

Ch 5. 객체지향 디자인

전문가를 위한 C++ 4판

오기환, 2021. 09. 14

사고방식

절차형 사고방식

- 프로시저: 하나의 작업만 담당하는 작은 단위의 코드
- 이 프로그램이 무슨 일을 어떻게 하는가
- 예시) 주식 시세 정보 확인 및 분석
 - `retrieveQuotes()`, `sortQuotes()`, `analyzeQuotes()`, `outputRecommendation()`
- C에서는 함수가 프로시저 이지만, C가 함수형 언어는 아님
 - C는 절차형 언어, Lisp 이 함수형 언어

객체지향 철학

접근 방식

- 어떤 대상을 모델링 하는가
- 작업 단위가 아닌 대상 모델 단위
- 설계적으로 클래스, 컴포넌트, 프로퍼티 (속성), 동작의 관점으로 분석 가능

객체지향 철학

클래스와 객체

- 클래스: 대상에 대한 정의
 - 오렌지 클래스: 나무에서 자라고 주황색을 띠고 독특한 향과 맛을 내는 과일의 한 종류
- 객체: 하나의 클래스에 속한 특정한 종류의 구체적인 대상 (instance)
 - 오렌지 객체: 나무에 매달린 오렌지, 한입 베어 먹은 오렌지
 - 객체는 클래스의 모든 속성을 포함
- 작은 예로 int, double 과 같은 타입은 클래스 abc, def 같은 변수는 객체라고 비유가능

객체지향 철학

컴포넌트

- 하나의 대상을 구성하는 작은 단위
 - 비행기: 동체, 제어판, 랜딩기어, 엔진 등의 컴포넌트들의 집합
 - OOP에서는 컴포넌트들이 모여 **객체** 를 구성
- 프로그래밍 언어에서는 클래스로 구현됨

객체지향 철학

프로퍼티 (속성)

- 클래스나 객체의 상태를 나타내는 구체적인 값
 - 한 클래스에 속하는 모든 객체가 같은 프로퍼티를 지니는 경우 - 클래스 프로퍼티
 - 각 객체가 갖는 고유의 프로퍼티 - 객체 프로퍼티
- 프로퍼티 값을 통해 서로 다른 객체를 구분

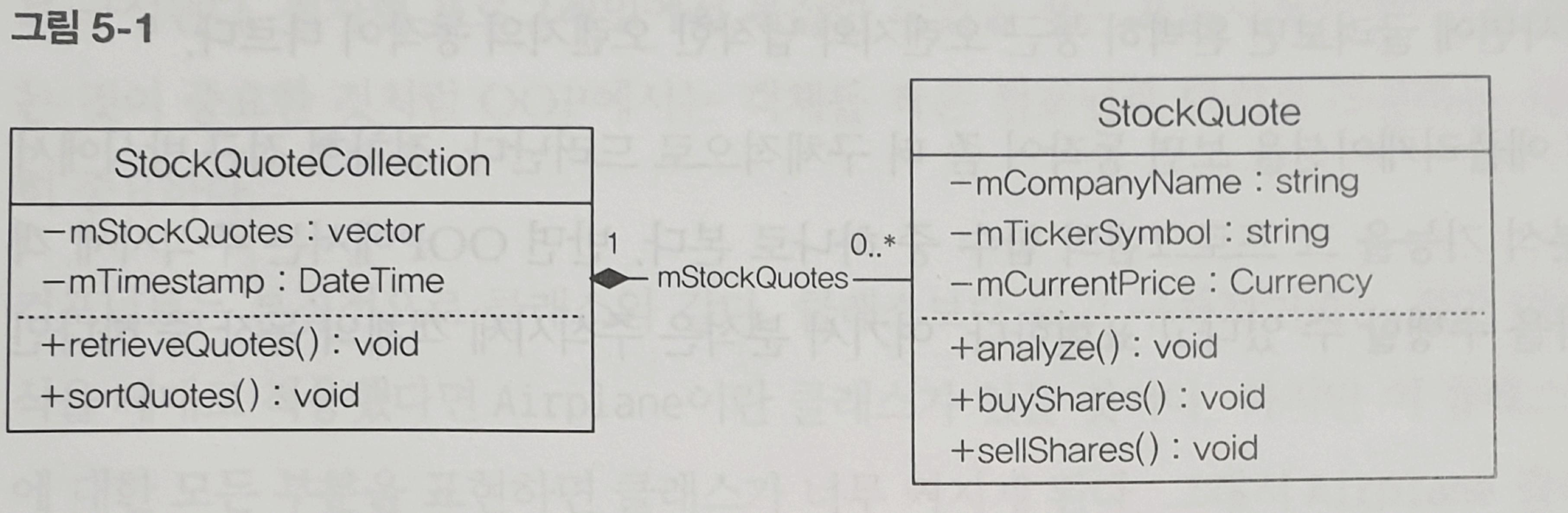
객체지향 철학

동작

- 객체가 하는 일 또는 그 객체로 할 수 있는 일
- OOP에서 클래스의 메소드 단위로 묶어서 표현

객체지향 철학

예제 - 주식 시세 분석



객체 관점

객체 관점

절차형에서 객체지향으로

- 객체를 단순히 데이터와 기능을 잘 묶어주는 수단으로 사용하는 경우
 - 가독성과 유지보수성 증가
 - 코드에서 독립된 부분들을 객체로 전환함
 - 객체를 도구로 다룸
- 처음부터 모든 것을 객체로 표현하는 경우
- 위의 두 경우를 적절히 조합하는 것이 좋다

과도한 객체화

- 과하면 불편해진다
- Tic-Tac-Toe 게임의 객체화
 - 게임 시작/승/패를 관리하는 매니징 클래스 - 딱 저거만 함
 - 게임보드를 격자 객체로 표현 - 그리드 표현만 함
 - 게임에 사용되는 O, X 말 을 객체로 표현 - O, X만 표기함
 - 굳이?
- 고민을 하되 단순히 보여주기식의 객체지향은 지양해야 함

지나친 일반화

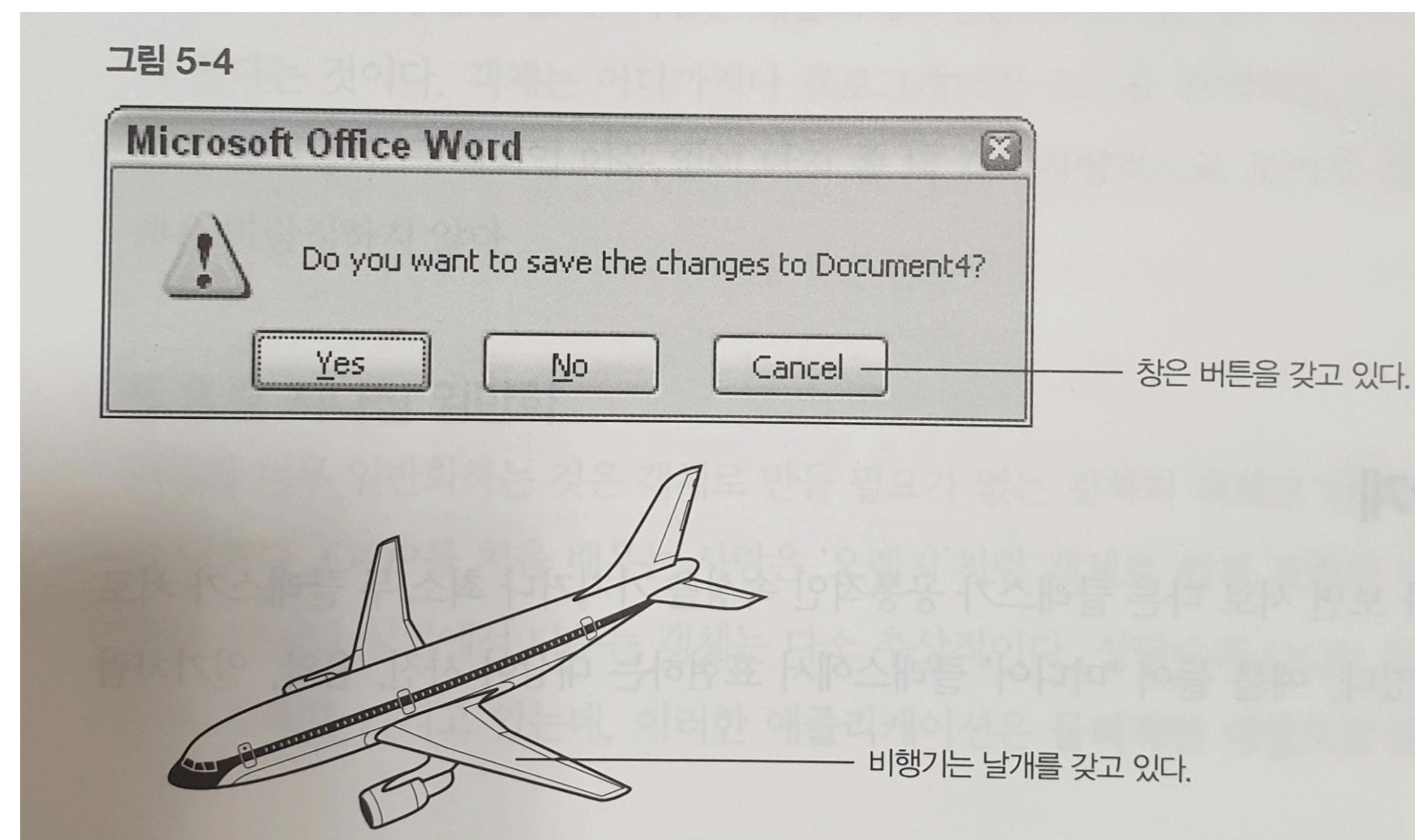
미디어 관리 프로그램 예시

- 모든 파일을 미디어 클래스로 일반화
 - run() 메소드를 정의
 - 그림 - 화면에 띄우기
 - 음악 - 재생
 - 일기 - 텍스트 편집기 실행
 - 서로 완전히 다른 행위를 하는 객체들이 한곳에 묶임
 - 미디어 클래스 이름만으로 어떤 대상을 구체화 하는 것인지 알기 어려움

객체 관계

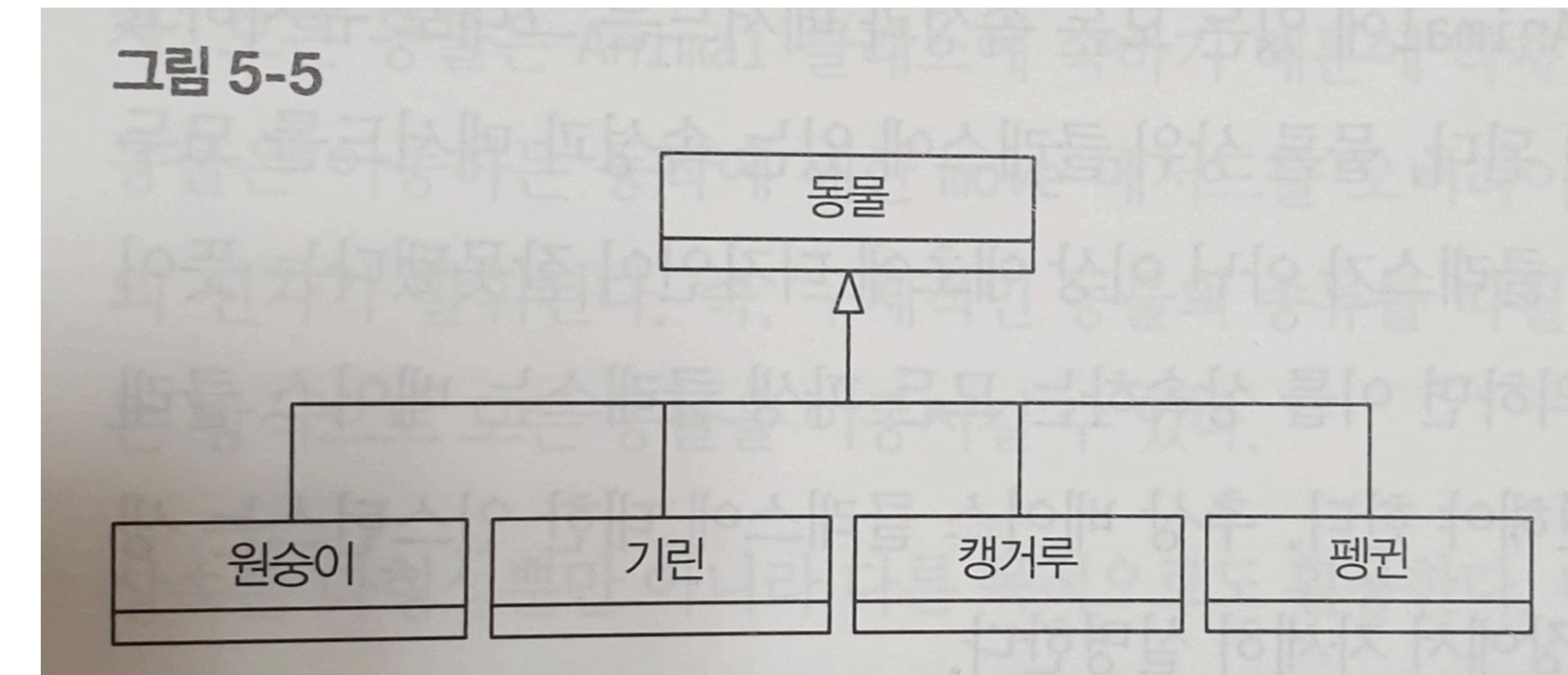
has-a 관계

- A에 B가 있다 혹은 A가 B를 갖는다 - 포함 관계, 소유 관계, 집합 (aggregation) 관계
- 한 객체가 다른 객체의 일부분



is-a 관계 정의

- OOP의 핵심 개념 중 하나인 상속 (inheritance)
 - 파생클래스, 서브클래스, 확장클래스의 표현
 - 계층 구조를 표현
- A는 일종의 B이다



is-a 관계 구현 (1)

상속 기법

- A는 일종의 B이다
 - A: 파생 클래스
 - B: 부모 클래스, 상위 클래스, 베이스 클래스, 슈퍼클래스
- 파생클래스
 - 기능 추가, 기능 변경 (오버라이드), 프로퍼티 추가, 프로퍼티 변경 (오버라이드)

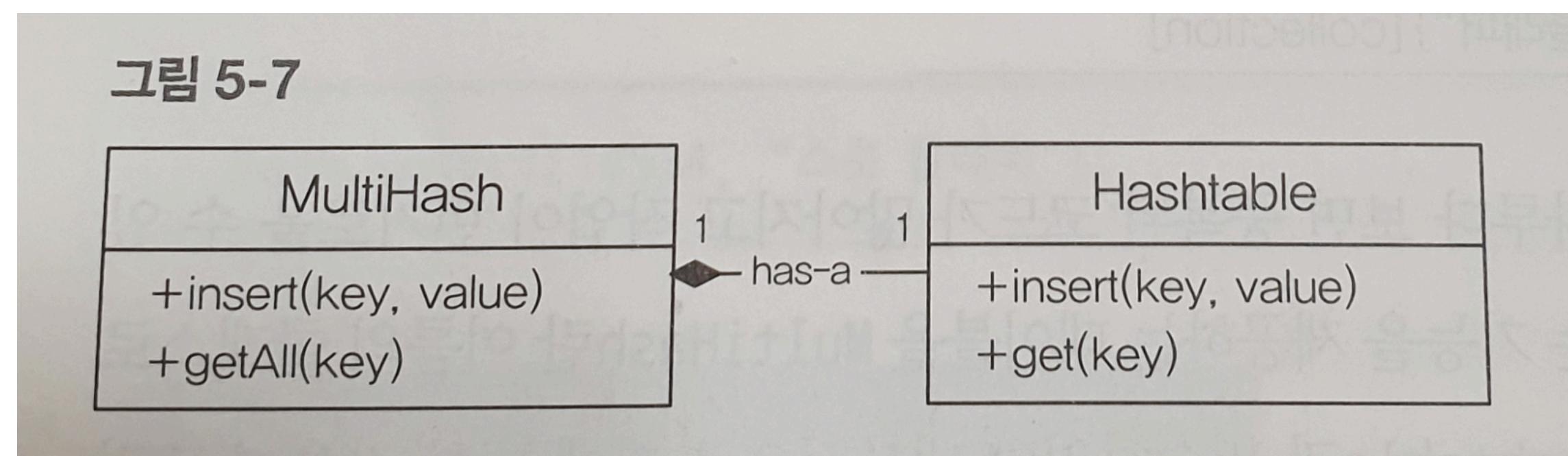
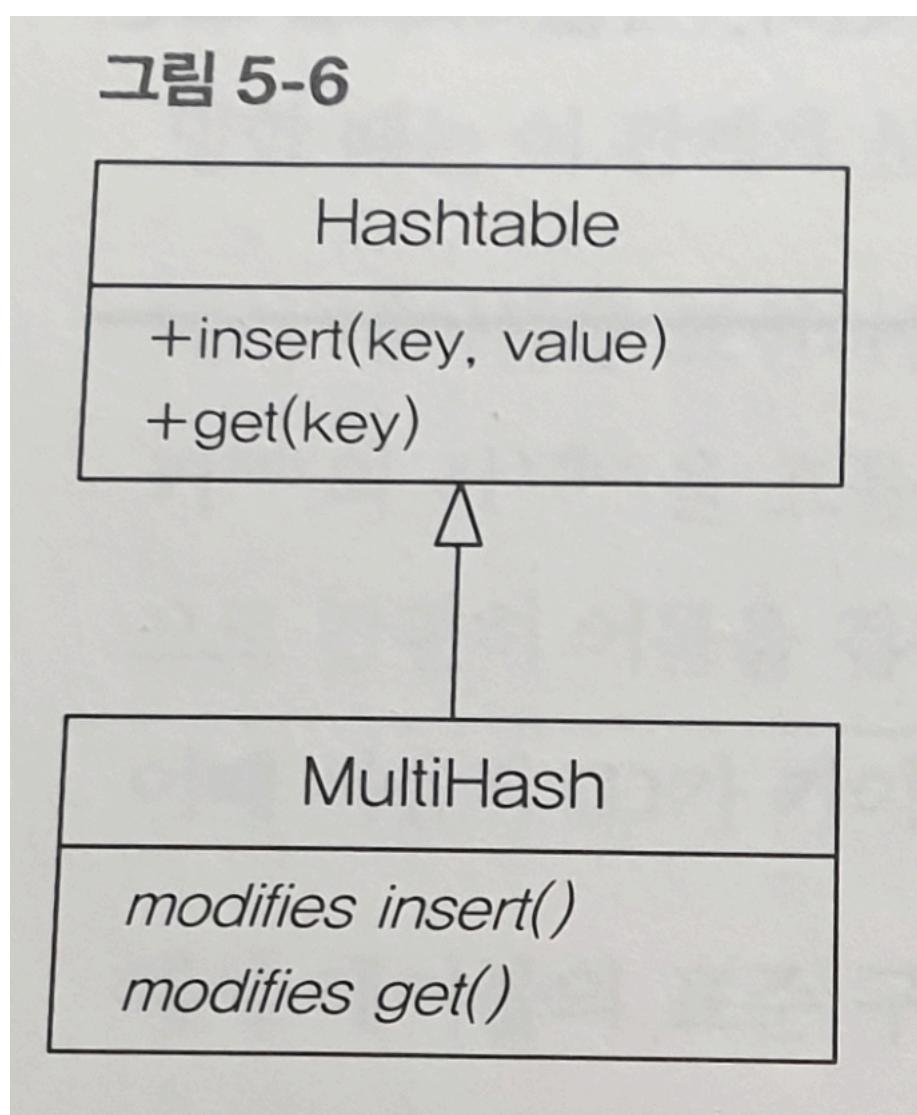
is-a 관계 구현 (1)

다형성과 코드 재사용

- 다형성 (polymorphism) - 프로퍼티와 메소드 형식이 동일한 객체를 서로 바꿔서 사용
- 파생 클래스는 베이스 클래스의 모든 기능을 제공해야함
 - Animal <- 원숭이, 캥거루 등등 관계에서 move() 메소드로 모두 이동 가능
- 기존 코드를 재활용 하기 쉬워짐

has-a 와 is-a 의 구분 (1)

구현 방식

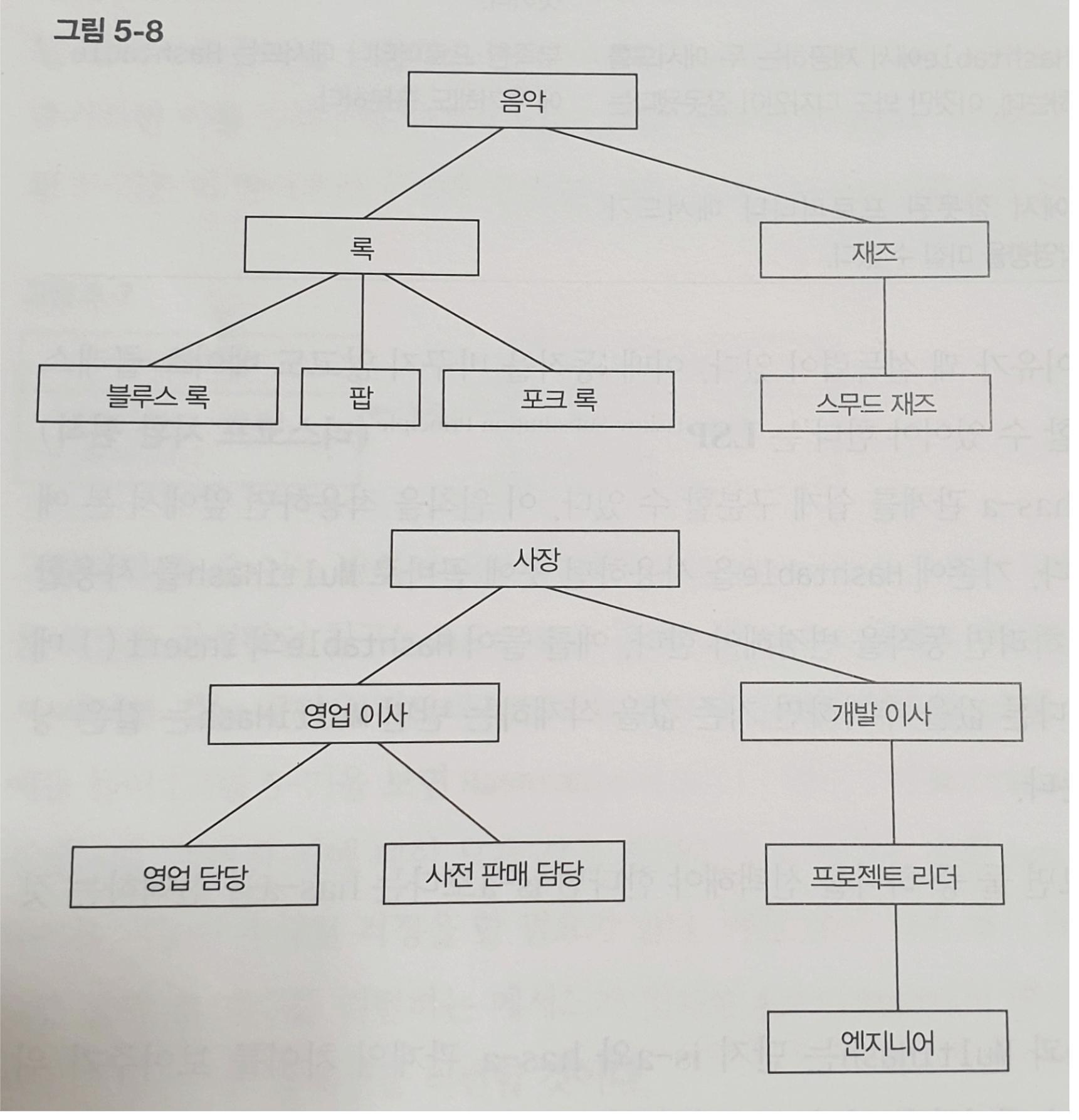


has-a 와 is-a 의 구분 (2)

is-a		has-a
지지 이유	<p>속성만 다를 뿐 추상화 방식은 기본적으로 같다. MultiHash에서 제공하는 메서드는 Hashtable과 별 차이 없다.</p>	<p>MultiHash는 Hashtable에서 제공하는 메서드에 구애받지 않고 마음껏 원하는 메서드를 추가할 수 있다.</p> <p>외부에 드러난 메서드를 변경하지 않고도 Hashtable과 다른 방식으로 얼마든지 변경할 수 있다.</p>
반대 이유	<p>해시 테이블의 정의에 따르면 반드시 키 하나에 값도 하나만 가져야 한다. 따라서 MultiHash는 해시 테이블이라 볼 수 없다.</p> <p>MultiHash는 Hashtable에서 제공하는 두 메서드를 모두 오버라이드하는데, 이것만 봐도 디자인이 잘못됐다는 것을 알 수 있다.</p> <p>Hashtable에서 잘못된 프로퍼티나 메서드가 MultiHash에 악영향을 미칠 수 있다.</p>	<p>MultiHash는 메서드만 새로 고칠 뿐 실질적으로는 기존에 있는 것을 다시 만드는 것이다.</p> <p>부족한 프로퍼티나 메서드는 Hashtable에 추가해도 충분하다.</p>

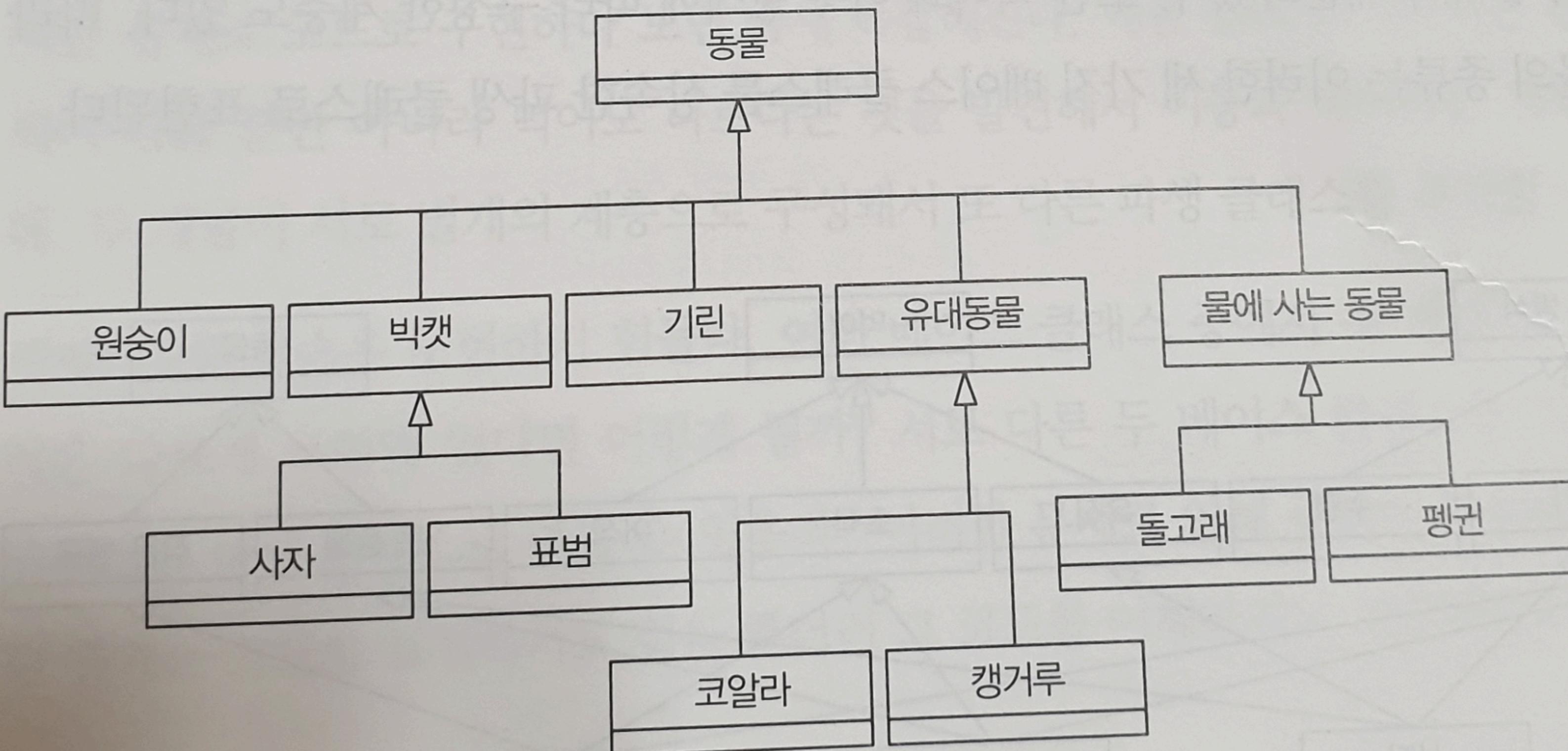
not-a 관계

- 불필요한 관계를 잘못 구현하지 않도록 신중해야함
 - 가능한 유사성 관점으로 생각해야 함
 - 계층 구조 관점에서는 의미가 있으나 실제 코드에서 의미가 없는 경우도 있다
 - 파생클래스에서 베이스 클래스의 프로퍼티와 메서드를 모두 오버라이드 -> 잘못된 상속 관계

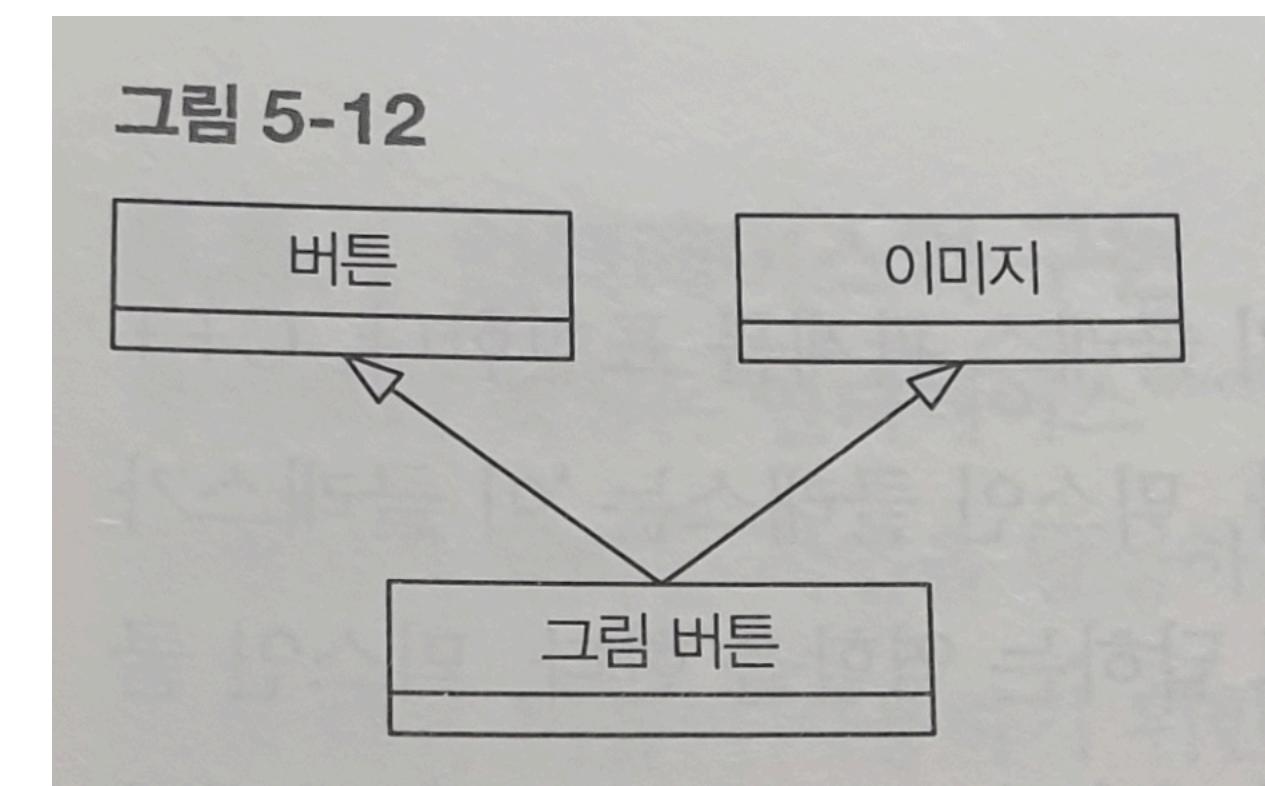
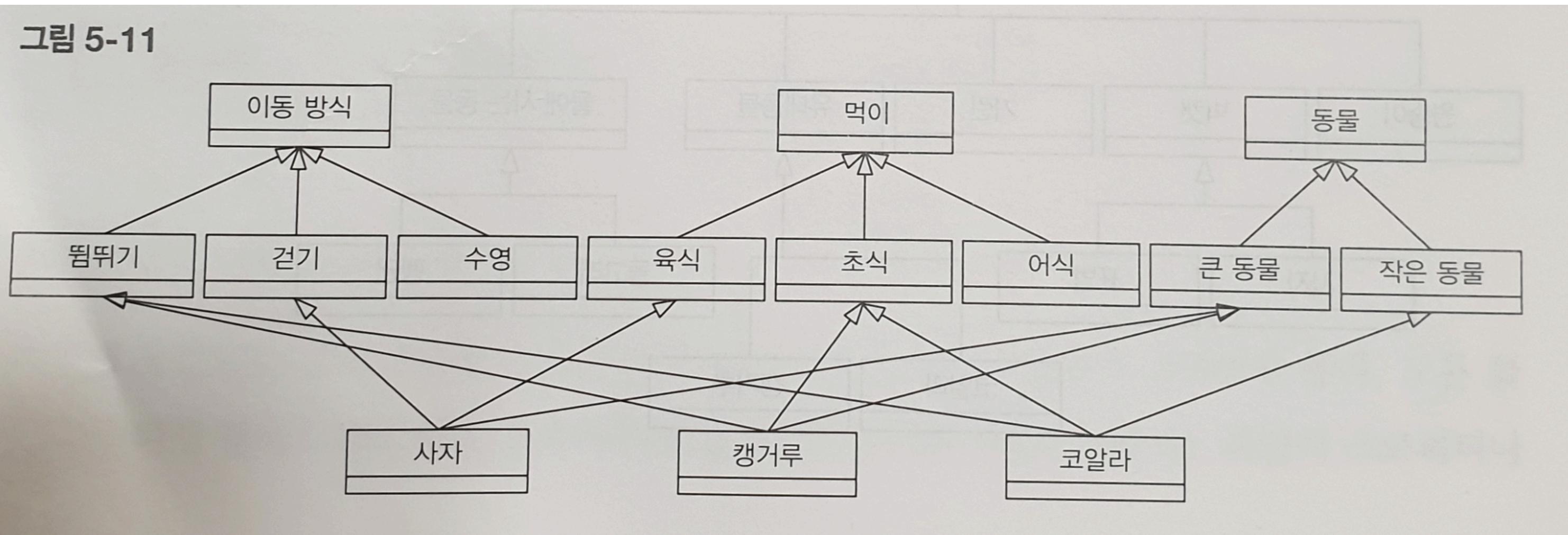


클래스 계층

그림 5-10



다중 상속



다중 상속의 위험성

- 시각적으로 표현하기 복잡함
 - 코드 구조 파악이 복잡해짐
- 클래스 계층 구조의 명확성이 사라짐
- 구현의 복잡성 증가
- 다중 상속은 피할 수 있다면 피하라

믹스인 (mixin) 클래스

- 프로그래밍 문법적으로는 상속과 동일할 수 있음
- '도 할 수 있다' 정도의 표현, 공유 관계
 - is-a 관계를 구현하고 싶지 않을 때
- 예시) Clickable
 - 이미지인데 Clickable 기능만 추가 - ImageButton 대신 Image with Clickable

주상화

인터페이스와 구현

- 인터페이스 (interface)와 구현 (implementation)을 분리
 - C의 헤더파일: 인터페이스
 - OOP: 클래스의 프로퍼티와 메서드
 - 프로퍼티는 최대한 private하게 해야함

외부에 공개할 인터페이스 설정

- C++에서는 public, private, protected 키워드로 공개여부 설정 가능
 - 객체가 아닌 클래스 단위에 적용됨
 - 클래스 내부에서는 같은 클래스인 다른 객체의 private 접근 가능

외부에 공개할 인터페이스 결정 (1)

사용자 고려하기

- 코드를 사용할 사용자를 고려함
 - 개인적인 사용: 아무렇게나 해도 됨
 - 다른 프로그래머가 사용: 다른 프로그래머가 원하는 기능을 모두 구현해야함
 - 시작전에 사용대상인 다른 프로그래머와 미리 인터페이스를 합의
 - 외부 고객: 고객과 협의를 통해 외부에 공개할 기능을 정의
 - 고객에게 익숙한 표현으로 인터페이스를 정의

외부에 공개할 인터페이스 결정 (2)

용도 고려하기

- API 용도: 최대한 변경하지 않을 수 있도록 정의
 - 사용성과 유연성의 절충점
 - 고객의 니즈를 반영
 - 쉬운 일은 쉽게, 복잡한 일은 가능하게
- 유틸리티 또는 라이브러리: 제공하는 모든 기능 공개, 범용성을 최우선
- 서브시스템 인터페이스: 서브시스템의 핵심 목적을 제공
- 컴포넌트 인터페이스: 작은 규모의 인터페이스, 다른 사용자가 있다고 고려하고 설계

외부에 공개할 인터페이스 결정 (2)

미래 고려하기

- 미래에 사용할 일을 고려
- 확장성을 제공하기 위한 플러그인 아키텍처
- 다양한 활용 사례 파악
- 불분명한 기능은 제거

바람직한 추상화 디자인

- 다양한 경험이 필요
- 이미 정립된 디자인 패턴들을 습득하여 빠르게 활용
- 끊임없는 개선
- 동료와의 리뷰
- public 메소드로만 구성된 추상화
- 프로퍼티는 getter, setter로 접근