Chapter 02

객체지향 개념

학습목표

- 절차지향 패러다임과 객체지향 패러다임의 근본적인 차이를 이해할 수 있다.
- 클래스와 객체를 구분하여 설명할 수 있다.
- 인스턴스 변수와 클래스 변수, 메소드와 오퍼레이션의 의미를 이해할 수 있다.
- 클래스의 상속 계층이 무엇이며, 다형성이 어떻게 적용되는지 설명할 수 있다.
- 객체지향 모델링이 어떻게 이루어지며, 표준 언어인 UML이 무엇인지 설명할 수 있다.
- 객체지향 설계와 코딩이 어떤 관계인지 이해할 수 있다.

00 개요

최근 완공된 건물은 10년 전에 지은 건물과 비교하여 자재, 설계 기술, 공법, 마감재 등 여러 면에서 발전된 기술이 적용되었다. 소프트웨어 개발 분야도 프로그래밍, 설계, 테스트 기술이 빠르게 발전하고 있는데, 최근 많이 사용하는 기술 중 하나가 객체지향이다. 이 장에서는 이 책의 주제인 객체지향 방법으로 소프트웨어를 설계하고 구현하는 데 기초가 되는 개념을 설명한다.

먼저 '객체'라는 용어의 사전적 의미는 '의도나 행위가 미치는 대상'으로 다소 포괄적이고 법률적이다. 여기에 '지향'이라는 말을 붙이면 '행위가 미치는 대상으로 쏠리려는 의지' 정도로 풀이할 수 있는데, 이는 객체지향object-oriented이라는 의미를 잘 나타내지 못하는 듯하다.

객체지향의 개념을 잘 설명하려면 객체지향적 소프트웨어를 구성하는 성질을 나열하고 개념을 설명해야 한다. 이 장에서는 객체지향의 중요한 개념인 클래스, 객체, 오퍼레이션, 상속, 캡슐화, 다형성 등을 살펴볼 것이다. 이러한 개념은 객체지향 프로그램을 작성하는 데 필수적이지만 모델링과 연계할 필요도 있다. 객체지향 개념이 자바 같은 특정 프로그램뿐만 아니라 설계에서 어떤 모델링 언어로 표현되는지 알아야 하기 때문이다. 이 책에서는 객체지향 모델링 언어인 UML과 UML 다이어그램에 대해 개괄적으로 소개한다.

01 객체지향이란?

1 프로시저 추상화와 데이터 추상화

객체지향 프로그램은 '추상화abstraction'라는 개념을 이용한다. 추상화란 자세한 사항을 다루는 것에서 해방시켜주는 개념으로, 회화에 비유하자면 대상의 자세한 묘사를 생략하고 특징만 포착해서 그리는 것과 같다고 할 수 있다. 객체지향 프로그램은 프로시저 추상화와 데이터 추상화를 묶어놓은 것이다.

■ 프로시저 추상화

프로시저(=함수) 추상화란 특정 프로시저를 사용할 때 프로시저 안에서 어떤 계산이 이루어지는지 자세히 모르더라도 어떻게 호출하고 무엇을 수행하는지만 알고 있으면 자세한 것은 걱정할 필요가 없는 것을 의미한다. 결국 프로그램에 대한 프로그래머의 관점이 단순해지고 복잡한 것을 쉽게 이해할 수 있게 된다.

■ 데이터 추상화

데이터 추상화란 데이터 자체만 정의하는 것이 아니라 데이터에 대한 조작, 즉 오퍼레이션도 함께 정의하는 개념이다. 데이터 추상화를 적용한 시스템은 서비스를 제공하는 객체들의 집합으로 볼 수 있다. 데이터 추상화는 시스템의 복잡성을 줄이는 데 도움이 된다.

객체지향 패러다임을 잘 이해하려면 초기 프로그래밍 기법인 절차지향 패러다임과 비교해 보아야 한다. 단순한 프로그래밍 문법의 차이가 아니라 프로그래밍적인 사고의 틀을 바꾸어야 올바른 객체지향 설계와 프로그래밍을 할 수 있기 때문이다. 프로시저 추상화와 데이터 추상화의 관점에서 절차지향 패러다임을 분석하면 다음과 같다.

■ 프로시저 추상화 관점

절차지향 패러다임에서 프로그램은 프로시저의 집합으로 구성된다. 메인 프로시저가 다른 프로시저를 호출하고 이것들이 계속 다른 프로시저를 호출하는 것이다. 절차지향 프로그램은 단순한 데이터를 계산하는 목적에는 적합하지만 복잡한 형태의 데이터를 가진 응용

문제에는 적합하지 않다. 왜냐하면 각기 다른 데이터 타입에 대해 다른 프로시저가 작동하고, 각 프로시저마다 많은 데이터 타입을 고려해야 하므로 프로그램이 매우 복잡해지기 때문이다.

■ **데이터 추상화 관점**

초기에 소개된 데이터 추상화 방법은 레코드나 구조체 개념을 적용한 것으로, 의미 있는 데이터들을 모아 큰 단위를 만들어 정의함으로써 편하게 조작하는 것이었다. 하지만 이러한 데이터 추상화를 사용하더라도 절차지향 패러다임을 사용하는 프로그래머는 복잡한 코드를 작성해야 한다.

예를 들어 은행에서 사용하는 '예금 계좌 관리' 프로그램을 절차지향 방법으로 프로그래밍한다고 생각해보자. 예금 계좌의 성격에 따라 이자 계산, 만기 계산, 인출 등의 방식이 다를 것이고, 따라서 [그림 2-1]과 같이 데이터 타입을 체크하는 부분이 프로그램 여기저기에 있을 것이다. 은행의 고객은 다른 계좌를 여러 개 가지고 있고, 계좌들은 서로 연결되어 자동이체되는 식으로 복잡하게 얽혀 있다. 계좌의 성격에 따라 이자 계산, 예금, 인출 방법, 만기 계산 등 규칙이 달라 결국 [그림 2-1]과 같은 코드 조각이 '예금 계좌 관리' 시스템 여러 곳에 산재되어 이자율 등을 변경하기가 매우 어려워진다.

```
if (account == '정기적금') then
    do something
else if (account == '정기예금') then
    do something else
else if (account == '보통예금') then
    do something else
else
    do yet another thing
endif
```

그림 2-1 **'예금 계좌 관리'를 위한 절차지향 프로그램의 알고리즘**

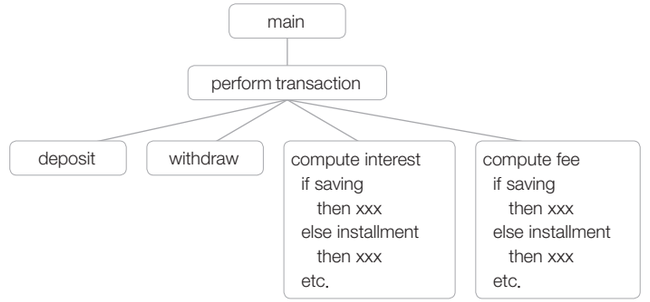
2 객체지향 패러다임

객체지향 패러다임이란 데이터 및 데이터에 접근하고 수정하는 프로시저를 같은 범주 안에 두고 캡슐화하는 접근 방법이다. 객체는 프로그램을 이루는 구문 단위인 클래스의 인스턴스로서 데이터 추상화와 프로시저 추상화를 모두 실현한 개념이다.

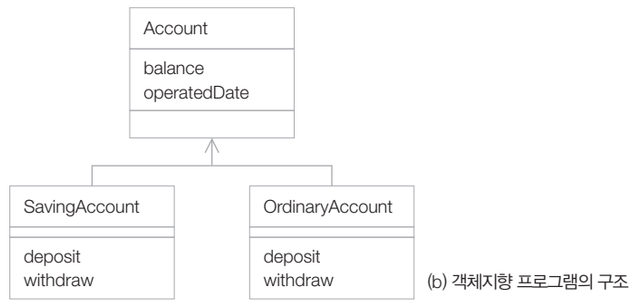
객체지향 패러다임은 1980년 후반부터 복잡한 대규모 시스템을 모듈화하고 단순화할 수 있는 좋은 방법으로 받아들여지게 되었다. 객체지향 패러다임에서는 주어진 작업을 수행하기 위해 협력하는 객체들의 모임이 바로 프로그램이 된다.

[그림 2-2]는 절차지향 프로그램과 객체지향 프로그램의 구조를 보여준다. 절차지향 프로그램은 프로시저가 계층적인 구조를 이루며, 각 프로시저는 여러 가지 타입의 데이터를 처리한다. 프로시저는 함수와 같은 개념의 구문으로, 프로시저 안에서만 사용하는 데이터(로컬 데이터)를 정의하여 사용하거나, 그렇지 않은 공용 데이터는 프로시저 밖에서 별도로 정의하여 사용한다. 반면에 객체지향 프로그램은 클래스의 모임으로 이루어지며, 각 클래스에는 클래스의 인스턴스인 객체의 값을 조작할 수 있는 프로시저들이 모여 있다.

결론적으로 절차지향 프로그램은 최고층이 존재하며 각 프로시저가 하위 계층의 프로시저를 호출하고 실행을 제어하는 계층적인 구조로 되어 있다. 그러나 객체지향 프로그램은 클래스들이 프로그램 구문의 기초 단위를 이루며 호출 관계가 아닌 다른 형태의 관계들로 계층을 형성한다.



(a) 절차지향 프로그램의 구조



(b) 객체지향 프로그램의 구조

그림 2-2 **절차지향 프로그램과 객체지향 프로그램의 구조**

예를 들어 화씨온도를 섭씨온도로 바꾸는 프로그램을 생각해보자. 절차지향 프로그램은 [그림 2-3]의 (a)와 같이 프로시저를 호출하여 여러 가지 계산을 수행한다. 즉 프로시저가 서로 데이터를 주고받으며 작업하고, 프로시저와 데이터는 철저히 분리된다.

객체지향 프로그램은 [그림 2-3]의 (b)와 같이 객체들 사이에 메시지를 주고받으며 작업한다. 절차지향 프로그램에서는 프로시저를 호출하여 파라미터로 데이터를 전달했는데, 그럴 필요 없이 데이터를 가진 객체를 불러 여기에 속한 오퍼레이션을 호출한다.

```
float c=getTemperature();          // assume Celsius
float f=toFahrenheitFromCelsius(c);
float k=toKelvinFromCelsius(c);
float x=toKelvinFromFahrenheit(f);
float y=toFahrenheitFromKelvin(k);
```

(a) 절차지향 프로그램의 예

```
Temp temp=getTemperature();   // temp is an object
float c=temp.toCelsius();             // toCelsius is a message to temp
float f=temp.toFahrenheit();
float k=temp.toKelvin();
```

(b) 절차지향 프로그램의 예

그림 2-3 **화씨 온도를 섭씨 온도로 바꾸는 프로그램**

NOTE_ • Fahrenheit: 화씨온도 • Celsius: 섭씨온도 • Kelvin: 절대온도

02 클래스와 객체

클래스와 객체는 객체지향 패러다임에서 가장 기초가 되는 개념이다. 절차지향 프로그램에서는 언제나 알고리즘, 즉 컴퓨터가 계산하거나 제어하는 작업과 그 순서를 생각했다. 그러나 객체지향에서는 프로그램이 다루는 대상thing을 중심으로 문제를 풀어나간다. 예를 들어 학교 수강 신청 시스템을 객체지향으로 설계한다면 수강 신청 절차보다는 교수, 학생, 과목, 강좌, 강의실 등의 대상을 중심으로 사고하여 설계한다.

1 객체

객체는 실행되는 프로그램에 존재하는 구조화된 데이터의 덩어리로, 절차지향 프로그램에서 구조체로 선언된 변수 또는 배열에 저장된 값과 같이 생각할 수 있다. 즉 데이터를 객체라 불리는, 식별 가능한 분리된 개체로 만든 것이다.

모든 객체는 상태state와 행위behavior라는 두 가지 요소를 가지고 있다. 예를 들어 하천의 댐을 제어하는 시스템이 있다면 댐은 객체가 되고, 수문이 닫힌 상태와 열린 상태는 객체의 상태로 볼 수 있다. 그리고 수문을 여는 행위, 즉 닫힌 상태에서 열린 상태로 변화시키는 행위는 객체의 행위이며, 수문을 여는 행위가 적용되면 자동으로 사이렌이 울리도록 하여 강가의 행락객들을 대피시킬 수 있다.

객체의 또 다른 예는 다음과 같다.

- 급여 시스템에서 사원 개개인을 나타내는 객체
- 대학의 수강 신청 시스템에서 학생, 강좌, 교수를 나타내는 객체
- 공장 자동화 시스템에서 조립 라인, 로봇, 조립 부품, 완제품을 나타내는 객체

객체지향 프로그램을 개발할 때 분석 단계에서는 어떤 객체가 중요한지 결정하고 이 객체들의 구조, 관계, 행위 등을 파악하는 작업을 한다. 이때 객체를 프로그래밍 언어로 어떻게 표현할지, 디스크에 어떻게 저장할지 등은 걱정할 필요가 없다. 분석이 완료된 후 객체지향 설계

단계로 접어들 때까지 이런 문제는 접어두고 객체의 결정에 집중해야 한다. 그만큼 객체지향 프로그램에서는 어떤 것을 객체로 만들 것이냐가 중요하다.

2 클래스

클래스는 객체지향 프로그램에서 데이터를 추상화하는 단위이다. 더 정확히 말하면 클래스는 유사한 객체들(이를 클래스의 인스턴스라고 함)을 정의한 프로그램 모듈이다. 같은 상태와 행위를 가진 모든 객체는 같은 클래스에 속한다.

2.1 클래스의 구성 요소

클래스는 객체와 관련된 다음 두 가지 사항을 포함하고 있다.

- 속성: 각 객체에 저장된 자료의 특성과 이름을 정의한 코드
- 메소드: 객체의 행위를 구현한 프로시저

다음 그림은 은행 시스템에 필요한 객체를 UML로 표현한 것이다. 은행의 사원, 각종 계좌 및 은행 업무가 객체로 표현되어 있다.

NOTE_ UML: 객체지향 설계를 표현하기 위한 그래픽 언어

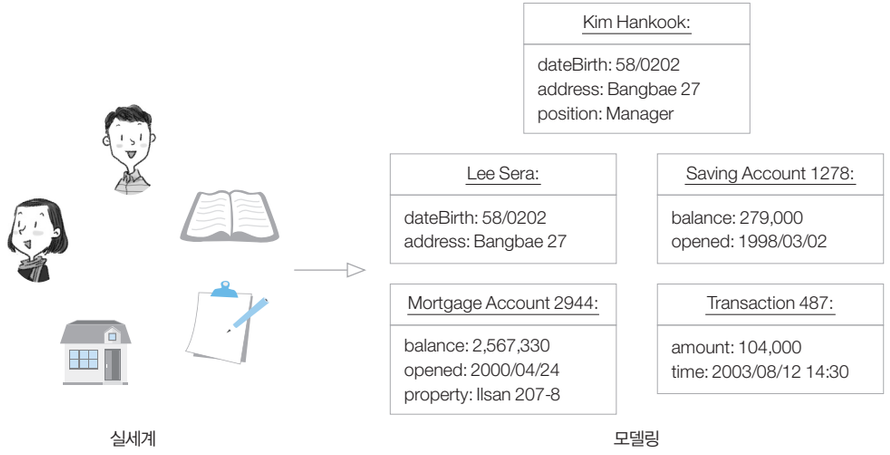
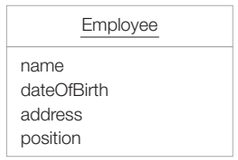


그림 2-4 **은행 시스템에 필요한 객체**

[그림 2-4]에서 Kim Hankook과 Lee Sera는 은행의 사원이며, 이들은 Employee라는 클래스로 묶인다. [그림 2-5]는 Employee 클래스에 속하는 모든 객체가 이름name, 생년월일dateOfBirth, 주소address, 직위position 등의 속성attribute을 가진다는 것을 정의한 것이다. 속성은 객체의 상태를 나타내는 값이다.



(a) Employee 클래스

```
class Employee
{
    public:
        promote(from, to);
        increase_salary(new_salary);
        ...
    prvate:
        char*           name;
        int             dateOfBirth;
        char*           address;
        positionType    position;
}
```

(b) Employee 클래스 정의

그림 2-5 **은행 시스템의 사원 클래스**

그런데 [그림 2-5]에 표현된 Employee 클래스 안에는 name과 address 같은 속성만 정의하는 것이 아니라 새로운 사원을 추가하거나 사원의 이름, 주소, 직위 등을 변경시키는 메소드도 정의해야 한다. 클래스를 찾아내고 속성과 메소드를 정의하는 정적 모델링 방법은 4장에 구체적으로 살펴보겠다.

결국 클래스란 각각의 객체가 가진 속성과 적용 가능한 메소드가 정의되어 있는 일종의 틀template이다. 프로그램에서는 클래스의 인스턴스, 즉 객체를 다음과 같이 생성하여 사용한다.

```
Employee    KimHankook, LeeSera;
```

2.2 클래스가 될 조건

무엇이 클래스가 될 수 있는지를 정하는 것은 초보자에게 매우 어려운 작업이다. 다음의 두 가지 규칙이 도움이 될 것이다.

- 일반적으로 인스턴스를 가질 수 있는 것은 클래스이다.
- 일반적으로 정의된 클래스에 속하는 멤버가 될 수 있는 것은 인스턴스이다.

예를 들어 의료 시스템에서 사용될 Doctor나 Hospital을 생각해보자. Doctor는 진료 기관에 따라 차이가 있지만 여러 명의 의사가 있다면 클래스이다. 그리고 Hospital은 시스템에서 하나만 존재하는 병원 이름 정도로 생각할 수 있으나 여러 병원을 관리하는 의료 시스템이라면 클래스로 정의해야 한다.

예제 2-1 **클래스 찾기**

다음 중 어떤 것이 클래스 또는 인스턴스가 될 수 있는지 판단해보자. 판단 결과 클래스라면 그 인스턴스를 예로 들고, 인스턴스라면 그것이 속할 수 있는 클래스를 제시한다.

① 영화

② 영화 CD

③ 시리얼 넘버 KR-200402가 붙은 영화 CD

④ 대한극장 1번 관에서 7시에 상영하는 영화 〈태극기를 휘날리며〉

정답

① 클래스. 인스턴스의 예는 〈바람과 함께 사라지다〉, 〈미션 임파서블 5〉, 〈국제시장〉

② 클래스. 인스턴스의 예는 구체적인 어떤 영화 동영상이 담긴 CD

③ 인스턴스. 클래스는 영화 CD

NOTE_ 객체지향 관련 문서를 읽을 때나 프로젝트를 진행할 때 '객체'와 '클래스'라는 말이 혼용되는 경우도 많다. "Passenger 객체의 디자인을 끝냈다"고 말했을 때 정확한 의미는 'Passenger 객체'가 아니라 'Passenger 클래스'이다.

2.3 인스턴스

'인스턴스'와 '객체'라는 용어는 같은 의미이다. 인스턴스는 '어떤 클래스에 속하는 특정 사례'라는 뜻으로 관계를 나타내는 용어이다. 예를 들면 '딸'은 관계를 나타내는 단어이지만 '여자 아이'는 독립된 개념의 단어이다. 따라서 인스턴스는 어떤 클래스의 인스턴스로 정확하게 표현하는 것이 좋다. "인스턴스가 메모리에 저장되었다"보다 "어떤 클래스의 인스턴스 또는 특정 객체가 메모리에 저장되었다"가 더 정확한 표현이다. 또한 "Passenger 클래스가 10개의 객체를 가지고 있다"보다 "Passenger 클래스가 10개의 인스턴스를 가지고 있다"가 더 정확한 표현이다.

NOTE_ 클래스 이름 붙이기

객체지향 프로그램에서 가장 중요한 것 중 하나는 클래스를 찾아내는 일이다. 다음은 클래스에 이름을 붙일 때 주의할 사항이다.

■ **단수형 명사 사용**

클래스에 이름을 붙일 때는 Employee, Hospital, Doctor와 같이 영어 명사를 사용하되 단수형 명사여야 한다. 클래스는 개체, 즉 인스턴스를 모아놓은 것이라기보다 인스턴스들의 공통 속성을 정의한 것, 다시 말해 대표성을 지닌 유형을 정의한 것이므로 단수형을 사용한다.

■ **첫 글자는 대문자로**

다른 인스턴스 이름이나 변수 이름과 구별하기 위해 클래스 이름의 첫 글자는 대문자로 하는 것이 좋다. 또한 두 단어 이상을 사용할 경우 단어 사이에 공백 없이 밑줄 문자(_)을 넣거나 뒤 단어의 첫 글자를 대문자로 바꾼다. 예를 들면 Part_Time_Employee나 PartTimeEmployee와 같이 사용한다.

■ **구체적으로 명시**

클래스의 이름은 프로그램의 용도와 범위를 고려하여 되도록 구체적으로 명시하는 것이 좋다. 예를 들면 교통 정보 프로그램을 만들 때 Bus라는 클래스 이름은 버스 차량을 의미할 수도 있고, 버스가 운행되는 일정 노선을 의미할 수도 있다. 따라서 버스 차량은 BusVehicle로, 노선은 BusRoute로 구체적인 이름을 붙이는 것이 좋다.

■ **범위를 제한하는 표현 지양**

클래스의 이름을 너무 제한적으로 정하는 것도 피해야 한다. 예를 들어 본적 주소 클래스의 이름을 City로 정의했다면, 모두 도시에만 사는 것이 아니므로 출생지라는 의미의 Municipality로 정하는 것이 더 좋다.

■ **컴퓨터 시스템과 관련된 단어 지양**

클래스에 이름을 붙일 때 컴퓨터 시스템 내부를 반영하는 단어, 즉 Record, Table, Data, Structure, Information 같은 단어는 사용하지 않는 것이 좋다. 예컨대 고객 정보 클래스는 CustomerData가 아닌 Customer로 정의하면 충분하다.

03 인스턴스 변수, 클래스 변수, 메소드, 오퍼레이션

1 인스턴스 변수

변수는 데이터를 저장하는 장소이다. 클래스는 각 인스턴스에 존재하는 데이터를 저장하기 위해 여러 개의 변수를 정의하는데, 이러한 변수를 인스턴스 변수라 부른다. 인스턴스 변수는 클래스의 속성을 나타내는 변수와 클래스 사이의 관계를 나타내는 변수로 나눌 수 있다.

속성은 객체의 상태를 나타내는 값이다. 예를 들면 Person이라는 클래스는 다음과 같은 속성을 가질 수 있다.

- name(이름)
- dataOfBirth(생년월일)
- idCardNumber(주민등록번호)
- telephoneNumber(전화번호)
- address(주소)

관계association는 클래스의 인스턴스와 다른 클래스의 인스턴스 간 연관 관계를 나타낸다. 예를 들면 특정 회사의 비즈니스 시스템에서 Person 클래스는 다음과 같은 연관 관계를 가질 수 있다.

- supervisor(Person의 관리자인 Manager 클래스와의 연관 관계)
- taskToDo(Person이 하는 작업인 Task 클래스와의 연관 관계)

클래스의 인스턴스들 간 관계 설정은 4장에서 자세히 다루겠다.

NOTE_ 변수와 객체

객체지향 프로그램에서 변수와 객체를 혼동하는 경우가 있는데 변수와 객체는 구별해서 사용해야 한다. 변수는 어떤 시점에 객체를 담고 있을 수도 있고 그렇지 않을 수도 있다. 이를 레퍼런스라고 하는데, 프로그램이 수행되는 동안 변수는 객체 레퍼런스가 변동되기도 한다. 또한 하나의 객체가 여러 변수에 담겨 있는 경우도 있다. 변수의 타입 선언은 변수가 어떤 클래스 타입에 속하는 객체를 담고 있을지 결정하는 것이다.

2 클래스 변수

클래스 안에 var이라는 인스턴스 변수를 선언했다면 클래스에 속하는 모든 객체는 var이라는 이름의 자리가 생기게 된다. 예를 들어 Employee라는 클래스 안에 supervisor라는 인스턴스 변수가 선언되어 있다고 하자. 이 변수 안에 담기는 값은 객체마다 다르다. 즉 Employee에 속하는 각 객체는 supervisor라는 변수 안에 Manager에 속하는 여러 객체를 담게 된다.

클래스를 운영하다 보면 클래스에 속하는 객체들이 공유해야 하는 값이 필요한 경우가 있는데 이를 클래스 변수 또는 정적 변수라 부른다. 클래스에 속하는 객체가 클래스 변수의 값을 저장하면 다른 객체들이 저장한 값에 접근할 수 있다. 주의할 점은 초보자의 경우 클래스 변수를 남용하여 인스턴스 변수를 사용해야 하는 부분에 클래스 변수로 정의하는 것이다. 클래스 변수는 다음과 같은 정보를 저장하는 경우에만 적합하다.

- 클래스 안에 정의한 메소드에 의해 널리 사용되는 디폴트 또는 상수 값
- 특정 클래스 안에 존재하는 알고리즘에 의해 사용되는, 항목을 가려내기 위한 테이블이나 구조체

3 메소드와 오퍼레이션

메소드는 클래스의 행위를 구현하는 데 사용하는 일종의 프로시저이다. 절차지향 프로그램에서 사용되던 프로시저, 함수에 가까운 것으로 클래스 안에 존재한다.

오퍼레이션은 클래스 행위의 타입을 뜻하는 용어로 메소드의 추상적 개념이다(행위를 구현한 코드, 즉 메소드와 독립된 함수 타입을 뜻한다). 다른 클래스 안에 같은 이름과 같은 개념의 오퍼레이션을 다르게 구현한 메소드가 존재할 수 있다. 메소드라는 말은 오퍼레이션을 수행하는 방법이 다르다는 의미를 담고 있다.

3.1 오퍼레이션 다형성

실행 중인 프로그램은 같은 이름의 메소드 중 어떤 것을 호출할지 결정해야 하는데 이를 오퍼레이션의 다형성polymorphic이라 한다. 결정은 호출되는 메소드 앞에 표시된 객체가 어떤 클래스에 속하느냐에 달려 있다. 예를 들어 은행 시스템에서 SavingAccount(예금 계좌)와 MortgageAccount(대출 계좌) 객체의 이자를 계산하는 calculateInterest(이자 계산)라는 오퍼레이션이 있을 때 어떤 객체에 대해 calculateInterest(이자 계산)를 호출했느냐에 따라 해당 메소드가 결정된다.

일단 메소드가 호출되면 해당 객체의 속성에 조작을 가할 수 있다. SavingAccount 객체 KimsAccount를 만든 후 이자율을 적용하여 잔고를 변화시켜보자.

```
KimsAccount      SavingAccount;
KimsAccount.calculateInterest(Date date);
```

여기서 사용한 calculateInterest라는 메소드는 SavingAccount 객체인 KimsAccount의 인스턴스 변수 balance의 값을 변화시킨다. 이를 메시지라고 부르며 [그림 2-6]과 같이 UML로 비주얼화할 수 있다.

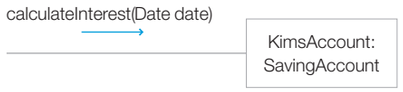


그림 2-6 **메시지 호출의 형상화**

일반적으로 객체 Obj1이 객체 Obj2에게 메시지를 보내려면 다음과 같은 세 가지 조건을 만족해야 한다.

① Obj2를 부르는 이름을 알고 있어야 한다. 즉 Obj2를 변수 중의 하나로 인식하고 있어야 한다.

② Obj2가 가지고 있는 오퍼레이션 이름(위의 예에서는 calculateInterest)을 알고 있어야 한다.

③ 실행시키려는 오퍼레이션의 보조 정보, 매개변수의 타입, 리턴 타입 등을 알고 있어야 한다.

여기서 사용한 calculateInterest라는 메소드는 SavingAccount 객체인 KimsAccount의 인스턴스 변수 balance의 값을 변화시킨다. 이를 메시지라고 부르며 [그림 2-6]과 같이 UML로 비주얼화할 수 있다.

04 상속

1 상속

여러 클래스가 같은 속성, 관계, 오퍼레이션을 가진다면 공통적인 사항을 별도의 클래스에 중복 정의하는 것은 비효율적이다. 따라서 기본 클래스를 정의하고 여기서 파생된 구체적인 클래스로 특수화하는 것이 좋다.

예를 들어 은행의 '예금 계좌 관리' 시스템에서 여러 종류의 계좌를 구현하려면 상속 개념이 적용되어야 한다. 잔고와 예금주 같은 속성은 모든 계좌에 공통적이며, 계좌를 개설하고 말소하는 오퍼레이션 또한 모든 종류의 계좌에 적용된다. 그러나 조금씩 다른 오퍼레이션, 예컨대 주택 마련 대출 계좌 같은 것은 마이너스 잔고가 가능하고 계좌의 종류마다 이자 계산법이 다르다. 따라서 Account를 기본 클래스로 정의하고 대출 계좌MortgageAccount, 적금 계좌InstallmentAccount, 예금 계좌SavingAccount 등과 같이 더 구체화된 서브클래스를 정의한다.

이처럼 여러 개의 서브클래스로부터 공통점을 찾아내어 하나의 슈퍼클래스를 도출하는 과정을 일반화generalization라고 한다. 그리고 일반화 관계를 여러 단계 형성한 것을 상속 계층 구조inheritance hierarchy 또는 is-a 계층 구조라 부른다. 서브클래스는 크게 보아 슈퍼클래스의 일종으로 더욱 구체화된 것이기 때문이다. 그러므로 상속 관계가 있는지를 판단하려면 is-a 관계가 두 클래스 사이에 성립하는지 확인하면 된다.

상속 구조는 [그림 2-7]과 같이 나타낼 수 있다. 화살표의 머리가 슈퍼클래스를 가리키도록 그리고 슈퍼클래스를 위쪽에, 서브클래스를 아래쪽에 배치한다.

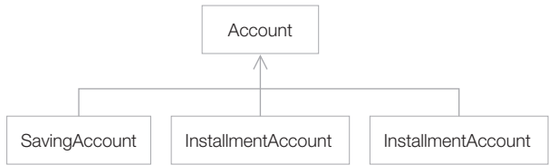


그림 2-7 **은행 계좌의 상속 구조**

상속이란 슈퍼클래스에 정의한 변수와 메소드들을 서브클래스가 묵시적으로 소유하게 하는 것이다. 어떤 클래스를 슈퍼클래스로 정의하고 이에 대한 상속 클래스를 정의하면 클래스에 속한 모든 객체는 자동적으로 상속 관계를 맺는다. 따라서 객체지향 프로그램을 설계할 때 상속 계층 구조를 만들어 상속 관계를 조정한다.

예를 들면 Account의 모든 기능은 SavingAccount, InstallmentAcount, MortgageAccount에 모두 존재한다. [그림 2-8]은 [그림 2-7]의 내용을 구체화한 것으로, Account 클래스에 존재하는 속성과 오퍼레이션이 세 가지 서브클래스에 상속된다는 것을 보여준다. 클래스를 나타내는 상자에서 중간층이 속성을 표시하는 부분이고 하위층이 오퍼레이션을 표시하는 부분이다. 상속된 기능은 서브클래스에 중복해서 표시하지 않고 서브클래스에 새로 추가된 기능만 표시한다.

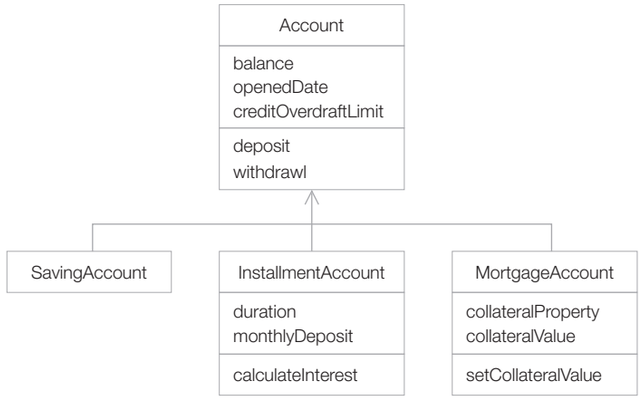


그림 2-8 **속성과 오퍼레이션이 포함된 은행 계좌의 상속 구조**

2 is-a 규칙

클래스를 상속 계층 구조로 구성하는 것은 객체지향 프로그램을 설계하고 구현하는 데 중요한 작업이다. 그런데 설계를 하다 보면 잘못된 계층 구조를 만들기 쉽다. 이때에는 상속 관계를 가진 클래스들이 is-a 규칙을 만족하는지 살펴보아야 한다. 이 규칙은 'an A is a B'가 성립하면 A 클래스가 B 클래스의 서브클래스가 된다는 뜻이다.

예를 들어 '예금 계좌SavingAccount는 은행 계좌Account의 일종이다'는 성립한다. 따라서 예금 계좌는 은행 계좌의 서브클래스가 된다. 하지만 '예금 계좌는 보통 예금의 일종이다'는 성립하지 않는다. 따라서 예금 계좌는 보통 계좌의 서브클래스가 될 수 없다.

is-a 규칙을 만족하지 않는다면 일반화 개념을 잘못 적용한 것이다. 그러나 is-a 규칙을 만족하는 모든 경우를 일반화 개념이 적용된 사례로 볼 수 없다. 다음 예를 살펴보자.

- 슈퍼클래스나 서브클래스 이름이 모호하면 일반화 관계를 정의하기 어렵다. 예를 들어 구체적으로 BusVehicle이라 하지 않고 Bus라고만 한 경우 일반화된 클래스를 발견하기 어렵다.
- 서브클래스는 항구적으로 고유한 속성을 가져야 한다. 예를 들어 OverdrawnAccount를 서브클래스로 만들었을 때, is-a 규칙을 적용하면 '잔고 부족 통장은 통장의 일종이다'라고 말할 수 있다. 그러나 잔고 부족 통장은 잔고를 충분히 채우면 OverdrawnAccount가 아니다. 따라서 OverdrawnAccount는 일반화를 적용하여 별도의 클래스로 만들기에 부적당하다.
- 상속된 모든 기능은 서브클래스에서도 의미가 있어야 한다. [그림 2-8]에 있는 세 가지 서브클래스는 balance(잔고), openDate(개설일), creditOverdraftLimit(마이너스 한도) 속성 모두가 필요한 것이어야 한다. 또한 오퍼레이션 deposit(예금)과 withdrawal(인출)이 각 서브클래스에서 수행되는 것이 의미 있고 일관되어야 한다. withdrawal(인출)이 MortgageAccount(대출 계좌)에서 의미가 없는 것 같지만 처음에 많은 금액을 대출하는 것을 인출로 볼 수 있다.

위의 세 가지 사항을 간과하면 필요하지 않은 상속을 구현하기 위한 조건 체크가 많아지고 이해하기 어려워진다. 결국 상속을 이용하여 일반화 관계를 정의하면 중복 정의를 피하게 되고 재사용이 향상된다. 잘못 설계된 일반화 관계는 문제를 더욱 악화시킬 수 있다.

예제 2-2 **상속 구조 만들기**

다음과 같은 클래스를 상속 구조로 구성해보자.

> 원을 나타내는 Circle, 점을 나타내는 Point, 사각형을 나타내는 Rectangle, 표를 나타내는 Matrix, 타원을 나타내는 Ellipse, 선을 나타내는 Line, 평면을 의미하는 Plane

정답

여러 가지 방법이 있으며 한 예로 다음 그림처럼 구성할 수 있다.

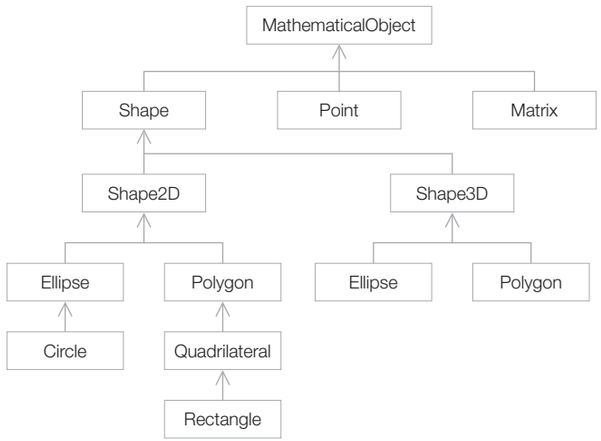


그림 2-9 **도형 관련 객체의 상속 구조**

위 그림에서 제시한 설계에 대해 다음과 같은 추가 의견을 생각해볼 수 있다.

- Point를 Shape의 일종이며 퇴화된 것으로 생각할 수도 있다. Point를 몇 차원으로 표시하느냐에 따라 Point2D, Point3D로 나누어 정의할 수 있다.
- 마찬가지로 Line도 Line2D, Line3D로 나눌 수 있다.
- 수학적으로 원Circle은 타원Ellipse이 지닌 모든 성질을 가지므로 [그림 2-9]와 같이 원을 타원의 서브클래스로 생각할 수 있다. 그러나 타원은 두 개의 초점을 가지고 있고 원은 두 개의 초점이 원점으로 같기 때문에, 타원의 경우 한 초점의 위치를 변경하는 오퍼레이션이 가능하지만 타원의 서브클래스인 원은 이러한 오퍼레이션이 적용될 수 없다. 따라서 원을 타원의 일종으로 정의하여 두 개의 초점이 같은 것으로 정의하지 않고 [그림 2-10]과 같이 원과 타원의 속성을 구별하여 따로 정의하는 방법도 있다. 또 다른 방법은 원을 완전히 배제하고 타원 클래스만 정의하여 사용하는 것이다. 이런 경우 타원을 항상 원으로 만들어주는 constrainAsCircle이라는 속성을 추가한다.

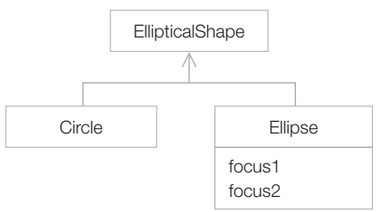


그림 2-10 **타원과 원을 정의하는 다른 방법**

05 다형성

1 다형성

객체지향 패러다임의 장점은 대부분 다형성과 상속 개념에서 비롯된다. 다형성이란 '다양한many 모양form을 가지는 성질'을 뜻한다. 객체지향 패러다임에서 다형성은 다음 두 가지로 정의할 수 있는데 두 가지 모두 객체지향의 장점이 된다.

ⓐ 다형성은 오퍼레이션이나 속성의 이름이 하나 이상의 클래스에서 정의되고 각 클래스에서 다른 형태로 구현될 수 있는 개념이다.

ⓑ 다형성은 속성이나 변수가 서로 다른 클래스에 속하는 객체를 지칭할 수 있는 성질이다.

[그림 2-11]과 같은 여러 가지 다각형을 그래픽 프로그램에서 사용하기 위해 이차원 평면상의 다각형 폴리곤Polygon을 구성하는 문제를 다시 생각해보자.

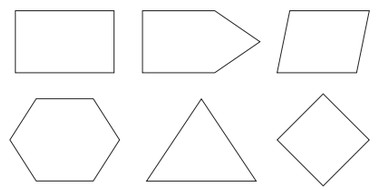


그림 2-11 **이차원 다각형의 예**

폴리곤은 다각형의 면적을 계산하는 getArea 오퍼레이션을 가질 수 있다. 계산 후의 결과는 area라는 속성에 저장되도록 Polygon 클래스에 정의한다. getArea 오퍼레이션은 [그림 2-11]에 보듯이 다각형의 모양에 따라 다양한 알고리즘으로 구현되어야 한다.

다각형에는 다양한 모양이 있지만 일단 삼각형Triangle, 사각형Rectangle, 육각형Hexagon을 Polygon의 서브클래스로 추가해보자. 삼각형, 사각형, 육각형은 일종의 다각형이기 때문에 is-a 관계가 성립한다. 따라서 [그림 2-12]와 같은 상속 관계를 가진다.

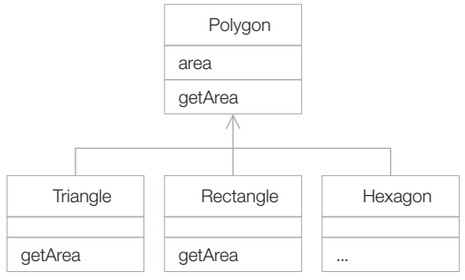


그림 2-12 olygon과 세 개의 서브클래스

Triangle과 Rectangle은 getArea라는 이름의 오퍼레이션을 모두 가진다. getArea 오퍼레이션은 모양에 따라 면적 계산법이 다르지만 Polygon들의 면적을 계산하는 데 같은 이름으로 사용된다. 이것이 같은 모양의 다른 뜻을 가진 다형성이다. Rectangle의 getArea는 Triangle의 getArea 계산법과 다르기 때문에 getArea 메소드의 프로그램 내용이 달라진다. 하지만 오퍼레이션의 이름은 같게 하는 것이 다형성의 핵심이다.

그렇다면 왜 실제적으로는 다른 메소드를 같은 이름으로 만드는 것일까? 예를 들어 twoDShape로 불리는 객체의 면적을 계산하기 위해 다음과 같은 프로그램을 작성했다고 하자.

```
twoDShape.getArea;
```

면적을 구하는 메소드 중 어떤 것이 사용되었는지 알 수 없다. twoDShape이 어떤 클래스에 속하는 객체인지 실행 전에는 정확히 알 수 없기 때문이다. 결과적으로 다음의 다섯 가지 경우가 가능하다.

① twoDShape이 Triangle의 인스턴스인 경우: Triangle에 속한 getArea가 수행될 것이다.

② twoDShape이 Rectangle의 인스턴스인 경우: Rectangle에 속한 getArea가 수행될 것이다.

③ twoDShape이 Hexagon의 인스턴스인 경우: Hexagon에 getArea 오퍼레이션이 정의되어 있지 않으므로 Polygon에 정의된 getArea를 상속받아 수행될 것이다.

④ twoDShape이 일반적인 임의의 Polygon의 인스턴스인 경우: Polygon에 정의된 getArea 오퍼레이션이 수행될 것이다.

⑤ twoDShape가 위의 네 가지 클래스가 아닌 어떤 클래스 C에 속하는 객체인 경우: 클래스 C에는 getArea 오퍼레이션이 정의되어 있지 않아 컴파일 오류로 체크될 것이다.

이때 메시지를 보내려는 대상 객체를 모르고 보낼 수 있냐는 의문이 생긴다. 이런 일은 실행 과정에서 흔히 발생하는데 다음 예를 살펴보면 이해가 될 것이다. Polygon 타입으로 정의된 p 변수가 컴파일 타임에는 어떤 객체가 될지 알 수 없다. 실행될 때 사용자의 선택에 따라 결정되기 때문이다.

```
Polygon   p;
Triangle  t=new Triangle();
Hexagon  h=new Hexagon();
…
if(user says OK)
     p=t;
else
     p=h;
…
p.getArea;     // p may refer to a Triangle or Hexagon
```

위 프로그램에서는 p.getArea가 어떤 버전의 getArea 오퍼레이션을 실행하는지 알아보기 위해 테스트하는 것도 무의미하다. p는 Polygon을 상속한 어떤 서브클래스의 객체가 되더라도(예를 들면 Octagon) 원시코드를 변경하지 않고 수행될 수 있다. 타깃 객체가 면적을 계산할 수 있는지가 관건이며 메시지를 보내는 프로그램은 걱정할 필요가 없다.

또 한 가지 주의할 점은 Polygon p; 문장이다. 이 문장의 의미는 p라는 변수가 너무 많은 형태로 변하는 것에 대한 안전한 제한 장치라고 할 수 있다. 구문의 의미를 살펴보면 변수 p는 Polygon 클래스의 객체나 Polygon 서브클래스의 객체를 담을 수 있다. 만일 p가 완전히 다른 Customer 클래스의 객체를 갖게 된다면 런타임 오류가 발생한다.

다형성의 정의 ⓐ에 명시된 것처럼 getArea 오퍼레이션은 여러 클래스 안에서 정의될 수 있다. 다형성의 정의 ⓑ는 위의 사례에서 변수 p가 다른 여러 클래스, Triangle이나 Hexagon 등의 객체를 지칭할 수도 있다는 것을 의미한다. 지금까지 사례에서 살펴본 다형성의 두 가지 측면은 객체지향 프로그램을 간결하게 만드는 데 기여한다.

객체지향에서 다형성은 동적 바인딩dynamic binding으로 실현된다. 동적 바인딩은 실행되는 정확한 코드가 런타임(컴파일 타임과 반대되는)에 결정되는 기법이다. 즉 메시지를 보내는 타깃 객체가 프로그램의 최종 실행 단계에 가서 확정되는 개념이다.

2 함수 재정의

[그림 2-12]에서 getArea가 Polygon에서도 정의되고 Triangle에서도 정의되었는데 이를 함수 재정의overriding라 한다. 함수 재정의는 슈퍼클래스에서 정의된 메소드가 서브클래스에서 다시 정의되는 것을 말한다. 원래 Polygon에서 정의된 getArea 오퍼레이션은 Triangle에서 재정의되었다. getArea 오퍼레이션은 같은 이름을 가지고 있으나 다른 알고리즘으로 구현된다. 이렇게 같은 이름의 다른 뜻을 가진 오퍼레이션을 서브클래스에 다시 정의하는 것은 '재정의'라고 일컫는다.

경우에 따라서 슈퍼클래스에서 정의된 오퍼레이션을 서브클래스에서 취소시키기 위해 재정의를 사용하기도 한다. 이때 재정의한 오퍼레이션의 구현에는 오류를 리턴하는 문장이 포함된다. 하지만 과다하게 사용되면 상속 구조를 뒤흔드는 결과가 되기 때문에 피하는 것이 좋다.

재정의와 관련하여 혼동하기 쉬운 개념으로 오버로딩overloading이 있다. 오버로딩은 같은 클래스 안에 이름이 같은 여러 오퍼레이션을 정의하는 것이다. 다형성과 오버로딩 모두 수행될 오퍼레이션을 런타임에 선택하게 하는 방법이다. 그러나 다형성은 같은 이름의 오퍼레이션을 다른 여러 클래스에 정의하는 것이고, 오버로딩은 같은 이름의 오퍼레이션을 같은 클래스 안에 여러 번에 걸쳐 다르게 정의한다는 것이 차이점이다.

예제 2-3 다형성 이해하기

[예제 2-2]에서 살펴본 이차원 평면Shape2D의 다각형 클래스를 상속 구조로 구성하고 속성과 오퍼레이션을 정의해보자.

정답

[그림 2-13]과 같이 네 가지 수준의 계층 관계로 일반화할 수 있다. 포함된 클래스에 대한 자세한 설명은 다음과 같다.

- 타원Ellipse은 장축과 단축, 두 개 축의 길이로 정의된다. 장축의 반을 semiMajorAxis라고 하며 원은 반지름과 같다.
- 정다각형RegularPolygon은 모든 꼭짓점이 같은 원주상에 있고 각 변의 길이가 같은 것이다. 정삼각형, 정사각형, 정오각형이 여기에 해당된다.
- 부정다각형ArbitraryPolygon은 정다각형이 아닌 것으로 꼭짓점의 위치로 정의된다.

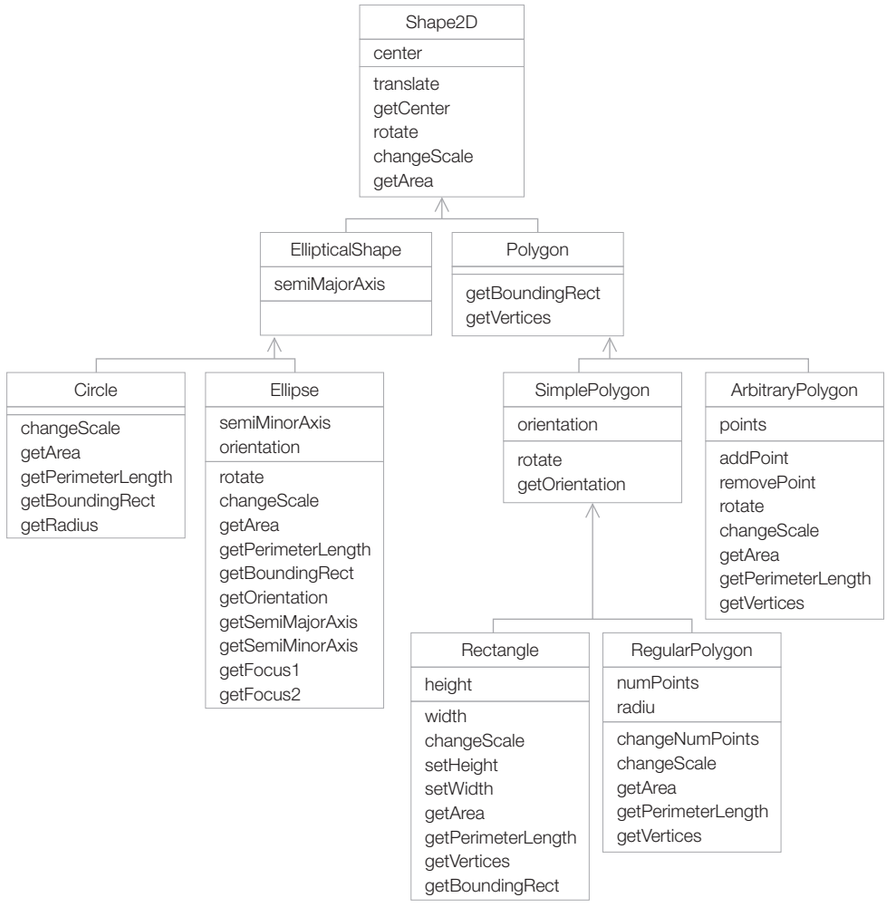


그림 2-13 Shape2D 클래스의 상속 구조

클래스 Shape2D는 일곱 개의 오퍼레이션(translate, getCenter, rotate, changeScale, getArea, getPerimeterLength, getBoundingRect)을 정의했다. 일곱 개의 오퍼레이션을 여기에 정의한 이유는 하부에 있는 여덟 개의 서브클래스에서도 오퍼레이션이 모두 의미 있고 동작할 수 있기 때문이다. 서브클래스에서는 모두 의미가 다르므로 다른 메소드로 구현될 것이다. 클래스 Shape2D에 있는 오퍼레이션의 수행 내용은 다음과 같으며, 그 효과는 [그림 2-14]에 나타냈다.

- translate: x 좌표 값과 y 좌표 값, 두 개의 매개변수를 받아 x축, y축을 기준으로 이동한다.
- getCenter: center 값을 반환한다.
- rotate: 한 개의 매개변수를 받아 주어진 각도만큼 도형을 회전시킨다.
- changeScale: 한 개의 매개변수 퍼센트 값을 받아 비율만큼 크거나 작게 만든다.
- getArea: 면적을 계산하여 리턴한다.
- getPerimeterLength: 둘레를 계산하여 리턴한다.
- getBoundingRect: 도형의 주위를 충분히 포함하는 큰 사각형을 리턴한다.

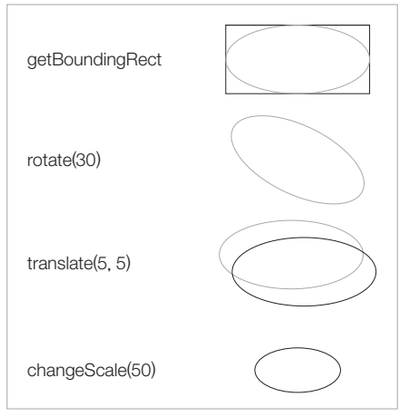


그림 2-14 Ellipse에 대한 오퍼레이션 결과

3 추상 클래스

[그림 2-13]의 네 개 클래스(Circle, Ellipse, SimplePolygon, ArbitraryPolygon)를 보면 rotate 오퍼레이션을 따로 정의하고 있다. 각 클래스에 고유한 rotate 메소드를 살펴보자.

- Circle: 원은 돌리더라도 모양의 변화가 전혀 없기 때문에 rotate 메소드는 의미가 없다. 즉 메소드는 존재하지만 내용이 없다.
- Ellipse, SimplePolygon: 이 도형은 회전시키는 각도, orientation 속성이 필요하다.
- ArbitraryPolygon: 회전시키는 방법이 더 복잡하다. 즉 회전 후 각 변을 그리는 방법을 SimplePolygon과 다르게 접근해야 하기 때문이다.

이처럼 rotate 오퍼레이션을 구현하는 방법이 다 달라서 오퍼레이션의 구현을 슈퍼클래스인 Shape2D에 할 수 없다. 오퍼레이션의 정의는 Shape2D에 하더라도 회전시키는 방법이 다르기 때문에 메소드를 서브클래스에 구현해야 한다. 이렇게 오퍼레이션 정의만 하는 것을 가상함수virtual function라 부른다.

Shape2D, EllipticalShape, Polygon, SimplePolygon은 개념만 가진 클래스로 구체적인 인스턴스를 가질 수 없는데 이를 추상 클래스라 한다. [그림 2-13]과 같은 상속 계층을 나타내는 트리에서 단말 노드에 있는 클래스들은 구체적인 객체를 가질 수 있으나 중간 노드에 있는 클래스들은 추상 클래스로 선언해야 한다. 따라서 추상 클래스의 주요 목적은 서브클래스에 의해 상속받을 기능을 정의하는 것이다.

[그림 2-13]에서 발견할 수 있는 추상 클래스에 대한 사실은 다음과 같다.

- rotate와 더불어 클래스 Shape2D에는 두 가지 가상함수가 있다. 가상함수는 필요하면 단말 노드에 있는 클래스에 도달하기 전에 오퍼레이션을 구현할 수 있다. 예를 들면 SimplePolygon의 rotate는 하위 서브클래스에 공통 알고리즘이 적용되므로 SimplePolygon에 정의된다.
- SimplePolygon은 구체화된 두 개의 메소드가 있지만 추상 클래스이다. changeScale, getArea, getPerimeterLength와 같은 오퍼레이션의 구체적인 구현을 상속받을 수 없기 때문이다.
- 클래스 Shape2D에는 가상함수 getBoundingRect가 존재한다. 이 함수는 Polygon에서 구체화되어 구현할 수 있다. 도형의 꼭짓점을 계산할 수 있는 getVertices 메소드가 존재하므로 도형을 감싸는 사각형을 구할 수 있고, 따라서 Polygon에서 구현할 수 있다.

NOTE_ 객체지향의 역사

객체지향은 한순간에 탄생한 것이 아니라 여러 사람이 오랫동안 연구하여 얻은 산물이다. 오늘날 객체지향 개념이 사용되기까지 기여한 사람들을 통해 객체지향의 역사를 살펴보자.

- **콘스탄틴**Constantine

  객체지향 개념과 직접 관련된 것은 아니지만 1960년대에 좋은 소프트웨어 설계의 근본적인 기준이 무엇인지 연구하여, 프로그래밍하기 전에 소프트웨어를 설계할 수 있다는 것을 제안했다. 그때 콘스턴틴이 제안한 응집과 결합 개념이 객체지향 패러다임에서 실현되고 있다.

- **올레 요한 달**Ole-Johan Dahl**과 크리스텐 니가드**Kristen Nygaard

  1966년 시뮬라Simula라는 시뮬레이션 언어를 개발하면서 오늘날 객체지향 패러다임의 중요한 개념, 예를 들면 클래스와 같은 것들을 소개했다.

- **앨런 케이**Alan Kay**와 아델 골드버그**Adele Goldberg

  1970년 제록스 팰로앨토연구소에서 스몰토크Smalltalk라는 최초의 객체지향 언어를 개발하여 객체지향 패러다임을 실현했다. 메시지, 상속 등 대부분의 주요 개념이 여기서 제안되었으며, 스몰토크는 객체지향 개념에 충실한 언어로 아직도 사용되고 있다.

- **다익스트라**Dijkstra

  오류 없는 소프트웨어를 만들기 위해 증명에 대한 연구를 많이 했고, 의미적 분리를 위해 추상화 수준에 따라 소프트웨어를 구축하는 아이디어를 내놓았다. 객체지향에서 중요한 개념인 캡슐화를 제안했다.

- **리스코프**Liskov

  1970년대에 추상 데이터 타입에 대한 개념을 제안하여 객체지향의 기초를 다졌다. CLU라는 언어를 개발하여 내부 데이터를 숨기는 아이디어를 제안했다.

- **파르나스**Parnas

  1972년 모듈화 원리를 제안한 기념비적인 논문을 작성하여 객체지향 구문 형성에 공헌했다. 특히 정보 은닉이란 개념을 제안하여 객체지향 시스템에 적용하게 했다.

- **비야르네 스트로스트룹**Bjarne Stroustrup

  ++ 언어를 탄생시킨 장본인이다. 마틴 리처드Martin Richard의 BCPL이 B라는 언어로 발전하고 스트로스트룹이 B를 C로, 그리고 다시 C++로 발전시킨 것이다. 스트로스트룹은 어셈블러, C 언어를 사용하던 동료들이 좀 더 쉽게 프로그램을 작성하도록 C++ 언어를 창안했다. 자바가 소개된 후에도 많이 사용하던 C 언어와의 이식성, 호환성 때문에 여전히 많이 사용하고 있다.

- **베르트랑 메예르**Bertrand Meyer

  메예르는 컴퓨터 과학의 대표적인 아이디어를 객체지향이란 개념으로 잘 묶어놓았다. 그 결과 에펠Eiffel이라는 객체지향 언어를 개발했다. 에펠은 독특하게도 학계와 산업계 모두에서 좋은 반응을 얻고 있다.

- **그래디 부치**Grady Booch**, 이바르 야콥손**Ivar Jacobson**, 제임스 럼보**James Rumbaugh

  부치, 야콥손, 럼보는 협력하여 객체지향 설계를 표현하기 위한 그래픽 언어, UML Unified Modeling Language을 표준화했다. 1990년대 초까지 제각각 있던 다양한 객체지향 설계 표현 방법이 이들에 의해 단일화되었다.

06 UML

설계 작업에서는 복잡한 아이디어를 간결하고 정확하게 표현하여 이를 교환해야 한다. 특히 매우 다양한 기술과 문화적 배경을 지닌 기술자들이 참여하는 프로젝트에서 정확하고 간결하게 표현하는 것은 잘못된 의사소통을 줄이는 데 필수적이며 중요한 요건이다.

정확히 의사 전달을 하려면 먼저 표현 방법이 나타내는 의미semantic를 잘 정의해야 한다. 또한 시스템의 특성을 표현하는 데 적합해야 하고 프로젝트 참여자들이 이해하기 쉬워야 한다. 복잡한 의미를 잘 표현하려면 약자와 심벌을 사용하는 것이 효과적이다. 사람의 단기 기억 능력에는 한계가 있다. 상세한 것을 일차원적인 문자로 나타낸 것보다 그 내용을 적절히 숨겨 단계적으로 추상화하는 것이 의사 전달에 효과적이다. 심벌과 계층적 추상화를 이용하면 복잡한 문제를 다루기 쉽고 잘 표현할 수 있다.

또한 표현 방법의 표준이나 규격이 필요하다. 누구나 같은 표현 방법을 사용한다면 잘못되거나 모호한 해석이 없을 것이다. 반면에 유사한 표현 방법이 여러 가지 존재한다면 서로 해석이 달라 잘못된 이해를 유발하기 쉽다.

이 책에서는 객체지향 설계의 표현 방법으로 UML을 소개한다. UML은 심벌의 의미가 잘 정의되어 있고 시스템의 다양한 특성을 표현할 수 있어서 객체지향 설계 표현 방법의 표준으로 사용되고 있다.

1 UML

UML Unified Modeling Language은 객체지향 소프트웨어를 모델링하는 표준 그래픽 언어로, 심벌과 그림을 사용하여 객체지향 개념을 나타낼 수 있다. UML은 '나타내는 방법notation'이므로 소프트웨어 제품도 아니며 소프트웨어 도구도 아니다. 또한 방법론methodology도 아니다. 1988년부터 제안되어 사용된 여러 가지 객체지향 방법론은 모두 객체지향으로 모델링하는 과정modeling process과 모델링에 사용하는 언어modeling language를 정의하고 있다. 모델링 과정은 객체지향으로 분석하고 설계하는 프로세스를 말하고, 모델링 언어는 설계를 표현할 때 사용하

는 그래픽 심벌을 의미한다. UML은 프로세스나 방법을 안내하지 않고 표현 방법만을 제시할 뿐이다.

UML은 건축물의 설계 도면이 다양한 것처럼 소프트웨어에 대한 다양한 관점을 제공한다.

- 기능적 관점functional view: 사용자 측면에서 본 소프트웨어의 기능을 나타낸다. 즉 소프트웨어가 바깥 세계와 어떻게 상호작용하는지를 나타낸다. 사용 사례 모델링이라고도 부르며, 주로 요구 분석 단계에 사용한다.
- 정적 관점static view: 소프트웨어 내부의 구성 요소와 그것들 사이의 구조적 관계를 나타낸다. 클래스 및 그것들 사이의 관계, 패키지 안의 클래스 구성과 패키지 사이의 관계 등을 예로 들 수 있다.
- 동적 관점dynamic view: 소프트웨어의 내부 동작을 나타낸다. 각 요소가 시스템의 요구를 만족시키기 위해 어떻게 동작하고 상호작용하는지를 나타낸다.

2 UML 다이어그램

UML에는 여러 가지 다이어그램이 있다. 각 다이어그램은 개발될 소프트웨어에 대한 고객과 개발자의 관점을 다양하게 표현한 것이다. 예를 들어 건물을 설계할 때 평면도, 입면도, 조감도, 온수 배관도, 전기 배선도 등을 그리는데, 이는 건물을 보는 위치와 관점에 따라 차이가 있기 때문이다. 즉 소프트웨어의 복잡한 요소 중에서 관심 있는 요소를 초점으로 간략하게 그린 것이 다이어그램이다.

다음은 UML의 주요 다이어그램을 요약하여 설명한 것이다.

■ 사용 사례 다이어그램

액터와 사용 사례를 이용하여 시스템의 기능을 모델링하는 데 사용한다. 개발하려는 시스템의 기능적 요구 또는 업무 프로세스의 개관을 나타낸다.

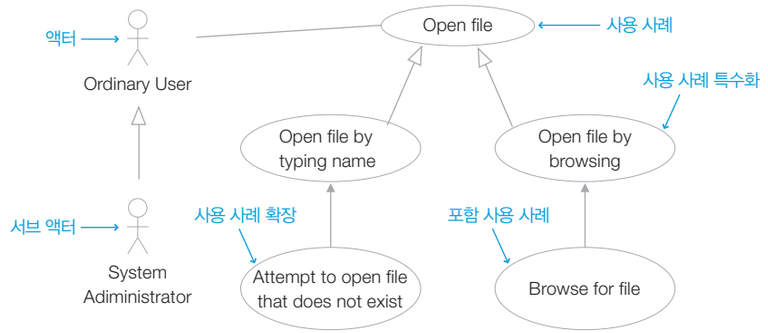


그림 2-15 **사용 사례 다이어그램의 예**

■ 클래스 다이어그램

객체지향 시스템의 가장 근간이 되는 다이어그램으로 시스템의 정적인 구조를 나타낸다. 또한 도메인(문제 영역)의 개념과 그것들 사이의 관계를 표시한다.

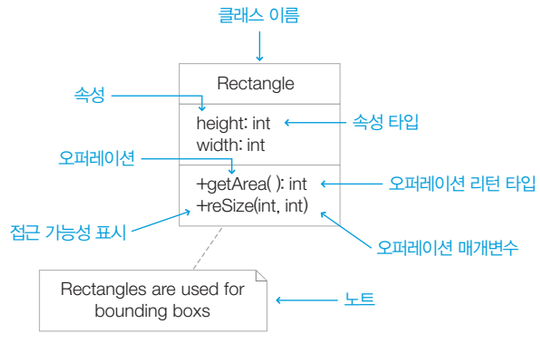


그림 2-16 **클래스 다이어그램의 예**

■ 시퀀스 다이어그램

객체 사이의 메시지 교환을 시간이 흐름에 따라 나타낸 것이다. 즉 사용 사례로 표시된 업무 프로세스에 대하여 시스템 안에 존재하는 객체가 어떤 식으로 개입하여 상호 작용하는지를 나타낸다.

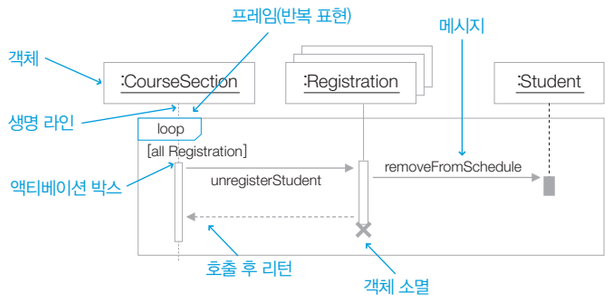


그림 2-17 **시퀀스 다이어그램의 예**

■ 상태 다이어그램

외부 자극에 대한 시스템의 동적 상태 변화를 나타낸다. 외부 이벤트에 대하여 민감하게 상태를 변화시키는 객체를 모델링한다.

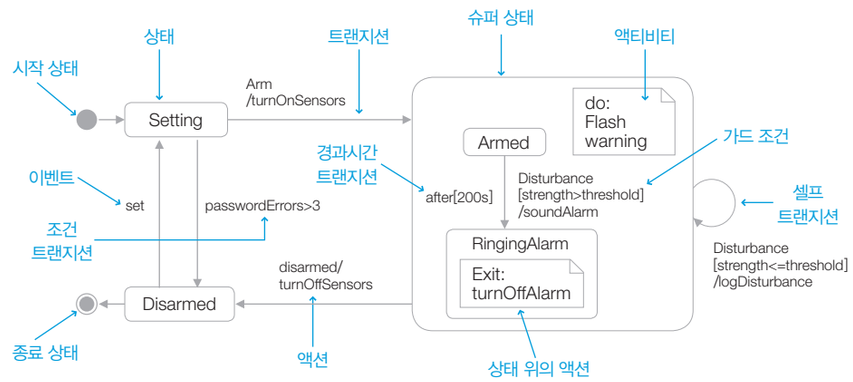


그림 2-18 **상태 다이어그램의 예**

■ 액티비티 다이어그램

시스템의 내부 프로세스를 단계별 작업 흐름 형태로 모델링한다. 시스템의 동적 특징을 나타낸다.

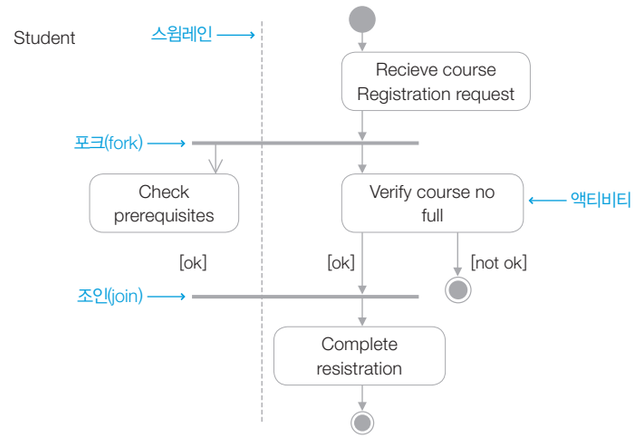


그림 2-19 **액티비티 다이어그램의 예**

■ 패키지 다이어그램

관련된 클래스를 패키지로 그루핑하여 복잡한 시스템을 조직화하는 데 사용한다.

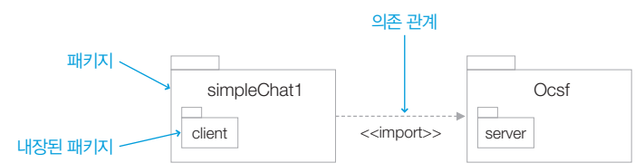


그림 2-20 **패키지 다이어그램의 예**

■ 배치 다이어그램

노드, 컴포넌트, 커넥터 등 시스템의 물리적 자원 배치를 나타낸다.

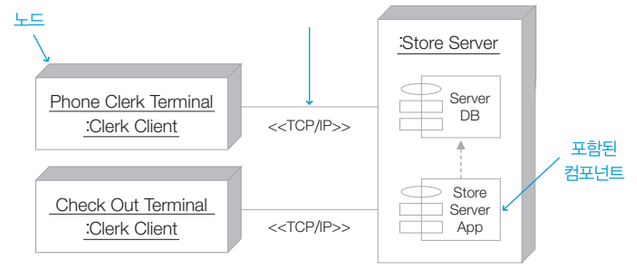


그림 2-21 **배치 다이어그램의 예**

UML 다이어그램의 목적은 소프트웨어 개발에 도움을 주는 것이다. 개발 과정에 여러 가지 다이어그램으로 설계하면 구조가 좋아지고 개발자들 사이에 의사 교환이 쉬워진다. 그러나 명확한 이해 없이 UML을 사용하면 오히려 객체지향 소프트웨어 개발에 방해가 될 수도 있다.

UML은 방법론이 아니기 때문에 단계별로 어떻게 작업해야 하는지 자세하게 나타내지 않는다. 래셔널Rational 사에서 제안한 Unified ProcessRUP와 같은 것이 대표적인 객체지향 개발 방법론이다.

3 UML의 특징

UML은 소프트웨어를 시각적으로 모델링하기 위해 사용한다. 객체지향 소프트웨어 개발에서는 모델이 매우 중요하다. 소프트웨어 시스템을 표현하고 검증하기 위해 모델을 사용하는 데 UML이 사용된다. UML은 다음과 같은 특징을 지니고 있다.

■ **비주얼화**

UML은 소프트웨어의 구성 요소와 그것들의 관계 및 상호작용을 시각화한 것이다. 즉 UML은 다이어그램을 구성하는 그래픽 요소들의 집합이다. 각 다이어그램은 소프트웨어의 비주얼 표현이며 소프트웨어에 대한 관점을 가진다.

■ **명세**

UML은 소프트웨어 개발의 중요한 작업인 분석, 설계, 구현 작업의 정확하고 완벽한 모델을 제공한다. 세 가지 뷰 모델은 개발 사이클에 반복적으로 사용된다. 분석 단계에서는 기능적 모델이 개념적인 수준으로 작성되고, 설계 단계에서는 동작 수준의 모델로, 구현 단계에서는 더 자세한 상호작용 모델 수준으로 명세된다.

■ **구축**

UML 모델은 객체지향 언어와 호환이 된다. UML은 프로그래밍 언어가 아니지만 모델이 객체지향 언어로 매핑될 수 있다. 따라서 UML 기반 모델링과 개발 도구는 어느 정도의 코드 자동 생성과 코드에서 모델을 추출하는 역공학을 제공한다. 따라서 비주얼 모델에 대한 변경을 코드에서 추적할 수 있으며, 거꾸로 코드의 변경을 모델에서 추적할 수도 있다.

■ **문서화**

UML은 소프트웨어 생명주기의 중요한 개발 작업을 추적하고 문서화하는 데 도움을 준다. UML 모델링은 개발 프로젝트에 포함된 작업으로 문서화, 명세화 작업과 같기 때문에 문서화 작업 중 모순되는 부분을 줄일 수 있다.

■ **테스트 기준**

UML 모델링에 의한 명세는 구현 결과에 대한 테스트 기준이 된다. 즉 설계에서 의도한 내용이 코딩 결과에 잘 담겨 작동하는지 검사하는 데 사용된다.

이와 같이 UML 모델은 설계 작업을 도와줄 뿐만 아니라 설계를 분석하고 검토하는 작업에도 사용된다. 따라서 시스템을 기술하는 문서의 핵심이라 할 수 있다. 이 책에서는 고품질의 다이어그램과 모순 없는 모델을 생성하는 여러 가지 방법을 소개할 것이다.

예제 2-4 **문제 은행 만들기**

지금까지 학습한 객체지향 개념을 적용하여 다음 문제에 적용해보자. 시험 문제를 출제할 때 사용할 문제 은행을 만드는 것이다.

정답

객체지향 언어로 구현하기 전에 먼저 정적 모델링 작업을 해 보자. 문제 은행 시스템은 [그림 2-22]와 같이 크게 두 부분으로 나눌 수 있다.

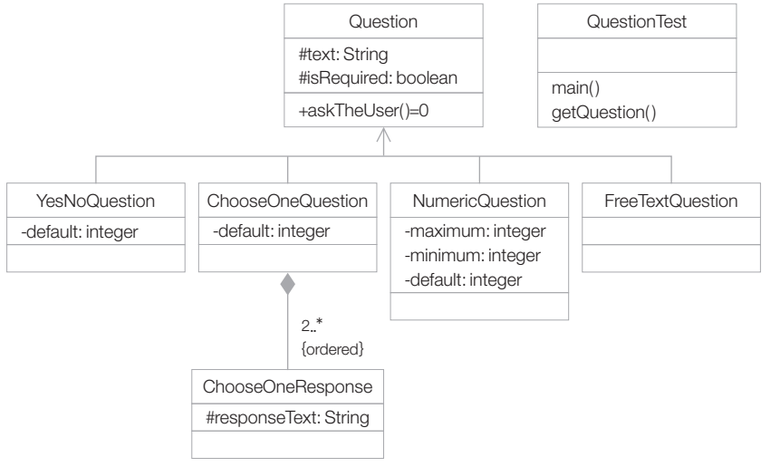


그림 2-22 **문제 은행 시스템을 위한 클래스**

먼저 여러 가지 형태의 문제 유형을 표현할 상속 구조를 찾아낸다. 다중 택일 유형뿐만 아니라 정오형, 숫자 단답형, 서술형을 다룰 수 있어야 한다면 [그림 2-22]와 같이 Question 클래스를 다양하게 세분하여 정의한다. 다음으로 필요한 것은 앞에서 정의한 Question 클래스에 담겨 있는 유형을 이용하여 문제를 만드는 프로그램, QuestionTest이다.

- Question 클래스를 상속하는 여러 유형의 문제에 대한 클래스 정의는 다음과 같다.
- Question 클래스는 추상 클래스로서 인스턴스가 생성될 수 없으며, 문제의 질문을 저장하는 text 필드와 필수 여부를 표시하는 isRequired 필드를 가진다. 또한 askTheUser()는 각 문제의 질문을 생성하는 함수로 서브클래스에서 구체적으로 유형에 맞게 구현되어야 한다.
- YesNoQuestion은 간단한 정오를 답하는 문제로, default에 문제의 정답을 저장해둔다.
- ChooseOneQuestion은 다중 선택형 문제로, 정답은 default에 저장되고 각 문항은 ChooseOneResponse에 저장된다. ChooseOneResponse는 2개부터 n개까지 다양하게 지정할 수 있다.
- FreeTextQuestion은 주관식 답변형 문제로, 답을 텍스트로 보관할 수 있는 필드가 필요하다.

QuestionTest 클래스는 Question 클래스를 이용하여 문제를 만드는 클라이언트 프로그램이다. 이 함수의 main() 함수에서는 만든 문제를 화면이나 프린터 등의 출력 장치에 출력하는 작업을 한다. getQuestion() 함수에서는 여러 유형의 실제 문제를 만들어내는 작업을 한다. Question 타입의 객체 배열을 문항 수만큼 생성하여 각각 문제 유형의 서브클래스 객체로 만들어넣는다. QuestionTest 클래스는 일종의 사용자 인터페이스를 담당하는 클래스이다.

다음은 위에서 설계한 문제 은행을 자바 프로그래밍 언어로 구현한 것이다.

```
package question;
abstract class Question  { // Full class name is question. QuestionTest
    public Question( String _text ) // Constructor     {
        theText = _text;
    }
    public abstract void askTheUser();
    protected String theText;
}

class YesNoQuestion extends Question {
    public YesNoQuestion( String _text ) { super( _text ); }
    public void askTheUser() {
        System.out.println( theText );
        System.out.println( "YES or NO ...?" );
    }
}

class FreeTextQuestion extends Question {
    public FreeTextQuestion( String _text ) { super( _text ); }
    public void askTheUser() {
        System.out.println( theText );
        System.out.println( "Well...? What's the answer...?" );
    }
}

public class QuestionTest {
    public static void main( String[] args ) {
        Question[] questions = getQuestions();
        for( int i = 0; i < questions.length; i++ ) {
              questions[ i ].askTheUser(); // Polymorphism !!!
        }
    }

    private static Question[] getQuestions() {
        Question[] qs = new Question[ 2 ];
        qs[0] = new YesNoQuestion( "Do you understand polymorphism?" );
        qs[1] = new FreeTextQuestion( "Why is polymorphism good?" );
        return qs;
    }
}
```

요약

- 객체지향의 중요한 원리는 소프트웨어가 데이터와 함수가 묶인 형태인 클래스로 구성된다는 것이다. 클래스의 실체인 객체들이 서로 메시지를 보내 원하는 작업을 수행하므로 복잡한 문제에 쉽게 접근할 수 있다.
- 객체지향 패러다임은 상속 계층과 다형성이라는 개념에서 비롯된다. 상속을 사용함으로써 클래스의 정의를 재사용할 수 있고, 다형성을 통해 프로그램의 복잡한 타입 체크를 피할 수 있다. 이런 강력한 기능을 남용한다면 유지보수가 어려워지므로 정확하게 이해하고 사용해야 한다.
- UML은 객체지향 소프트웨어를 표현하는 그래픽 언어 표준이다. UML의 다양한 다이어그램을 통해 객체지향 소프트웨어를 모델링하고, 그 결과를 기준으로 삼아 소프트웨어를 구현하면 효과적으로 개발할 수 있다.

연습문제

01 객체지향 기법에 대한 설명으로 거리가 먼 것은?

① 프로시저에 근간을 두고 프로그래밍을 구현하는 기법이다.
② 현실 세계를 모형화하여 사용자와 개발자가 쉽게 이해할 수 있다.
③ 소프트웨어의 재사용률이 높아진다.
④ 소프트웨어의 유지보수성이 향상된다.

02 객체지향 기법에서 다음 설명에 해당하는 것으로 가장 타당한 것은?

- 다른 객체에게 자신의 정보를 숨기고 자신의 연산만을 통해 접근한다.
- 유지보수와 소프트웨어 확장 시 오류를 최소화할 수 있다.

① abstraction
② information hiding
③ inheritance
④ polymorphism

03 C++ 언어에서 두 개의 클래스 A , B가 아래 보기와 같이 상속 관계에 있다고 할 때, 다음 설명 중 옳지 않은 것은? (다만, A의 데이터 멤버와 멤버 함수는 모두 public으로 정의되어 있다.)

```
class B : A {
...
};
```

① A를 상위 클래스라고 한다.
② B를 하위 클래스라고 한다.
③ A가 B에 상속될 때 삭제되는 A의 멤버 함수는 없다.
④ 위 보기의 상속은 멤버 함수와 데이터 멤버들의 재사용을 목적으로 한 것이다.
⑤ B가 사용되던 자리에 A를 대체해도 시스템은 이상 없이 작동한다.

04 객체지향 원리에 대한 설명으로 옳지 않은 것은?

① 인스턴스는 한 클래스의 모든 속성을 상속받고, 거기에 자신의 속성을 추가하여 만들어진 서브클래스이다.
② 캡슐화는 데이터와 그것의 처리에 관한 작업이 하나로 모아져 있는 것이므로 객체지향 원리만의 특징이라 할 수 없다.
③ 상속은 객체지향 원리의 특성이나, 다중 상속이 적용된 경우 또는 상속 깊이가 깊은 경우에는 개발 및 유지보수에 어려움이 따를 수 있다.
④ 다형성과 동적 바인딩은 유지보수에 부정적인 영향을 줄 수 있다.

05 객체지향 소프트웨어 개발에 관한 설명 중 옳지 않은 것은?

① UML은 객체지향 분석 및 설계 시에 사용되는 객체 관리 그룹OMG 표준 모델링 언어이다.
② 객체지향 기법 사용의 혜택 정도는 구현 단계에서 객체지향 언어의 사용 여부보다는 성공적인 객체지향 분석 및 설계에 달려 있다.
③ 객체지향 기법에서 객체와 객체 사이의 통신은 메시지 패싱을 통해 이루어진다.
④ 객체지향에서 각 객체는 관련 함수와 다른 객체와의 통신하기 위한 인터페이스로 구성된다.

06 객체지향 개념에서 연관된 데이터와 함수를 함께 묶어 외부와 경계를 만들고 필요한 인터페이스만을 밖으로 드러내는 과정을 무엇이라고 하는가?

① 메시지 ② 캡슐화 ③ 상속 ④ 다형성

07 객체지향 개념 중 하나 이상의 유사한 객체들을 묶어 공통된 특징을 표현한 데이터 추상화를 의미하는 것은?

① 메소드 ② 클래스
③ 상속성 ④ 메시지

08 객체지향 기법에서 캡슐화encapsulation에 대한 옳은 내용을 모두 나열한 것은?

㉠ 캡슐화를 하면 객체 간의 결합도가 높아진다.
㉡ 캡슐화된 객체들은 재사용이 용이하다.
㉢ 프로그램 변경에 대한 오류의 파급 효과가 적다.
㉣ 인터페이스가 단순해진다.

① ㉠, ㉡ ② ㉠, ㉢, ㉣
③ ㉡, ㉢, ㉣ ④ ㉠, ㉡, ㉢, ㉣

09 객체지향 기법 중 다음 설명이 의미하는 것은?

객체의 성질을 분해하고 공통된 성질을 추출하여 클래스를 선정하는 것이다. 즉 불필요한 부분을 생략하고 객체의 속성 중 가장 중요한 것에만 중점을 두어 개략화 · 모델화하는 것이다. 예를 들면 자동차와 말이란 클래스에서 '타는 것'이란 클래스를 만드는 것이다.

① inheritance ② abstraction
③ polymorphism ④ encapsulation

10 객체지향 기법에 관한 설명에서 빈칸에 공통으로 들어갈 말은?

( )은(는) 클래스 내의 객체에 의한 함수이거나 변형이다. 한 클래스 내의 모든 객체는 같은 ( )을(를) 공유하고, 개개 ( )은(는) 묵시적 아규먼트로서 목적 객체를 가지며 행위를 서술한다. 메소드는 한 클래스에 대한 ( )의 구현이며, 일반적으로 객체지향 설계에서는 동일시하고 함수지향 설계에서는 함수로 대응된다.

① 오퍼레이션 ② 인스턴스
③ 메시지 ④ 정보 은닉

11 다음 클래스 다이어그램에서 적용되지 않은 개념은?

① 일반화 ② 상속
③ 다형성 ④ 합성

12 다음은 어떤 개념에 대한 설명인가?

원Circle의 면적을 구하는 getArea) 함수를 가지고 있는 객체, 직사각형Rectangle의 면적을 구하는 getArea( ) 함수를 가지고 있는 객체, 삼각형Triangle의 면적을 구하는 getArea( ) 함수를 가지고 있는 객체는 getArea( )라는 함수를 공통적으로 가지고 있다. 여기서 getArea( ) 함수는 각 객체의 면적을 구하는 함수인데, 실제로 면적을 계산할 때에는 객체의 모양에 따라 계산 방법이 모두 다르다.

① 캡슐화 ② 상속성
③ 다형성 ④ 추상화

13 UML은 시스템의 정적인 부분과 동적인 부분을 표현하기 위해 여러 다이어그램을 제공한다. 정적인 부분을 표현하는 다이어그램은 시스템의 구조를 나타내는 데 사용되고, 동적인 부분을 표현하는 다이어그램은 시스템의 행위를 나타내는 데 사용된다. 다음 중 성격이 다른 다이어그램은?

① 클래스 다이어그램 ② 협동 다이어그램
③ 상태 다이어그램 ④ 활동 다이어그램

14 UML 다이어그램에 대한 설명 중 옳지 않은 것은?

① 사용 사례 다이어그램: 시스템의 기능을 모델링하는 데 사용된다.
② 상태 다이어그램: 객체 사이의 메시지 교환을 시간의 흐름에 따라 표현한다.
③ 클래스 다이어그램: 시스템의 정적인 구조를 나타낸다.
④ 액티비티 다이어그램: 시스템의 동적 특징을 나타낸다.

15 다음 중 UML 다이어그램이 아닌 것은?

① 클래스 다이어그램 ② 속성 다이어그램
③ 사용 사례 다이어그램 ④ 순차 다이어그램