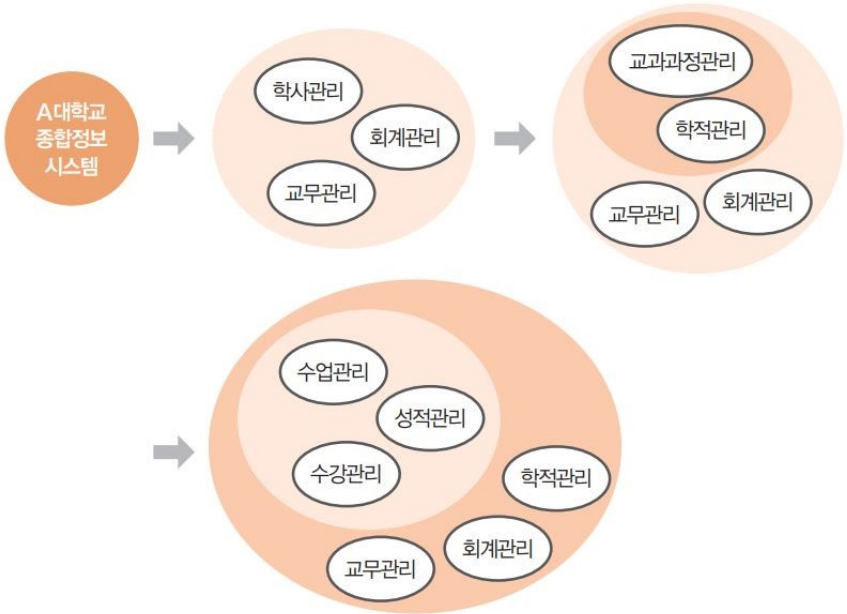


그림 5-2 분할의 예

02. 설계의 원리

- 분할의 기준
  - 분산 시스템은 클라이언트와 서버로 분할
  - 시스템은 여러 서브시스템으로 분할
  - 서브시스템은 하나 이상의 패키지로 분할
  - 패키지는 유스케이스나 여러 클래스로 분할

- 분할의 주의사항
  - 여러 개의 모듈로 분할하면 모듈끼리 통신 횟수가 많아지면서 모듈로 분할하는 장점보다 복잡도가 오히려 증가할 수 있음
  - 설계자는 어느 수준까지 모듈을 분할할지 결정할 때 복잡도 증가로 인한 부작용과 모듈 분할로 얻는 이득을 함께 고려

02. 설계의 원리

2. 추상화

추상화의 개요

자신에게 필요한 특징만 표현한 것

등산가는 등산로 그림이 필요하고 지형을 연구하는 사람은 등고선 그림이 필요

전기 배선 공사 담당자에 게는 전기 배선도, 배관 설치 담당자에게는 상하수도 배관도가 필요

주어진 문제에서 현재의 관심사에 초점을 맞추기 위해 특정한 목적과 관련된 필수 정보만 추출해 강조

관련 없는 세부 사항은 생략해 본질적인 문제에 집중할 수 있도록 하는 작업

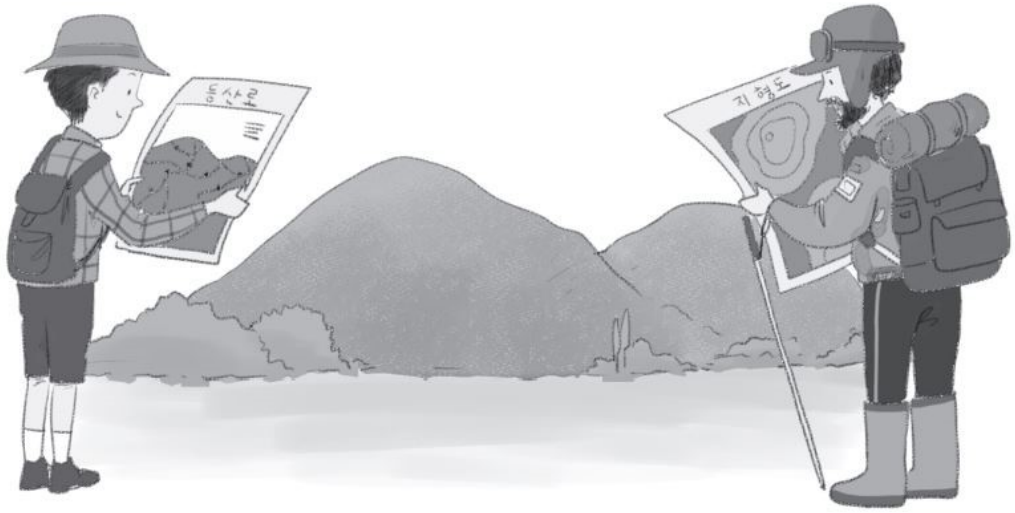


그림 5-3 추상화의 예

02. 설계의 원리

- 추상화의 개요
  - 객체지향 설계에서 추상화의 의미
    - 유사한 특성을 가진 것끼리 그룹화한 뒤 공통점을 뽑아 이름을 붙이는 것

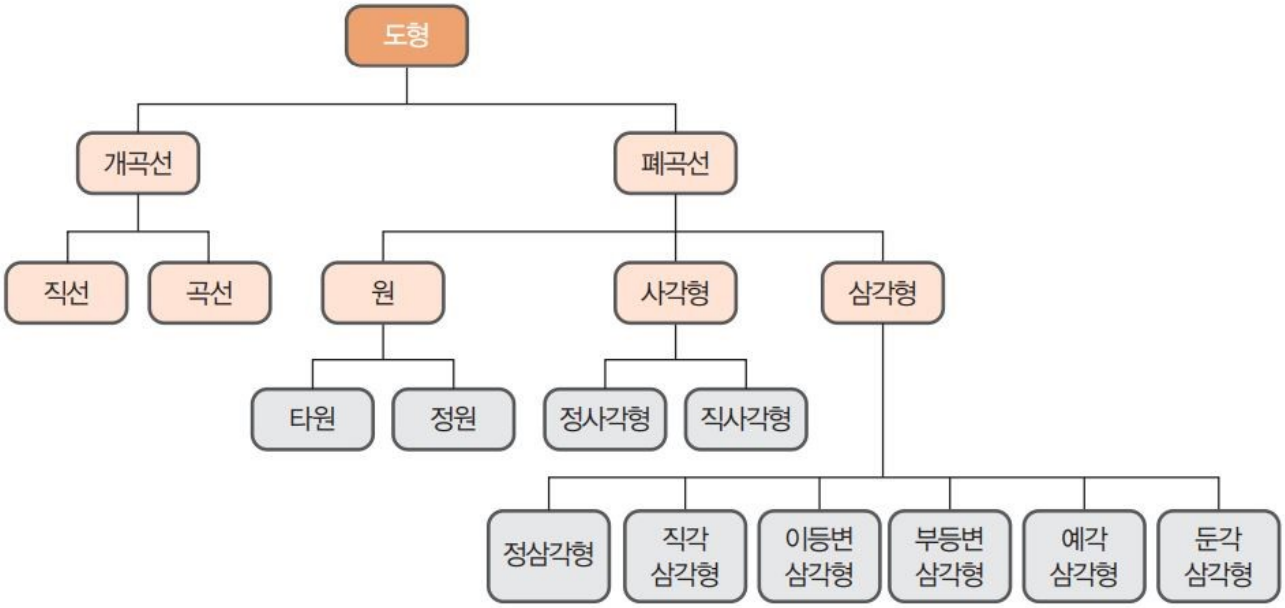


그림 5-4 객체지향 설계에서 추상화의 예

02. 설계의 원리

- 추상화의 종류
  1. 과정 추상화
  2. 데이터 추상화
  3. 제어 추상화

02. 설계의 원리

▪ 과정 추상화

- 주어진 문제에 대해 프로그래밍하기 전에 상세 부분은 생략하고 전체 흐름만 파악할 수 있는 알고리즘 형태로 작성하는 것

• 과정 추상화 예시

- 학점을 구하는 과목의 이름은 구체적으로 언급하지 않고 '합계 및 평균을 구한다'라고만 표현
- 과목의 점수 합계를 정렬할 때도 구체적인 방법(버블 정렬, 퀵 정렬, 선택 정렬 등)은 언급하지 않고 '정렬한다'라고만 표현
- main( ) 함수에서 GetSum( )이라는 함수명은 한 줄로 표현했지만 이 함수의 구현 부분은 5줄짜리 코드

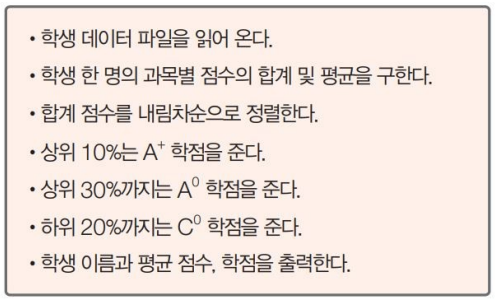


그림 5-5 과정 추상화의 예 1: 학점 계산 알고리즘

```
int main(void) {
   int result;

   result = GetSum(10);
   printf("1~10의 합계 = %d\n", result);
   return 0;
}

int GetSum(int num) {
   int i;
   int sum = 0;
   for(i=1; i<=num; i++)
      sum += i;
   return sum;
}
```

그림 5-6 과정 추상화의 예 2: 함수 호출

02. 설계의 원리

▪ 데이터 추상화

- 데이터와 데이터 구조를 감추는 것으로 대표적인 예가 C++ 언어의 클래스
- 데이터와 메서드를 클래스 형태로 캡슐화해 숨겨 놓고 사용자에게는 꼭 필요한 기능만 사용할 수 있게 개방한 구조

• 데이터 추상화 예시(스택)

- 데이터가 1개씩 증가할 때마다 스택 안에 쌓임(push( ))
- 1개씩 감소할 때마다 스택 안에 있는 요소를 1개씩 제거(pop( ))

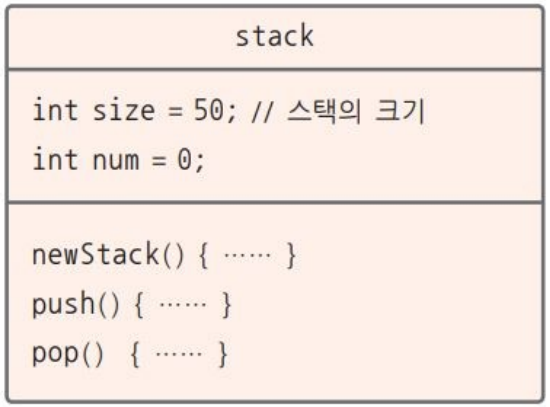


그림 5-7 데이터 추상화의 예: 스택

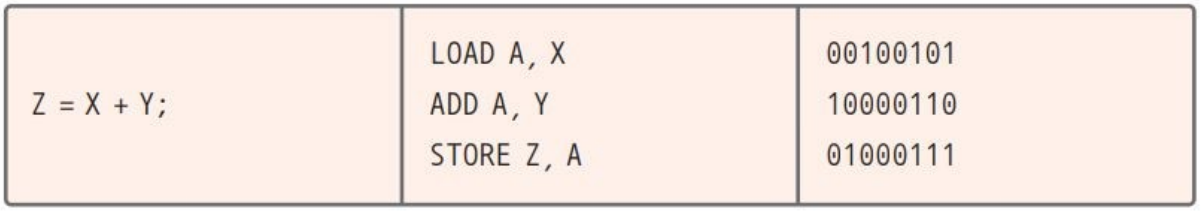
02. 설계의 원리

▪ 제어 추상화

- 프로그래밍 언어에서 쓰는 제어 구조를 추상화
- 제어 추상화는 단계가 올라갈수록 표현이 더욱 간결해지고 특징만 나타낸다는 장점
- 프로그래밍에서 조건을 나타내는 if 문이나 반복을 나타내는 for 문도 사용자가 사용하기 쉽게 추상화한 것

• 제어 추상화 예시

- (a)는 C 언어와 같은 고급 언어로 표현한 것이고 (b)는 이를 어셈블리 언어, (c)는 기계 언어



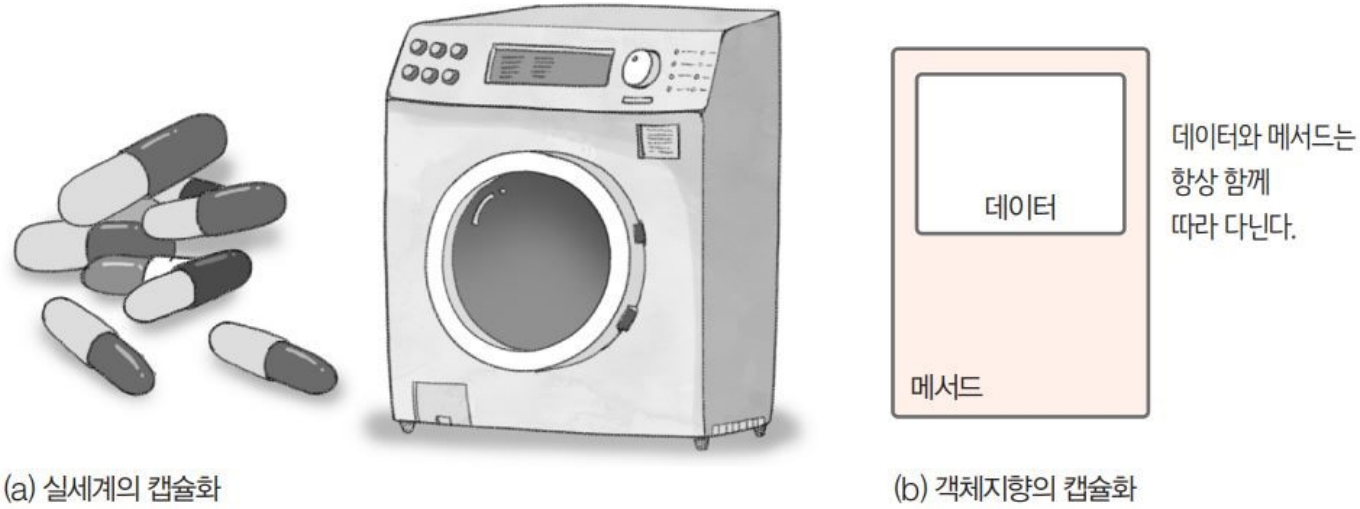
Z = X + Y;	LOAD A, X ADD A, Y STORE Z, A	00100101 10000110 01000111
(a) 고급 언어 표현	(b) 어셈블리 언어 표현	(c) 기계 언어 표현

그림 5-8 제어 추상화의 예

02. 설계의 원리

3. 캡슐화

- 캡슐화의 개요
  - 전자제품은 내부가 어떻게 구성되어 있으며 어떤 방식으로 돌아가는지 몰라도 기능과 사용법을 아는게 중요함
  - 사용자에게 해당 객체의 기능(서비스)과 사용법만 제공해 사용하기 쉽게 하고 내부는 함부로 변경할 수 없게 감추는 개념
  - 블랙 박스와 같은 것으로 클래스를 사용해 서로 관련된 정보와 처리 방식을 같이 묶고 외부에는 감추어두는 것



(a) 실세계의 캡슐화 (b) 객체지향의 캡슐화

그림 5-9 캡슐화

02. 설계의 원리

- 캡슐화의 장점
  - 추상화를 통해 문제를 쉽게 개념화할 수 있음
  - 객체 제공자와 객체 이용자(외부 객체)를 명확히 분리할 수 있음
  - 메서드의 구현 방법이 바뀌어도 사용자에게는 영향을 미치지 않아 사용하기 쉬음
  - 메서드의 기능만 알면 객체를 쉽게 사용할 수 있음
  - 객체 내 자료구조나 알고리즘이 바뀌어도 다른 객체에 미치는 영향이 적음
  - 캡슐화(데이터+메서드)로 객체 사이의 독립성이 구조적으로 보장
  - 그 객체와 인터페이스로 통신하는 사용자에게는 영향을 주지 않으므로 부담 없이 자료구조를 변경할 수 있음
  - 프로그램을 개발할 때 제공하는 기능만 알면 되므로 사용자(프로그래머)가 모듈을 이해하기 쉬움
  - 모듈 내의 데이터와 알고리즘을 변경하기 쉬우므로 기능을 추가하기도 쉬움

4. 정보은닉

▪ 정보은닉의 개요

- 타인이 무슨 생각을 하는지 알아내려면 많은 대화를 통해 생각을 알아내야 함
- 외부(다른 객체)에서 객체의 내부(데이터)를 들여다볼 수 없다는 개념
- 캡슐화는 캡슐의 내부와 외부를 구분하지만 그 자체로 내부 정보가 외부에 숨겨지지는 않음 이때 정보은닉이 필요함

그림 5-10 일상에서 정보은닉의 예

02. 설계의 원리

- 정보은닉의 속성
  - +(공개, public)
    - public으로 설정된 요소는 같은 시스템에 있는 모든 클래스가 접근할 수 있음
    - 클래스는 클래스 이름, 속성, 메서드로 구성되는데 public으로 설정되는 부분은 주로 메서드
  - -(은닉, private)
    - private으로 설정된 요소는 같은 시스템 내의 다른 클래스가 직접 접근 할 수 없음
    - 해당 클래스의 메서드를 통해서만 접근할 수 있음
    - 클래스에서 대부분의 속성은 private으로 설정
  - #(부분 공개, protected)
    - protected로 설정된 요소는 다른 클래스가 접근할 수 없음
    - 해당 클래스의 메서드와 클래스를 상속받은 하위 클래스만 접근

표 5-2 UML, C++, 자바 언어의 정보은닉 표기

	UML	C++	자바
공개	+	public	public
은닉	–	private	private
부분 공개	#	protected	protected

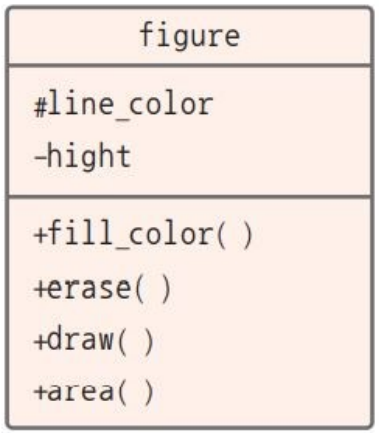


그림 5-11 정보은닉의 표기

02. 설계의 원리

5. 상속

▪ 상속의 개요

- 상위 클래스의 모든 것을 하위 클래스가 물려받아 내 것처럼 사용함을 의미
- 물려주는 클래스: 상위 클래스 또는 부모 클래스 / 물려받는 클래스: 하위 클래스 또는 자식 클래스
- 클래스 간의 상속 관계는 속이 빈 삼각형 모양의 화살표를 사용

• 상속의 장점

- 상속 관계를 이용하면 개별 클래스를 상속 관계로 묶어서 구조를 파악하기 쉬움
- 상속 관계에 속한 클래스, 데이터, 메서드를 추가하기도 쉬움
- 데이터와 메서드를 변경할 때 상위에 있는 것만 수정할 수 있음

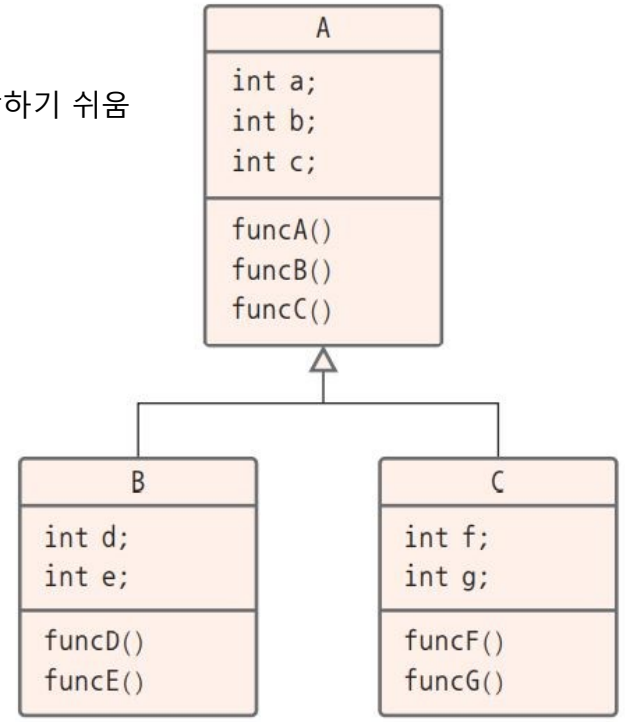
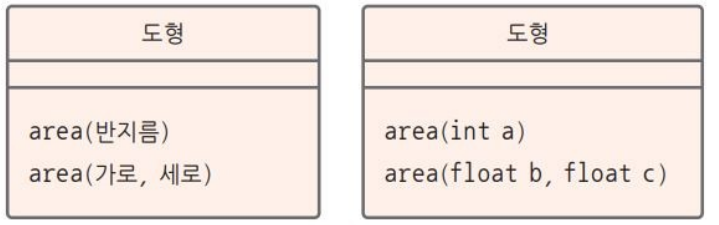


그림 5-12 상속의 개념

02. 설계의 원리

6. 다형성

- 오버로딩(중복 정의)
  - 연산자 중복 정의
    - + 기호: '3+5'처럼 두 수를 더하는 연산자로 사용 / 'go+stop'이 라고 쓰면 문자열을 연결하는 연결자 역할
    - + 기호가 서로 다른 용도로 사용되는 경우를' 연산자 중복 정의' 라고 함
  - 메서드 중복 정의
    - C 언어의 경우에는 한 모듈 안에 동일한 함수명을 사용하는 것이 문법적으로 불가능하지만 객체지향 언어에서는 가능
    - 한 클래스 안에서 이름이 같은 메서드를 중복해서 사용할 수 있음
  - 같은 메서드를 구별하는 법
    - 첫째: 매개변수의 개수로 구별
    - 둘째: 매개변수의 자료형으로 구별
    - 시그니처: 동일한 메서드가 호출되었을 때 구별할 수 있는 매개변수의 개수나 자료형 같은 요소



(a) 매개변수 개수로 메서드 구별 (b) 매개변수 자료형으로 메서드 구별

그림 5-13 메서드 중복 정의의 예

02. 설계의 원리

- 오버라이딩(재정의)
  - 메서드 재정의
    - 상위 클래스에서 정의한 메서드는 무시하고 하위 클래스 에서 다시 정의해 사용하는 것
    - 메서드 재정의를 잘못 사용하면 설계 원칙에 위배
    - 예시) 상위 클래스도 일반 클래스이고 메서드도 일반 메서드를 사용하고 있어 리스코프 교체 원칙을 위반
      - 추상 클래스와 추상 메서드를 사용하면 리스코프 교체 원칙을 위반하지 않음
      - 리스코프 교체 원칙: 상위 클래스의 객체는 언제나 자신의 하위 클래스의 객체로 교체할 수 있어야 함

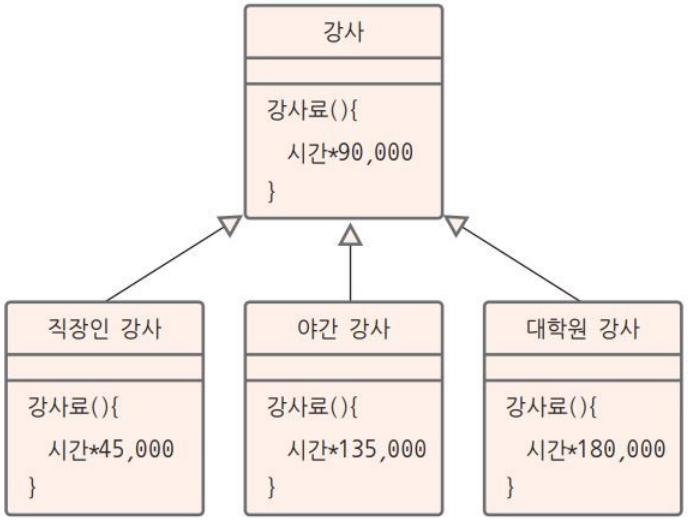


그림 5-14 잘못 사용한 메서드 재정의의 예

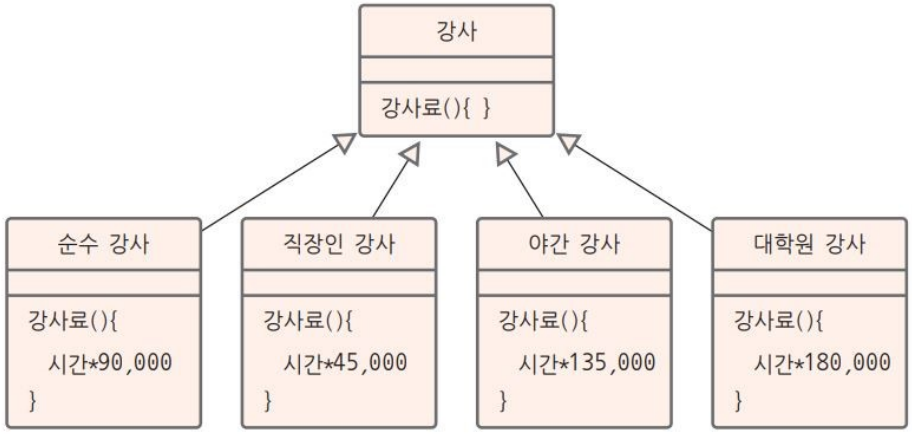


그림 5-15 바르게 사용한 메서드 재정의의 예

03. 모듈화

■ 모듈화

- 모듈의 개요
  - 규모가 큰 것을 여러 개로 나눈 조각, 소프트웨어 구조를 이루는 기본 단위
  - 독립 프로그램도 하나의 모듈이 될 수 있고 함수도 하나의 모듈이 될 수 있음

- 모듈의 특징
  - 다른 것과 구별되는 독립적인 기능을 갖는 단위
  - 유일한 이름을 가짐
  - 모듈에서 또 다른 모듈을 호출할 수 있음
  - 다른 프로그램에서도 모듈을 호출할 수 있음

- 모듈의 원칙
  - 모듈화를 하기 전에 먼저 어느 정도의 크기로 나눌 것인지를 생각함
  - 문제의 유형이나 특성을 고려해 결정
  - 모듈 간의 결합은 느슨하게
  - 모듈 내 구성 요소 간의 응집은 강하게

03. 모듈화

■ 모듈화

- 모듈화의 장점
  - 분할과 정복의 원리가 적용되어 복잡도가 감소
  - 변경하기 쉽고 변경으로 인한 영향도 적음
  - 프로그램을 효율적으로 관리할 수 있음
  - 설계 및 코드를 재사용할 수 있음
  - 문제를 이해하기 쉽게 만듦
  - 유지보수가 용이
  - 오류로 인한 파급 효과를 최소화할 수 있음
- 모듈화의 적정 수준

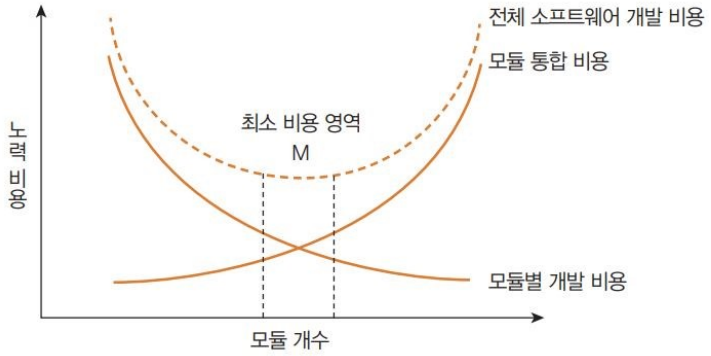


그림 5-16 모듈 개수와 모듈화의 효과

- 모듈 간 관계
  - 호출 관계: 모듈 A가 모듈 B를 호출하는 관계
  - 데이터 전달: 매개변수 등을 이용한 데이터 전달로 이루어지는 관계
  - 제어: 모듈 A가 모듈 B에게 제어 플래그를 전달하는 것과 같은 제어를 통해 이루어지는 관계

03. 모듈화

■ 모듈 평가 기준 1: 응집도

- 응집도의 개요
  - 모듈 내부에 존재하는 구성 요소 사이의 밀접한 정도, 즉 하나의 모듈 안에서 구성 요소 간에 뭉쳐 있는 정도로 평가
  - 응집도가 높을수록 꼭 필요한 구성 요소만 모여 있고, 응집도가 낮을수록 서로 관련성이 적은 구성 요소들이 모임
  - 응집도가 가장 높은 것은 모듈 하나가 단일 기능으로 구성된 경우
  - 응집도가 가장 낮은 것은 구성 요소가 필요에 의해 모듈에 존재하는 것이 아니라 우연히 함께 묶인 경우

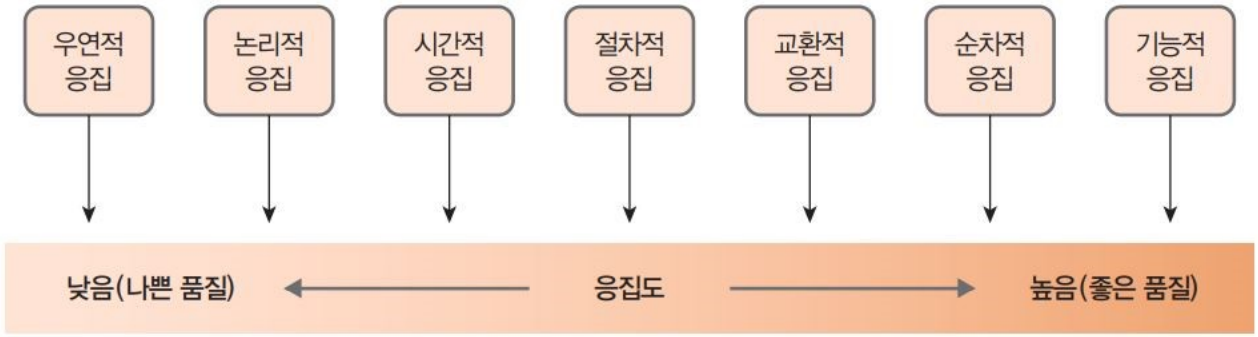


그림 5-17 모듈 내 구성 요소 간의 응집도

03. 모듈화

■ 모듈 평가 기준 1: 응집도

▪ 기능적 응집

- 함수적 응집이라고도 하며 응집도가 가장 높은 경우로 단일 기능의 요소가 하나의 모듈을 구성
- 단일 기능을 갖는 함수가 해당

그림 5-18 기능적 응집의 예

▪ 순차적 응집

- 두 요소가 하나의 모듈로 구성된 경우
- 두 요소가 아주 밀접하므로 하나의 모듈로 묶을 만한 충분한 이유가 됨

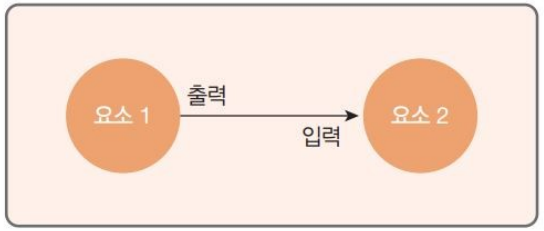


그림 5-19 순차적 응집의 예

03. 모듈화

■ 모듈 평가 기준 1: 응집도

▪ 교환적 응집

- 정보적 응집이라고도 하며 입력을 사용하는 구성 요소가 하나의 모듈로 구성
- 구성 요소가 동일한 출력을 만들어낼 때도 교환적 응집이 됨
- 요소간의 순서는 중요하지 않음
- 순차적 응집 보다 묶인 이유가 조금 약하므로 순차적 응집보다 응집력이 약하다고 할 수 있음

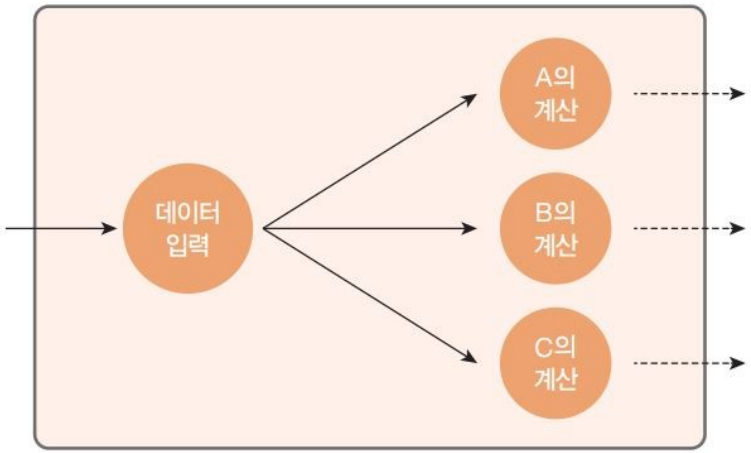


그림 5-20 교환적 응집의 예

03. 모듈화

■ 모듈 평가 기준 1: 응집도

▪ 절차적 응집

- 순서가 정해진 몇 개의 구성 요소가 하나 의 모듈로 구성된 경우
- 순차적 응집과는 어떤 구성 요소의 출력이 다음 구성 요소의 입력으로 사용되지 않고 순서에 따라 수행된다는 점이 다름
- 한 요소의 출력이 다음 요소의 입력으로 사용되지 않으므로 순차적 응집보다는 묶인 이유가 조금 약한 편

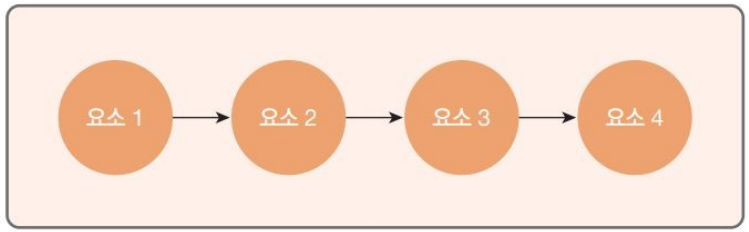


그림 5-21 절차적 응집의 예

03. 모듈화

■ 모듈 평가 기준 1: 응집도

▪ 시간적 응집

- 모듈 내 구성 요소의 기능이 각기 다르고 요소의 출력을 입력으로 사용하는 것도, 요소 간에 순서가 정해진 것도 아님
- 구성 요소들이 같은 시간대에 함께 실행된다는 이유로 하나의 모듈로 구성
- 초깃값 설정 모듈이 시간적 응집의 예

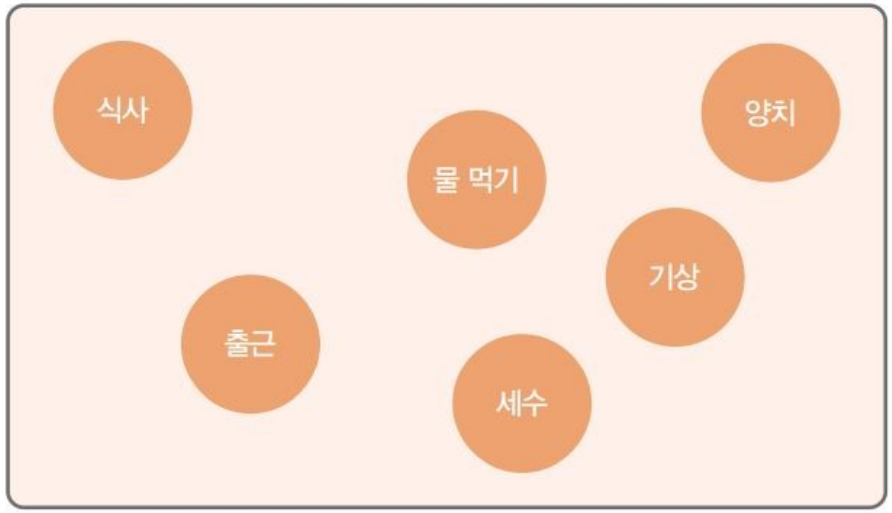


그림 5-22 시간적 응집의 예

03. 모듈화

■ 모듈 평가 기준 1: 응집도

▪ 논리적 응집

- 구성 요소 간에 공통점이 있거나 관련된 임무가 존재하거나 기능이 비슷해서 하나의 모듈로 구성된 경우
  - 비슷한 기능(입출력): scanf( ), printf( )를 결합한 입출력 모듈
  - 공통점(덧셈): 정수의 덧셈과 행렬의 덧셈을 결합한 덧셈 모듈
  - 공통점(출력): 단말기 출력 기능과 파일 출력 기능을 결합한 출력 모듈

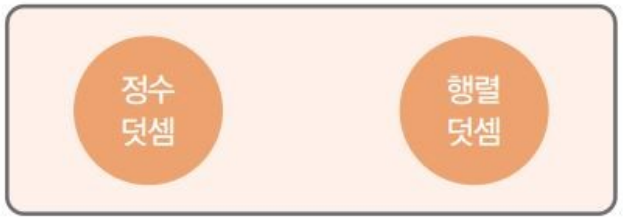


그림 5-23 논리적 응집의 예

03. 모듈화

■ 모듈 평가 기준 1: 응집도

▪ 우연적 응집

- 구성 요소들이 말 그대로 우연히 모여 구성
- 특별한 이유 없이 몇 개의 모듈로 나누는 과정에서 우연히 같이 묶인 것
- 구성 요소 간에 관련이 별로 없어 응집도가 가장 낮음

그림 5-24 우연적 응집의 예

03. 모듈화

■ 모듈 평가 기준 2: 결합도

- 결합도의 개요
  - 모듈과 모듈 사이의 관계에서 관련 정도를 나타냄
  - 모듈 간의 결합에도 간섭하는 관계와 좋은 관계(간섭이 적은)가 있음
  - 모듈 간에는 관련이 적을수록 상호 의존성이 줄어 모듈의 독립성이 높아지고 독립성이 높으면 모듈 간에 영향이 적음
  - 결합에서 좋은 관계는 데이터만 주고받는 관계 / 나쁜 관계는 필요한 데이터만 주지 않고 직접 관여(간섭)하는 관계

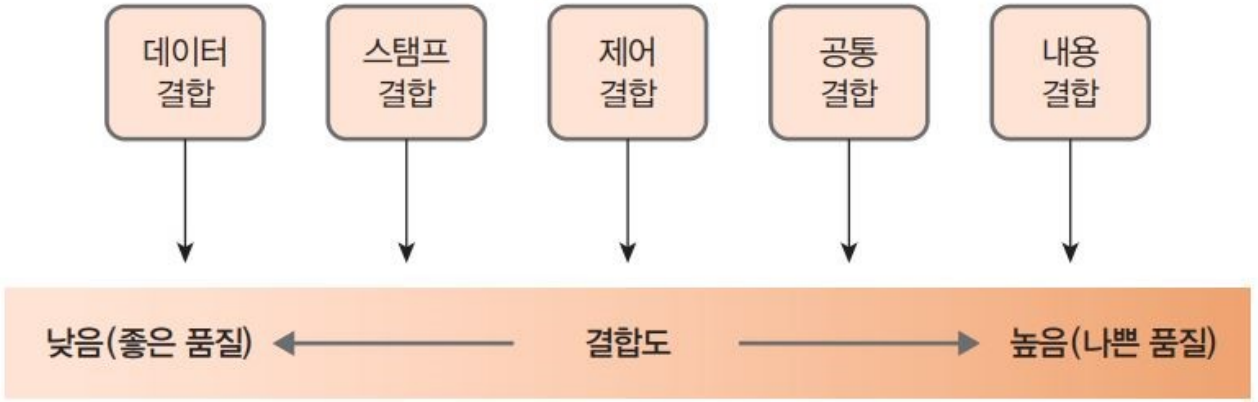


그림 5-26 모듈 간의 결합도

03. 모듈화

■ 모듈 평가 기준 2: 결합도

▪ 데이터 결합

- 가장 좋은 모듈 간 결합
- 모듈이 매개변수를 통해 데이터만 주고받음으로써 서로 간섭을 최소화
- 모듈 간의 독립성이 보장되고, 관계가 단순해 하나의 모듈을 변경해도 다른 모듈에 미치는 영향이 아주 적음
- 유지보수도 매우 쉬움

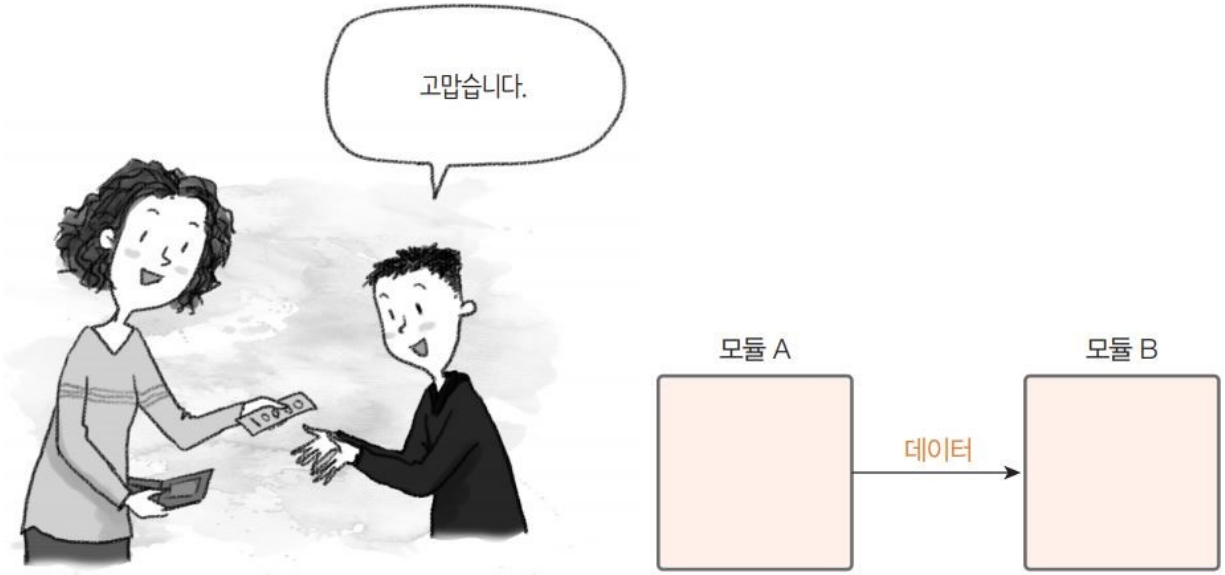


그림 5-27 데이터 결합의 예

03. 모듈화

■ 모듈 평가 기준 2: 결합도

▪ 스탬프 결합

- 두 모듈이 정보를 교환할 때 필요한 데이터만 주고받을 수 없고 필요 없는 데이터까지 전체를 주고받아야 하는 경우
- 스탬프에 해당하는 것은 레코드나 배열 같은 자료구조로 하나의 데이터만 필요한데도 레코드 전체가 넘어옴
- C 언어의 구조체도 스탬프 결합의 예
- 데이터 하나가 변경되면 관련된 모듈에 있는 자료구조를 모두 바꿔야 하므로 데이터 결합보다 모듈간의 관련성이 더 높음

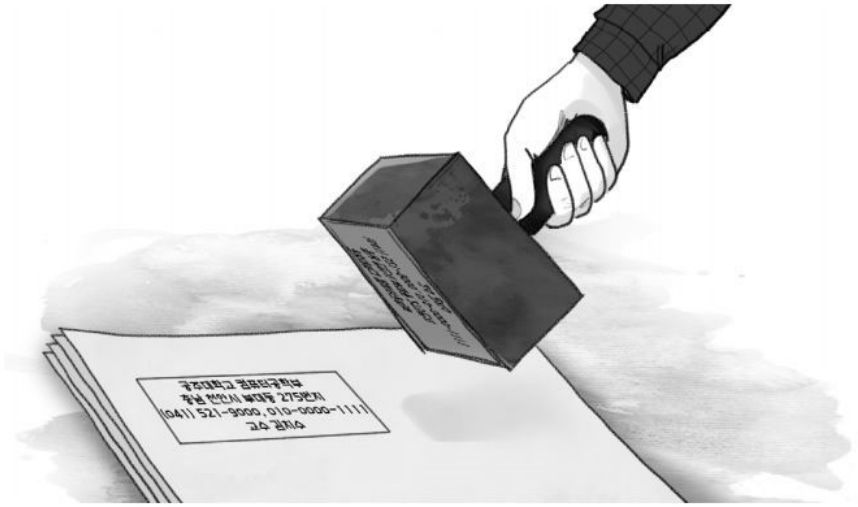


그림 5-28 스탬프

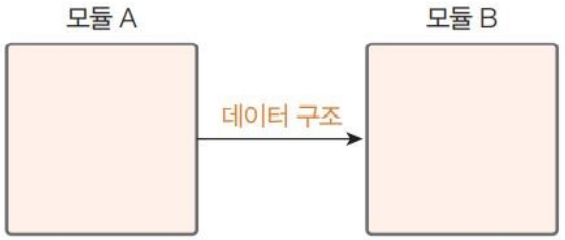


그림 5-29 스탬프 결합의 예

03. 모듈화

■ 모듈 평가 기준 2: 결합도

▪ 제어 결합

- 호출하는 모듈이 호출되는 모듈의 내부 구조를 잘 알고 논리적 흐름을 변경하는 관계로 묶이는 관계
- 데이터 결합이 데이터를 매개변수로 정보를 교환했다면 제어 결합은 제어 플래그를 매개변수로 사용
- 정보은닉을 크게 위배하여 다른 모듈의 내부에 관여해 관계가 복잡해지고 그로 인해 유지보수도 매우 어려워짐
- 스탬프 결합보다 모듈 간의 결합도가 더 높고 모듈의 독립성은 더 낮음

그림 5-30 제어 결합의 일상 예

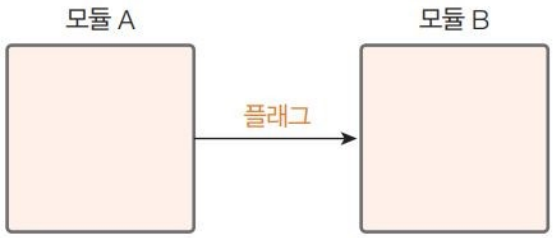


그림 5-31 제어 결합의 예

03. 모듈화

■ 모듈 평가 기준 2: 결합도

▪ 공통 결합

- 모듈이 공통 변수를 함께 사용하므로 모듈이 전역 변수를 사용할 때 공통 결합이 성립
- 변숫값이 변하면 모든 모듈이 함께 영향을 받는 문제가 있음
- 공통 영역에서 문제가 발생하거나 데이터가 변경되면 관련된 모듈을 모두 검토해야 하므로 유지보수가 어려움
- 데이터를 개별 모듈 내부에서 지역 변수로 선언하여 보완

그림 5-32 공통 결합의 일상 예

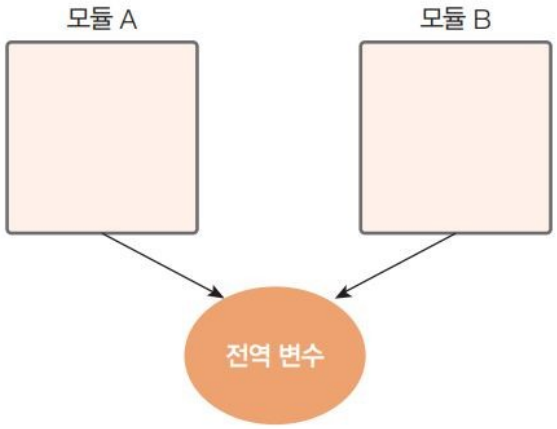


그림 5-33 공통 결합의 예

03. 모듈화

■ 모듈 평가 기준 2: 결합도

▪ 내용 결합

- 모듈 간에 인터페이스를 사용하지 않고 직접 왔다 갔다 하는 것
- 상대 모듈의 데이터를 직접 변경할 수 있어 서로 간섭을 가장 많이 함
- 가장 바람직하지 못한 관계로, 모듈이 서로 종속되어 독립적으로 설계하거나 변경할 수 없음
- 한 모듈이 다른 모듈 내부의 데이터를 직접 참조해 모듈의 독립성이 보장되지 않으므로 유지보수가 매우 어려움

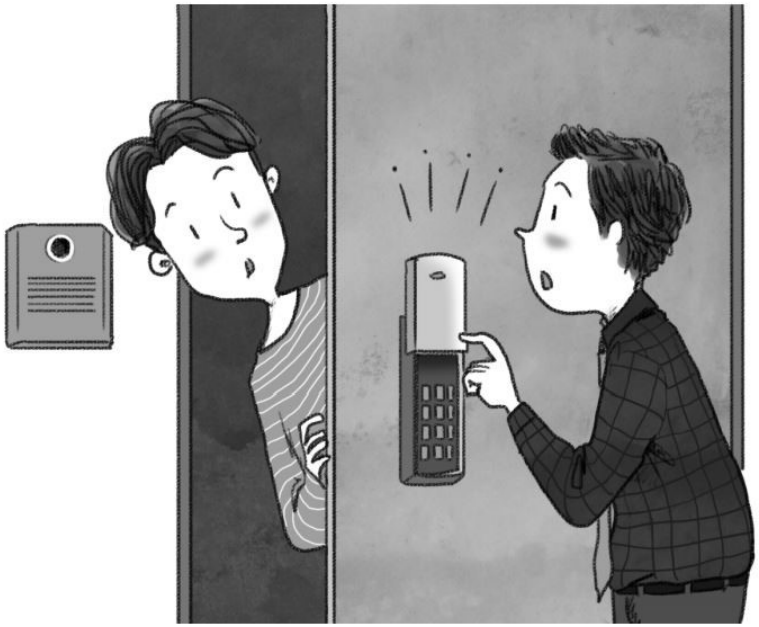


그림 5-34 내용 결합의 개념 예

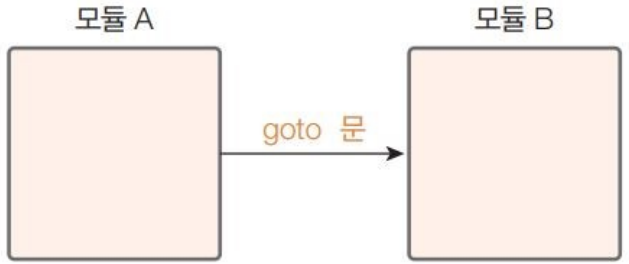


그림 5-35 내용 결합의 예

03. 모듈화

■ 모듈 평가 기준 2: 결합도

▪ 모듈 간의 좋은 관계

- 모듈 간에는 꼭 필요한 데이터만 주고받는 것이 좋음
- 약한 결합을 유지하는 것이 바람직하므로 인터페이스의 수가 적고 복잡하지 않아야 함
- 그러려면 매개변수로 제어 플래그를 사용하는 것보다 데이터를 사용하는 것이 유지보수의 용이성을 높일 수 있어 좋음
- 결론적으로, 설계를 할 때 가장 좋은 형태는 모듈 간의 결합도는 낮게, 응집도는 높게 하는 것

04. 사용자 인터페이스 설계

■ 사용자 인터페이스의 이해

- 사용자 인터페이스
  - 사람과 사람, 사람과 사물, 사물과 사물 사이를 연결해주는 매개체 (TV리모콘, 성적조회 화면 등..)
  - 예전에는 사용자가 문자 사용자 인터페이스 (CUI ,CLI)가 주로 쓰였으나 사용하려면 많은 명령어를 필요로 해 불편
  - 지금은 그래픽 사용자 인터페이스 (GUI), 내추럴 사용자 인터페이스 (NUI)가 쓰임

▪ 사용자 인터페이스 설계 지침

- 사용법을 배우기 쉬워야 함
  - 익숙하지 않은 인터페이스를 제공할 때는 사용자가 쉽게 배울 수 있게 해야 함
  - 아이콘에는 누가 봐도 어떤 기능을 하는지 알 수 있는 이미지를 사용하고 메뉴 이름도 기 능에 적합하게 붙여야 함
  - 현재 사용할 수 있는 메뉴만 활성화하고 사용할 수 없는 메뉴는 흐리게 만듬

- 사용하기 편리해야 함
  - 전화번호를 입력할 때 '-'를 넣어야 할지 말아야 할지 고민하지 않게 미리 메시지를 출력해 알림
  - 카드 번호를 입력할 때 다음 칸으로 옮겨야 하는 경우와 커서가 자동으로 다음 칸에 가 있는 경우

- 사용자가 데이터 입력을 제어할 수 있어야 함
  - 시스템이 원하지 않는 값을 입력 할 경우 시스템은 사용자가 입력한 값이 기대하는 것과 일치하는지 검증할 수 있어야 함
  - 잘못된 입력 값이라면 정확한 값을 입력하도록 메시지를 출력

04. 사용자 인터페이스 설계

■ 사용자 인터페이스의 이해

▪ 사용자 인터페이스 설계 지침

- 사용자의 입력에 반응해야 함
  - 사용자가 입력에 대한 응답을 기다리는 동안 시스템은 궁금하지 않게 처리 진행 상황을 바나 모래시계(⧖)와 같은 것으로 알려주는 반응을 해야 함
- 도움말을 제공해야 함
  - 소프트웨어를 설치하거나 사용할 때 참고할 수 있는 도움말을 제공해야 함
  - 도움말을 제공할 때는 사용자 입장을 충분히 고려해 현재 발생한 문제와 가장 가까운 해결책을 제공하도록 설계
- 일관성을 유지해야 함
  - 일관성 있는 사용자 인터페이스를 제공해야 사용자가 쉽게 기억하고 빠르게 적응할 수 있음
  - 조작 방법 역시 일관성을 유지해야 함
  - 의미가 같은 용어는 통일해서 사용해야 함
- 입력 작업은 최소로 해야 함
  - 사용자가 직접 입력하는 작업은 최소화하고 리스트를 제공해 선택할 수 있도록 설계한
  - 어느 하나가 빠지거나 틀렸다고 처음부터 입력하도록 하면 안 되고 빠뜨리거나 틀린 부분만 다시 입력하도록 해야 함

04. 사용자 인터페이스 설계

■ 사용자 인터페이스의 이해

▪ 사용자 인터페이스 설계 지침

- 효율성을 고려해야 함
  - 키보드 사용이 익숙한 사용자가 사용자 인터페이스의 메뉴를 효율적으로 사용할 수 있게 단축키를 적절히 사용
  - 메뉴에서 공통적으로 많이 사용하는 것은 디폴트 값으로 설정

- 사용자 오류에 대한 되돌리기 기능을 제공해야 함
  - 사용자가 입력을 잘못 했을 경우 시스템은 되돌리기 기능을 제공해 언제라도 이전 상태로 돌아갈 수 있도록 해야 함

- 삭제 또는 취소 버튼 클릭 시 재확인을 요구해야 함
  - 시스템은 사용자가 삭제 버튼을 클릭하면 다시 한 번 물어보는 메시지를 띄워서 사용자의 확인을 받는 과정이 필수

- 사용하기 쉽게 직관적이어야 함
  - 직관적인 사용자 인터페이스를 위한 대표적인 것은 누가 봐도 기능을 알 수 있는 그림을 아이콘에 사용하는 것
  - 심성 모형: 특정 시스템의 기능이나 구조, 가치에 대해 사용자가 잠재적 생각을 가진 모형, 예: 신호등의 색깔, 밑줄
  - 메타포: 은유적인 의미, 대표적인 예는 휴지통 아이콘
  - 직관적인 감각을 반영해 사용성을 최대한 높일 수 있게 설계