

## 객체지향(object oriented)이란 무엇일까?

⇒ 프로그래밍 방법론

첫 번째 오해

클래스가 객체지향의 핵심이다.

두 번째 오해

객체지향이란 현실 세계의 모방이다.

사실

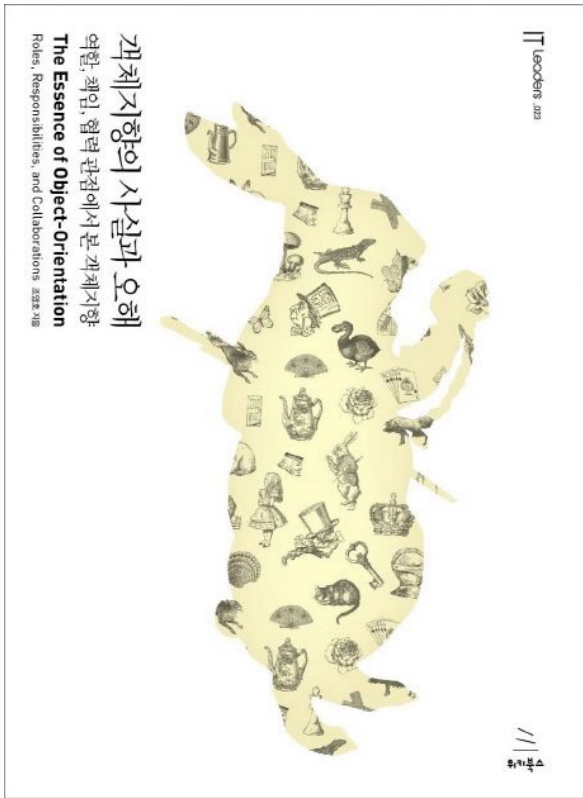
역할

타임

이러한

تشاتو

메시지

채임  
70

<https://youtu.be/dJ5C4gRgAgA>

첫 번째 오해 : 클래스가 객체지향의 핵심이다.

사람들은

역할 협력에 참여하는 사람이 협력 안에서 차지하는 책임이나 임무를 의미



협력



책임을 수행할 수 있는 적절한 역할을 가진 사람에 의해 수행

목표는 더 작은 책임으로 분할

목표는 사람들의 협력을 통해 달성

스스로 해결하지 못하는 문제를 해결하기 위해 도움을 요청한다.

협력의 핵심은 특정한 책임을 수행하는 역할들 간의 연쇄적인 요청과 응답을 통해 목표를 달성한다는 것

객체 공동체 안에 사는 객체시민은

자신에게 주어진 역할과 책임을 다하는 동시에 시스템의 더 큰 목적을 이루기 위해 다른 객체와도 적극적으로 협력한다

첫 번째 오해 : 클래스가 객체지향의 핵심이다.

객체지향 애플리케이션의 윤곽을 결정하는 것은 역할, 책임, 협력이지만 실제로 **협력에 참여하는 주체는 객체다.**

객체는 **충분히 협력적** 이어야 한다. (~~명령에 복종~~ **요청에 응답**)

객체가 **충분히 자율적**이어야 한다. (협력에 참여하는 방법을 스스로 결정)

→ **상태(state)**와 **행동(behavior)**을 함께 지닌 실체

→ 다른 객체가 **무엇을 수행하는지**는 알 수 있지만 **어떻게 수행하는지**는 알 수 없다 ?

자율적인 객체로 구성된 공동체는 **유지보수가 쉽고 재사용이 용이**

객체지향 세계의 의사소통 수단 (message)



객체는 **협력을 위해** 다른 객체에게 **메시지를 전송 및 수신함**

전달하라

**메서드는** 수신된 **메시지**를 처리하는 방법



실행 시간에 메서드를 선택

메시지와 메서드의 분리는 객체의 협력에 참여하는 객체들 간의 자율성을 증진

→ 객체의 **자율성**을 높이는 핵심 메커니즘

→ **캡슐화** 개념과 연관.

첫 번째 오해 : 클래스가 객체지향의 핵심이다.

객체지향 본질

- 객체를 이용해 시스템을 분할
- 자율적인 객체는 상태와 행위를 지니며 자신을 책임짐
- 시스템의 행위를 구현하기 위해 다른 객체와 협력
- 협력을 위해 메시지를 전송하고 메서드를 자율적으로 선택

클래스는 OOP의 매우 중요한 구성요소는 맞지만 객체지향의 핵심을 이루는 중심 개념이라기에는 무리가 있음



중요한 것은 어떤 객체들이 어떤 메시지를 주고받으며 협력하는가다.

두 번째 오해 : 객체지향이란 현실 세계의 모방이다.

객체지향 패러다임의 목적은 현실 세계를 기반으로 새로운 세계를 창조하는 것이다.

객체는 구별 가능한 식별자, 특징적인 행동, 변경 가능한 상태를 가진다

## 상태

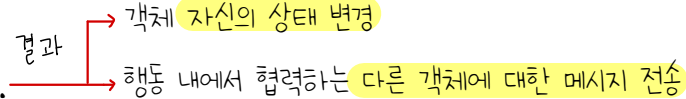
과거의 모든 행동 이력을 설명하지 않고도 행동의 결과를 쉽게 예측하고 설명할 수 있다.

객체의 상태는 단순한 값과 다른 객체를 참조하는 링크로 구분된다.

객체는 스스로의 행동에 의해서만 상태가 변경되는 것을 보장 → 객체의 자율성을 유지

## 행동

행동의 결과는 상태에 의존적이다.

객체가 다른 객체와 협력하는 유일한 방법은 다른 객체에게 요청을 보내는 것이다. 

메시지 송신자는 메시지 수신자의 상태 변경에 대해서는 전혀 알지 못한다. → 캡슐화

## 식별자

객체를 서로 구별할 수 있는 특정한 프로퍼티

훌륭한 객체를 만들기 위한 가장 중요한 덕목은 상태가 아닌 행동에 초점을 맞추는 것이다.

객체지향 설계는 필요한 협력을 생각하고 필요한 행동을 생각하고 행동을 수행할 객체를 선택

두 번째 오해 : 객체지향이란 현실 세계의 모방이다.

## 의인화

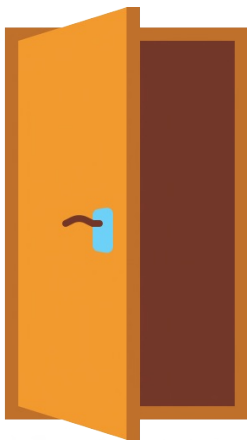
현실 속에서는 수동적인 존재가 객체로 구현될 때는 능동적으로 변한다.

객체지향의 세계와 현실 세계 사이에는 모방이나 추상화 수준이 아닌 다른 관점에서 유사성을 가진다. → 은유

은유 관계에 있는 실제 객체의 이름을 객체의 이름으로 사용하면 표현적 차이를 줄여 구조를 쉽게 예측할 수 있다.

현실 세계

사람이 문을 연다.



객체지향 세계

'문을 연다'라는 메시지를 받은 문이 스스로 열린다

## 추상화

컴퓨터 프로그램을 작성하기 위한 중요한 전제 조건은 추상화를 정확하게 다루는 능력이다.

### 추상화를 통한 복잡성 극복

훌륭한 추상화는 목적에 부합하는 것이어야 한다. → 추상화의 수준, 이익, 가치는 목적에 의존적이다.

추상화는 어떤 양상, 세부사항, 구조를 좀 더 명확하게 이해하기 위해 특정 절차나 물체를 의도적으로 생략하거나 감춤으로써 복잡도를 극복하는 방법

- 구체적인 사물들 간에 공통점은 취하고 차이점은 버리는 일반화를 통해 단순하게 만드는 것
- 중요한 부분을 강조하기 위해 불필요한 세부사항을 제거함으로써 단순하게 만드는 것



## 개념

개념은 공통점을 기반으로 객체를 분류할 수 있다.

Symbol → 개념을 가리키는 간략한 이름이나 명칭 ⇒ (트럼프 사람)

Intension → 개념의 완전한 정의 ⇒ (몸이 납작하고 두 손과 발은 네모 귀퉁이에 달려 있는 등장 인물)

Extension → 개념에 속하는 모든 객체의 집합 ⇒ (정원사, 병사, 왕, 여왕)

분류란 특정한 객체를 특정한 개념의 집합에 포함시키거나 포함시키지 않는 작업을 의미

객체를 적절한 개념에 따라 분류한 애플리케이션은 유지보수가 용이하고 변경에 유연하게 대처할 수 있다.

개념은 객체들의 복잡성을 극복하기 위한 추상화 도구다.

# 타입

타입은 공통점을 기반으로 객체들을 묶기 위한 틀이다. → 타입은 개념의 정의와 완전히 동일하다.

객체를 타입에 따라 분류하고 그 타입에 이름을 붙이는 것은 프로그램에서 사용할 새로운 데이터 타입을 선언하는 것과 같다.

객체는 행위에 따라 변할 수 있는 상태를 가진다.

객체가 이웃하는 객체와 협력하기 위해 어떤 행동을 해야 할지를 결정하는 것이 가장 중요하다.

- 어떤 객체들이 동일한 행동을 수행할 수 있다면 내부 표현 방식이 다르더라도 그 객체들은 동일한 타입으로 분류될 수 있다.
- 객체의 내부적인 표현은 외부로부터 철저하게 감춰진다.

객체의 타입을 결정하는 것은 객체의 행동뿐이다.

같은 타입에 속한 객체는 행동만 동일하다면 서로 다른 데이터를 가질 수 있다.

→ 동일한 책임 → 동일한 메시지 수신

→ 내부의 표현 방식이 달라서 동일한 메시지를 처리하는 방식은 서로 다를 수 밖에 없다. → 다형성

내부 표현 방식과 무관하게 행동안이 고려대상이라는 사실은 외부에 데이터를 감춰야 한다는 것을 의미한다. → 캡슐화

⇒ 객체가 외부에 제공해야 하는 책임을 먼저 결정하고 그 책임을 수행하는 데 적합한 데이터를 나중에 결정한 후, 데이터를 책임을 수행하는 데 필요한 외부 인터페이스 뒤로 캡슐화해야 함.

⇒ 객체를 결정하는 것은 행동이다. 데이터는 단지 행동을 따를 뿐이다.

동일한 행동  
서로 다른 데이터

```
class Calculator:
    def add(a,b):
        return a+b

class WrongCalculator:
    def add(a,b):
        return a*b
```

덧셈함수

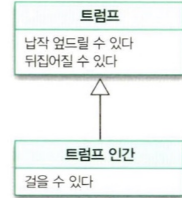
## 타입의 계층

### 일반화/특수화

일반적인 타입 (슈퍼타입)    특수한 타입이 가진 모든 행동들 중에서 일부 행동만을 가지는 타입

특수한 타입 (서브타입)    일반적인 타입이 가진 모든 행동과 자신만의 행동을 추가하는 타입

슈퍼타입의 행동은 서브타입에게 자동으로 상속된다.



## 타입의 목적

타입은 시간에 따라 동적으로 변하는 객체의 상태를 시간과 무관한 정적인 모습으로 다룰 수 있게 해준다.

⇒ 타입은 추상화다.

## 클래스

클래스와 타입은 동일한 것이 아니다.

클래스는 타입을 구현할 수 있는 여러 구현 매커니즘 중 하나일 뿐이다.

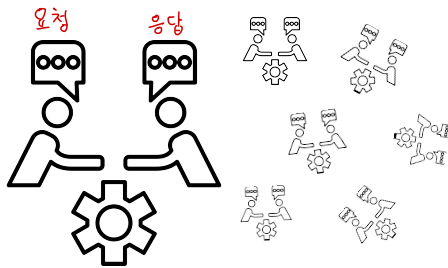
객체지향에서 중요한 것은 동적으로 변하는 객체의 '상태'와 상태를 변경하는 '행위'다.

## 협력

협력은 한 사람이 다른 사람에게 도움을 요청할 때 시작된다.

협력은 다수의 연쇄적인 요청과 응답의 흐름으로 구성된다.

협력에 참여하는 개체가 요청과 응답을 수행할 책임을 정의한다.



## 협력을 따라 흐르는 개체의 책임

개체가 존재하는 이유는 행위를 수행하며 협력에 참여하기 위해서다.

개체지향의 핵심은 클래스를 어떻게 구현할 것인지가 아니라 개체가 협력 안에서 어떤 역할을 수행할 것인지를 결정하는 것이다.

협력을 구성하는 데 필요한 책임을 먼저 고안하고 나면 그 책임을 수행하는 데 필요한 개체를 선택하게 된다.

→ 협력이라는 견고한 문맥이 갖춰지면 협력을 위해 필요한 책임으로 초점이 옮겨진다.

할당된 책임은 외부에 제공하게 될 행동을 정의한다.

행동은 각 개체가 필요로 하는 데이터를 정의할 수 있다.

이렇게 데이터와 행동이 결정된 후에야 구현에 사용될 클래스를 개발할 수 있다.



개체지향 시스템에서 가장 중요한 것은 충분히 자율적인 동시에 충분히 협력적인 개체를 창조하는 것이다.

이 목표를 달성할 수 있는 가장 쉬운 방법은 개체를 충분히 협력적으로 만든 후에 협력이라는 문맥 안에서 개체를 충분히 자율적으로 만드는 것이다.

## 책임

어떤 대상에 대한 요청은 그 대상이 요청을 처리할 책임이 있음을 암시한다.

### 책임의 분류

무엇을 할 수 있는가? (doing)

- 스스로 하는 것
- 다른 객체의 행동을 시작시키는 것
- 다른 객체의 활동을 제어하고 조절하는 것

객체가 무엇을 알고 있는가? (knowing)

- 개인적인 정보에 관해 아는 것
- 관련된 객체에 관해 아는 것
- 자신이 유도하거나 계산할 수 있는 것에 관해 아는 것

책임은 객체의 외부에 제공해줄 수 있는 정보(knowing)와 외부에 제공해 줄수 있는 서비스(doing)의 목록이다.

책임은 객체의 공용 인터페이스(public interface)를 구성한다.

↘ 캡슐화

### 책임과 메시지

- 두 객체 간의 협력은 메시지를 통해 이뤄진다.
- 책임은 객체가 협력에 참여하기 위해 수행해야 하는 행위를 상위 수준에서 개략적으로 서술한 것이다.
- 책임을 결정한 후 협력을 정제하면서 메시지로 변환할 때는 하나의 책임이 여러 메시지로 분할된다
- 책임과 협력의 구조가 자리를 잡기 전까지는 책임을 구현하는 방법에 대한 고민은 뒤로 미루자.
- 객체지향 설계는 협력에 참여하기 위해 어떤 객체가 어떤 책임을 수행해야 하고 어떤 객체로부터 메시지를 수신할 것인지를 결정하는 것으로 부터 시작된다.

## 역할

어떤 객체가 수행하는 **책임의 집합**은 객체가 협력 안에서 **수행하는 역할**을 암시한다.

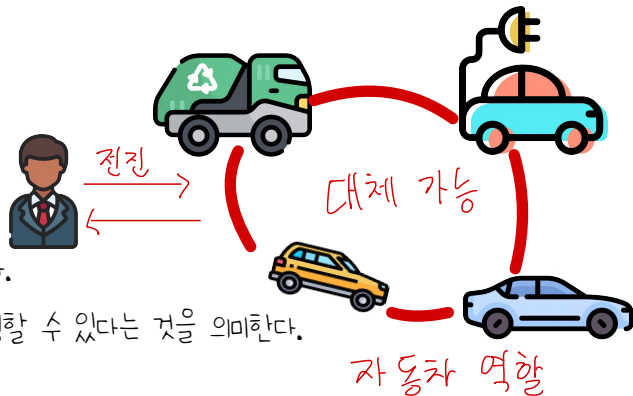
역할은 **협력 내에서 다른 객체로 대체할 수 있음**을 나타내는 일종의 표식이다.

역할을 **대체하기 위해서는** 각 역할이 **수신할 수 있는 메시지를 동일한 방식으로 이해**해야 한다.

동일한 역할을 수행할 수 있다는 것은 해당 객체들이 **협력 내에서 동일한 책임의 집합을 수행할 수 있다는 것**을 의미한다.

→ **동일한 메시지를 수신할 수 있기 때문에 매우 중요한 개념**

역할은 객체지향 설계의 **단순성(Simplicity)**, **유연성(flexibility)**, **재사용성(reusability)**을 뒷받침하는 핵심 개념이다.



## 협력의 추상화

역할은 **하나의 협력 안에 여러 종류의 객체가 참여할 수 있게 함**으로써 **협력을 추상화**할 수 있다.

협력의 추상화는 **설계자가 다뤄야 하는 협력의 개수를 줄이는 동시에 구체적인 객체를 추상적인 역할로 대체함**으로써 **협력의 양상을 단순화**한다.

## 대체 가능성

역할은 협력 안에서 **구체적인 객체로 대체될 수 있는 추상적인 협력자**다. 따라서 **역할은 다른 객체에 의해 대체 가능함**을 의미한다.

대체 가능하기 위해서는 **협력 안에서 역할이 수행하는 모든 책임을 동일하게 수행할 수 있어야** 한다.

역할이 협력을 추상적으로 만들 수 있는 이유는 **역할 자체가 객체의 추상화**이기 때문이다.

⇒ 역할의 **대체 가능성**은 **행위 호환성**을 의미하고, **행위 호환성**은 **동일한 책임의 수행**을 의미한다.

## 객체지향 설계 기법

역할, 책임, 협력의 관점에서 애플리케이션을 설계하는 유용한 기법

### 책임-주도 설계

객체의 책임을 중심으로 시스템을 구축하는 설계 방법

- 시스템 책임을 파악한다.
- 시스템 책임을 더 작은 책임으로 분할한다.
- 적절한 객체 또는 역할을 찾아 책임을 할당한다.
- 해당 객체 또는 역할에게 책임을 할당함으로써 두 객체가 협력하게 한다.

### 디자인 패턴

책임-주도 설계의 결과를 표현한다.

디자인 패턴은 반복적으로 발생하는 문제와 그 문제에 대한 해법의 쌍으로 정의된다.

패턴은 해결하려고 하는 문제가 무엇인지를 명확하게 서술하고 패턴을 적용할 수 있는 상황과 적용할 수 없는 상황을 함께 설명한다.

패턴은 반복해서 일어나는 특정한 상황에서 어떤 설계가 왜(why) 더 효과적인지에 대한 이유를 설명한다.

### 테스트-주도 개발

실패하는 테스트를 작성하고, 테스트를 통과하는 가장 간단한 코드를 작성한 후, 리팩토링을 통해 중복을 제거하는 것

객체가 이미 존재한다고 가정하고 객체에게 어떤 메시지를 전송할 것인지에 관해 먼저 생각하라고 충고한다.

책임-주도 설계를 통해 도달해야 하는 목적지를 테스트라는 안전장치를 통해 좀 더 빠르고 견고한 방법으로 도달할 수 있도록 해주는 최상의 설계 프랙티스다.



## 자율적인 책임

객체지향 공동체를 구성하는 기본 단위는 '자율적인 객체'다.

객체들은 애플리케이션의 기능을 구현하기 위해 협력하고, 협력 과정에서 각자 맡은 바 책임을 다하기 위해 자율적으로 판단하고 행동한다.

↳ 스스로 정한 원칙에 따라 판단하고 스스로의 의지를 기반으로 행동하는 객체다.

객체지향 설계의 아름다움은 적절한 책임을 적절한 객체에게 할당하는 과정 속에서 드러난다.

객체가 자율적이기 위해서는 객체에게 할당되는 책임의 수준 역시 자율적이어야 한다.

이동하라  
↳ 걸어서  
↳ 날아서  
↳ 기어서  
⇒ 책임의 수준이 자율적

요청에 반응해 책임을 완수할 수만 있다면 어떤 방법으로 처리했는지에 관해서는 신경쓰지 않는다.

책임은 협력에 참여하는 의도를 명확하게 설명할 수 있는 수준 안에서 추상적이어야 한다.

하라  
↳ ?  
↳ ?  
↳ ?  
⇒ 책임이 너무 추상적임

어떤 책임이 자율적인지를 판단하는 기준은 문맥에 따라 다르다

어떤 책임이 가장 적절한가는 설계 중인 협력이 무엇인가에 따라 달라진다.

자율적인 책임의 특징은 객체가 '어떻게(how)' 해야 하는가가 아니라 '무엇(what)'을 해야 하는가를 설명한다는 것이다.

책임이라는 말 속에는 어떤 행동을 수행한다는 의미가 포함돼 있다.

메시지는 객체로 하여금 자신의 책임, 즉 행동을 수행하게 만드는 유일한 방법이다.

# 메시지와 메서드

## 메시지

하나의 객체는 메시지를 전송함으로써 다른 객체에 접근한다.

메시지-전송 매커니즘은 객체가 다른 객체에 접근할 수 있는 유일한 방법이다.

메시지는 메시지 이름과 인자로 구성된다.

메시지 전송은 수신자, 메시지(메시지 이름, 인자)의 조합이다. (car.move(10))

송신자는 메시지 전송을 통해서만 다른 객체의 책임을 요청할 수 있고, 수신자는 오직 메시지 수신을 통해서만 자신의 책임을 수행할 수 있다.

객체가 수신할 수 있는 메시지의 모양이 객체가 수행할 책임의 모양을 결정한다.

수신자가 메시지를 변경하지 않는다면 책임을 수행하는 방법을 변경하더라도 송신자는 그 사실을 알 수 없다. (캡슐화)

메시지가 수신자의 책임을 결정한다.

수신 가능한 메시지가 모여 객체의 인터페이스를 구성한다.

## 메서드

메시지를 처리하기 위해 내부적으로 선택하는 방법을 메서드라고 한다.

메시지는 '어떻게' 수행될 것인지는 명시하지 않는다. 메시지는 오퍼레이션을 통해 무엇이 실행되기를 바라는지만 명시하며, 어떤 메서드를 선택할 것인지는 전적으로 수신자의 결정에 좌우된다.

## 다형성

다형성을 하나의 메시지와 하나 이상의 메서드 사이의 관계로 볼 수 있다.

서로 다른 객체들이 다형성을 만족시킨다는 것은 객체들이 동일한 책임을 공유한다는 것을 의미한다. 다형성에서 중요한 것은 송신자의 관점이다.

다형성은 동일한 역할(role)을 수행할 수 있는 객체들 사이의 대체 가능성을 의미한다.

다형성을 사용하면 송신자가 수신자의 종류를 모르더라도 메시지를 전송할 수 있다. (수신자의 종류를 캡슐화)

다형성은 송신자와 수신자 간의 객체 타입에 대한 결합도를 메시지에 대한 결합도로 낮춤으로써 달성된다.

다형성을 이용해 협력을 유연하게 만들 수 있다.

## 메시지와 메서드

### 유연하고 확장 가능하고 재사용성이 높은 협력의 의미

송신자가 수신자에 대해 매우 적은 정보만 알고 있더라도 상호 협력이 가능하다는 사실은 설계의 품질에 큰 영향을 미친다.

1. 협력이 유연해진다.
2. 협력이 수행되는 방식을 확장할 수 있다.
3. 협력이 수행되는 방식을 재사용할 수 있다.

### 송신자와 수신자를 약하게 연결하는 메시지

메시지는 송신자와 수신자 사이의 결합도를 낮춤으로써 설계를 유연하고, 확장 가능하고, 재사용 가능하게 만든다.

송신자는 오직 메시지만 바라본다. 단지 수신자가 메시지를 이해하고 처리해 줄 것이라는 사실만 알아도 충분하다. 수신자는 메서드 자체는 송신자에게 노출시키지 않는다.

설계의 품질을 높이기 위해서는 훌륭한 메시지를 선택해야 한다.

# 객체 인터페이스

인터페이스란 어떤 두 사물이 마주치는 경계 지점에서 서로 상호작용할 수 있게 이어주는 방법이나 장치를 의미한다.

- 인터페이스의 사용법을 익히지만 하면 내부 구조나 동작 방식을 몰라도 쉽게 대상을 조작하거나 의사를 전달할 수 있다.
- 인터페이스 자체는 변경하지 않고 단순히 내부 구성이나 작동 방식을 변경하는 것은 인터페이스 사용자에게 어떤 영향도 미치지 않는다.
- 대상이 변경되더라도 동일한 인터페이스를 제공하기만 하면 아무런 문제 없이 상호작용 할 수 있다.

인터페이스는 객체가 수신할 수 있는 메시지의 목록으로 구성된다.

## 인터페이스와 구현의 분리

객체지향적인 사고 방식을 이해하기 위해서는 다음 세 가지 원칙이 중요하다고 말한다.

- 좀 더 추상적인 인터페이스 → 추상적인 수준의 메시지를 수신할 수 있는 인터페이스를 제공하면 수신자의 자율성을 보장할 수 있다.
- 최소 인터페이스 → 외부에서 사용할 필요가 없는 인터페이스는 최대한 노출하지 말라는 것
- 인터페이스와 구현 간에 차이가 있다는 점을 인식
  - 내부 구조와 작동 방식을 가리키는 고유의 용어
  - 객체를 구성하지만 공용 인터페이스에 포함되지 않는 모든 것이 구현에 포함된다.
  - 객체의 외부와 내부를 분리하는 것은 객체의 공용 인터페이스와 구현을 명확하게 분리하라는 말과 같다.

## 인터페이스와 구현의 분리 원칙

객체를 설계할 때 객체 외부에 노출되는 인터페이스와 객체의 내부에 숨겨지는 구현을 명확하게 분리해서 고려해야 한다는 것을 의미한다.

## 책임의 자율성이 협력의 품질을 결정

책임이 자율적일수록 적절하게 '추상화'되며, '응집도'가 높아지고, '결합도'가 낮아지며, '캡슐화'가 증진되고, 인터페이스와 구현이 명확히 분리되며, 설계의 '유연성'과 '재사용성'이 향상된다.

## 기능 설계 대 구조 설계

깔끔하고 단순하며 유지보수하기 쉬운 설계는 사용자의 요구사항을 반영할 수 있도록 쉽게 확장 가능한 소프트웨어를 창조할 수 있는 기반이 된다.

소프트웨어 분야에서 예외가 없는 유일한 규칙은 요구사항이 항상 변경된다는 것이다. → 설계가 중요한 이유는 변경에 의한 필요성 때문이다.

미래에 대비하는 가장 좋은 방법은 변경을 예측하는 것이 아니라 변경을 수용할 수 있는 선택의 여지를 설계에 마련해 놓는 것이다.

객체지향 접근방법은 자주 변경되지 않는 안정적인 객체 구조를 바탕으로 시스템 기능을 객체 간의 책임으로 분배한다.

객체지향은 객체의 구조에 집중하고 기능이 객체의 구조를 다르게 만든다.

시스템 기능은 더 작은 책임으로 분할되고 적절한 객체에게 분배되기 때문에 기능이 변경되더라도 객체 간의 구조는 그대로 유지된다.

## 안정적인 재료: 구조

사용자가 프로그램을 사용하는 대상 분야

→ 대상을 단순화해서 표현한 것.

→ 필요한 지식만 선택적으로 단순화하고 의식적으로 구조화한 형태.

도메인 모델

→ 소프트웨어가 목적하는 영역 내의 개념과 개념 간의 관계, 다양한 규칙이나 제약 등을 주의 깊게 추상화한 것

→ 이해관계자들이 바라보는 멘탈 모델

→ 사람들이 동일한 용어와 동일한 개념을 이용해 의사소통하고 코드로부터 도메인 모델을 유추할 수 있게 하는 것이 도메인 모델의 목표

사용자의 멘탈 모델

이 도메인 모델은 이렇게 동작하겠구나 !



도메인 모델

제품을 설계할 때 제품에 관한 모든 것이 사용자들이 제품에 대해 가지고 있는 멘탈 모델과 정확하게 일치해야 한다.

은유를 통해 사용자가 도메인에 대해 생각하는 개념들을 투영해야 한다.

사용자들이 도메인의 본질적인 측면을 잘 이해하고 있기 때문에 사용자의 관점을 반영해야 한다.

도메인 모델의 장점은 비즈니스의 개념과 정책을 반영하는 안정적인 구조를 제공한다는 것이다.

## 불안정한 재표: 기능

기능적 요구사항이란 시스템이 사용자에게 제공해야 하는 기능의 목록을 정리한 것이다.

불충한 기능적 요구사항을 얻기 위해서는 목표를 가진 사용자와 사용자의 목표를 만족시키기 위해 일련의 절차를 수행하는 시스템 간의 상호작용 관점에서 시스템을 바라보아야 한다.

사용자의 목표를 달성하기 위해 사용자와 시스템 간에 이뤄지는 상호작용의 흐름을 텍스트로 정리한 것을 유스케이스라고 한다.

→ 시스템에 작업을 요청하면 시스템은 요청을 처리하고 사용자에게 원하는 결과를 제공

산발적으로 흩어져 있는 기능에 사용자 목표라는 문맥을 제공함으로써 각 기능이 유기적인 관계를 지닌 체계를 이룰 수 있게 한다.

유스케이스는 설계 기법도, 객체지향 기법도 아니다.

→ 유스케이스에는 단지 사용자가 시스템을 통해 무엇을 얻을 수 있고 어떻게 상호작용할 수 있는냐에 관한 정보만 기술된다.

→ 유스케이스는 단지 기능적 요구사항을 사용자의 목표라는 문맥을 중심으로 묶기 위한 정리 기법일 뿐이다.

→ 유스케이스 안에는 영감을 불러일으킬 수 있는 약간의 힌트만이 들어 있을 뿐이다.

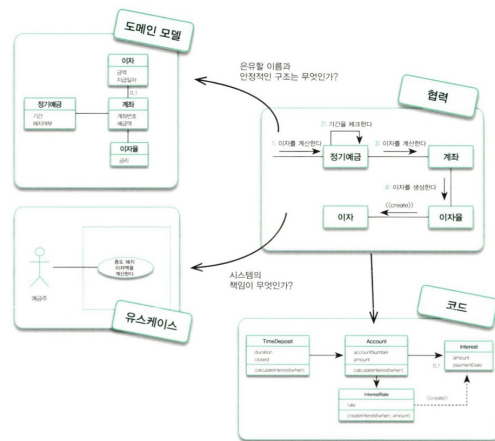


그림 6.8 도메인 모델은 구조를, 유스케이스는 협력의 출발점인 시스템 책임을 제공한다.

## 재표 합치기: 기능과 구조의 통합

도메인 모델이 안정적인 이유

→ 도메인 모델을 구성하는 개념은 비즈니스가 없어지거나 완전히 개편되지 않는 한 안정적으로 유지된다.

→ 도메인 모델을 구성하는 개념 간의 관계는 비즈니스 규칙을 기반으로 하기 때문에 비즈니스 정책이 크게 변경되지 않는 한 안정적으로 유지된다.

객체지향의 가장 큰 장점은 도메인을 모델링하기 위한 기법과 도메인을 프로그래밍하기 위해 사용하는 기법이 동일하다는 점이다. → 연결완전성

객체지향이 강력한 이유는 연결완전성의 역방향 역시 성립한다는 것이다. (코드의 변경으로 도메인 모델의 변경 사항을 유추할 수 있음) → 가역성(reversibility)

안정적인 도메인 모델을 기반으로 시스템의 기능을 구현 → 유지보수하기 쉽고 유연한 객체지향 시스템을 만드는 첫 걸음

예시

## 커피 전문점

손님은 어떤 식으로든 메뉴판을 알아야 한다.

손님과 바리스타 사이에도 관계가 존재한다.

바리스타는 자신이 만든 커피와 관계를 맺는다.

⇒ 동적인 객체를 정적인 타입으로 추상화해서 복잡성을 낮춰야 한다. 타입은 분류를 위해 사용된다. 상태와 무관하게 동일하게 행동할 수 있는 객체들은 동일한 타입의 인스턴스로 분류할 수 있다.

숙이 찬 마음모는 포함(containment) 관계 또는 합성(composition) 관계



그림 7.3 메뉴판 타입과 메뉴 항목 타입 간의 포함 관계

단순히 선으로 연결하는 경우 연관(association) 관계라고 한다. (한 타입의 인스턴스가 다른 타입의 인스턴스를 포함하지는 않지만 서로 알고 있어야 할 경우)

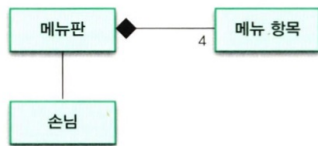


그림 7.4 손님과 메뉴판 사이의 연관 관계

소프트웨어가 대상으로 하는 영역인 도메인을 단순화해서 표현한 모델을 도메인 모델이라고 한다.

적절한 객체에게 적절한 책임을 할당해야 한다. → 협력을 설계하는 것

어떤 타입이 도메인을 구성하느냐와 타입들 사이에 어떤 관계가 존재하는지를 파악함으로써 도메인을 이해한다.

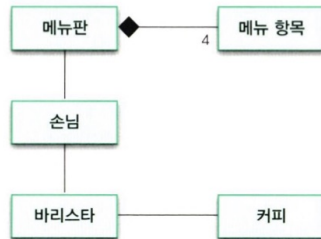


그림 7.5 커피 전문점을 구성하는 타입들

## 설계하고 구현하기

### 협력 찾기

메시지가 객체를 선택하게 해야 한다. ⇒ 메시지를 먼저 선택, 메시지를 수신하기에 적절한 객체를 선택.

메시지는 객체가 외부에 제공하는 공용 인터페이스에 포함됨

메시지를 처리할 책임을 맡음



그림 7.6 협력을 시작하게 하는 첫 번째 메시지

메시지를 처리할 객체를 찾는 방법은 먼저 도메인 모델 안에 책임을 수행하기에 적절한 타입이 존재하는지 살펴보는 것.

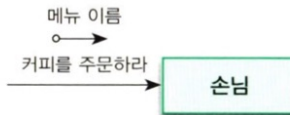


그림 7.7 첫 번째 메시지가 손님이라는 객체를 선택했다.

손님이 할당된 책임을 수행하는 도중에 스스로 할 수 없는 일이 있다면, 다른 객체에게 이를 요청해야 한다.

손님 객체에서 외부로 전송되는 메시지 정의



그림 7.8 스스로 할 수 없는 일은 메시지를 전송해 다른 객체에게 도움을 요청한다.

이제 메시지를 정제함으로써 각 객체의 인터페이스를 구현 가능할 정도로 상세하게 정제한다.

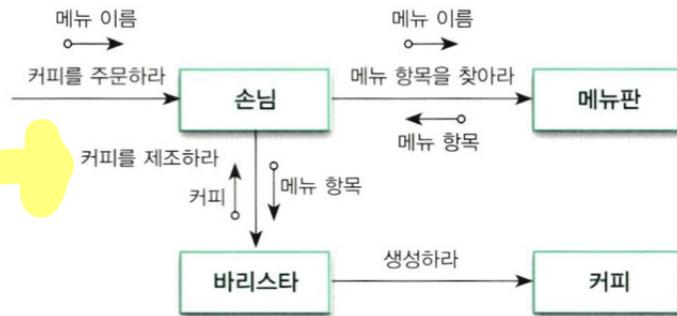


그림 7.12 커피 주문을 위한 객체 협력

의사소통이라는 목적에 부합한다면 용도에 맞게 얼마든지 UML을 수정하고 뒤틀어라



## 인터페이스 정리하기

객체가 수신한 메시지가 객체의 인터페이스를 결정한다. → 메시지가 객체를 선택했고, 선택된 객체는 메시지를 자신의 인터페이스로 받아들인다.

각 객체를 협력이라는 문맥에서 떼어내어 수신 가능한 메시지만 주려내면 객체의 인터페이스가 된다.

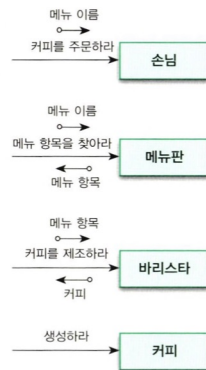


그림 7.13 각 객체들이 수신하는 메시지는 객체의 인터페이스를 구성한다.

소프트웨어의 구현은 동적인 객체가 아닌 정적인 타입을 이용해 이뤄진다. 따라서 객체들을 포괄하는 타입을 정의한 후 식별된 오퍼레이션은 타입의 인터페이스에 추가해야 한다.

협력을 통해 식별된 타입의 오퍼레이션은 외부에서 접근 가능하도록 공용(public)으로 선언되어야 한다.

```
class MenuItem {
}

class Menu {
    public MenuItem choose(String name) {}
}

class Barista {
    public Coffee makeCoffee(MenuItem menuItem) {}
}

class Coffee {
    public Coffee(MenuItem menuItem) {}
}
```

## 구현하기

객체가 다른 객체에게 메시지를 전송하기 위해 자신과 협력하는 객체에 대한 참조를 얻어야 한다.

```
class Customer {
    public void order(String menuName, Menu menu, Barista barista) {}
}
```

남은 것은 order() 메서드의 구현을 채우는 것뿐이다.

```
class Customer {
    public void order(String menuName, Menu menu, Barista barista) {
        MenuItem menuItem = menu.choose(menuName);
        Coffee coffee = barista.makeCoffee(menuItem);
        ...
    }
}
```

구현 도중에 객체의 인터페이스가 변경될 수 있다,

→ 설계 작업은 구현을 위한 스케치를 작성하는 단계이다. 중요한 것은 설계가 아니라 코드다. 최대한 빨리 코드를 구현해서 설계에 이상이 없는지, 설계가 구현 가능한지를 판단해야 한다. 코드를 통한 피드백 없이는 깔끔한 설계를 얻을 수 없다.

객체의 속성이 캡슐화된다

- 객체에게 책임을 할당하고 인터페이스를 결정할 때는 가급적 객체 내부의 구현에 대한 어떤 가정도 하지 말아야 한다.
- 객체가 어떤 책임을 수행해야 하는지를 결정한 후에야 책임을 수행하는데 필요한 객체의 속성을 결정하라.
- 객체의 구현 세부 사항을 공용 인터페이스에 노출시키지 않고 인터페이스와 구현을 깔끔하게 분리할 수 있는 방법이다.

```
class Menu {
    private List<MenuItem> items;

    public Menu(List<MenuItem> items) {
        this.items = items;
    }

    public MenuItem choose(String name) {
        for(MenuItem each : items) {
            if (each.getName().equals(name)) {
                return each;
            }
        }
        return null;
    }
}
```

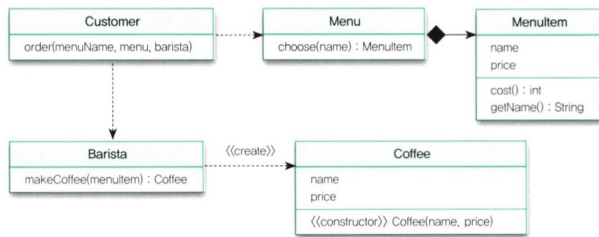


그림 7.15 커피 전문점을 구현한 최종 클래스 구조

인터페이스를 통해 실제로 상호작용하지 않은 채 인터페이스의 모습을 정확하게 예측하는 것은 불가능에 가깝다.

설계를 간단히 끝내고 최대한 빨리 구현에 돌입하라.

실제로 코드를 작성해가면서 협력의 전체적인 밑그림을 그려보라.

## 코드와 세 가지 관점

개념 관점 → 개념 관점에서 클래스가 도메인의 개념의 특성을 최대한 수용하면 변경을 관리하기 쉽고 유지보수성을 향상시킬 수 있다.

소프트웨어 클래스와 도메인 클래스 사이의 간격이 좁으면 좁을수록 기능을 변경하기 위해 뒤적거려야 하는 코드의 양도 점점 줄어든다.

명세 관점 → 명세 관점은 클래스의 인터페이스를 바라본다.

객체의 인터페이스는 수정하기 어렵다.

최대한 변화에 안정적인 인터페이스를 만들기 위해서는 인터페이스를 통해 구현과 관련된 세부 사항이 드러나지 않게 해야 한다.

구현 관점 → 구현 관점은 클래스의 내부 구현을 바라본다.

메서드의 구현과 속성의 변경은 원칙적으로 객체에게 영향을 미쳐서는 안 된다.

↳ 메서드와 속성이 철저하게 클래스 내부로 캡슐화 되어야 한다는 것을 의미

훌륭한 객체지향 프로그래머는 하나의 클래스 안에 세 가지 관점을 모두 포함하면서도 각 관점에서 대응되는 요소를 명확하고 깔끔하게 드러낼 수 있다.

도메인 개념을 참조하는 이유

소프트웨어 클래스가 도메인 개념을 따르면 변화에 쉽게 대응할 수 있다.

인터페이스와 구현을 분리하라

명세 관점은 클래스의 안정적인 측면을 드러내야 한다.

구현 관점은 클래스의 불안정한 측면을 드러내야 한다.

명세 관점이 설계를 주도하면 설계의 품질이 향상될 수 있다.

# 추상화 기법

추상화는 도메인의 복잡성을 단순화하고 직관적인 멘탈 모델을 만드는 데 사용할 수 있는 가장 기본적인 인지 수단이다.

## 분류와 인스턴스화

분류는 객체의 구체적인 세부 사항을 숨기고 인스턴스 간에 공유하는 공통적인 특성을 기반으로 범주를 형성하는 과정.

객체를 분류하고 범주로 묶는 것은 객체들의 특정 집합에 공통의 개념을 적용하는 것을 의미한다.

분류는 객체를 특정한 개념을 나타내는 집합의 구성 요소로 포함시킨다.

객체지향의 세계에서 개념을 가리키는 표준 용어는 **타입**이다.

↳ 개념, 속성과 행위가 유사한 객체에 공통적으로 적용되는 관념이나 아이디어

타입을 객체의 분류 장치로서 적용하려면 아래 세 가지 관점에서의 정의가 필요하다.

심볼 → 타입을 가리키는 간략한 이름이나 명칭

내연 → 타입의 완전한 정의

외연 → 타입에 속하는 모든 객체들의 집합

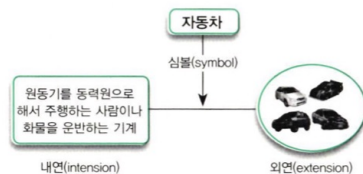
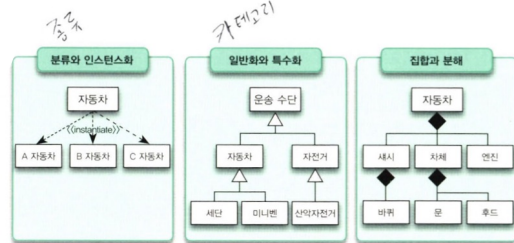


그림 A.3 자동차 타입을 정의하는 세 가지 관점

타입의 외연은 타입에 속하는 객체들의 집합으로 표현된다.

- 단순함을 위해 단일/정적 분류를 선택하는 것이 현명
- 단일 분류(single classification)** → 한 객체가 한 시점에 하나의 타입에만 속하는 것
  - 다중 분류(multiple classification)** → 한 객체가 한 시점에 여러 타입에 속하는 것
  - 동적 분류(dynamic classification)** → 객체가 한 집합에서 다른 집합의 원소로 자신이 속하는 타입을 변경할 수 있는 경우
  - 정적 분류(static classification)** → 객체가 자신의 타입을 변경할 수 없는 경우



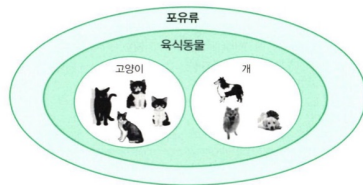
\* 추상화 메커니즘의 종류

## 일반화와 특수화

### 범주 간의 계층적인 구조

좀 더 세부적인 범주가 계층의 하위에 위치, 일반적인 범주가 계층의 상위에 위치

계: 동물계(Animalia)  
문: 척삭동물문(Chordata)  
강: 포유류강(Mallalia)  
목: 육식동물목(Carnivora)  
과: 고양이과(Felidae)  
속: 고양이속(Felis)  
종: 고양이종(Catus)



계층의 상위에 위치한 범주를 계층의 하위에 위치한 범주의 일반화 / 반대를 범주의 특수화

그림 A.7 외연의 관점에서 서브타입은 슈퍼타입의 부분집합이다.

일반적인 타입을 슈퍼타입(supertype)

100% 규칙

서브타입은 속성과 연관관계 면에서 슈퍼타입과 100% 일치해야 한다.

타입의 내연과 관련된 규칙

특수한 타입을 서브타입(subtype)

is-a 규칙

서브타입은 슈퍼타입이다. (Subtype is a supertype)

타입의 외연과 관련된 규칙

일반화와 특수화 관계를 구현하는 가장 일반적인 방법은 클래스 간의 상속을 사용하는 것이다.

일반화의 원칙은 한 타입이 서브타입이 되기 위해서는 슈퍼타입에 순응(conformance)해야 한다는 것이다.

구조적인 순응(structural conformance)

타입의 내연과 관련된 100% 규칙을 의미

서브타입은 슈퍼타입이 가지고 있는 속성과 연관관계 면에서 100% 일치해야 한다.

행위적인 순응(behavioral conformance)

타입의 행위에 관한 것

리스코프 치환 원칙(Liskov Substitution Principle, LSP)라고 한다

여러 클래스로 구성된 상속 계층에서 수신된 메시지를 이해하는 기본적인 방법은 클래스 간의 위임(delegation)을 사용하는 것이다.

## 집합과 분해

복잡성은 '계층'의 형태를 따른다.

단순한 형태로부터 복잡한 형태로 진화하는 데 걸리는 시간은 그 사이에 존재하는 안정적인 형태의 수와 분포에 의존한다.

안정적인 형태의 부분으로부터 전체를 구축하는 행위를 **집합**

전체를 부분으로 분할하는 행위를 **분해**

집합의 가치는 많은 수의 사물들의 형상을 하나의 단위로 다룸으로써 복잡성을 줄일 수 있다는 데 있다. 집합은 불필요한 세부 사항을 추상화한다.

집합은 전체의 내부로 불필요한 세부 사항을 감춰주기 때문에 추상화 메커니즘인 동시에 캡슐화 메커니즘임.

집합과 분해는 한 번에 다뤄야 하는 요소의 수를 감소시킴으로써 인지 과부하를 방지한다.

### 합성 관계

객체와 객체 사이의 전체-부분 관계를 구현하기 위해서 사용

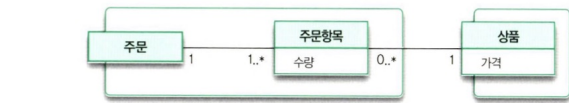


그림 A.9 합성 관계로 연결된 주문과 주문 항목

합성 관계는 부분을 전체 안에 캡슐화함으로써 인지 과부하를 방지한다.



그림 A.10 합성 관계는 주문 항목의 존재를 일시적으로 감춤으로써 복잡성을 낮춘다.

합성 관계로 연결된 객체는 포함하는 객체가 제거될 때 내부에 포함된 객체도 함께 제거된다. ← 합성 관계는 생애주기 측면에서 연관 관계보다 더 강하게 객체들을 결합한다.

연관 관계로 연결된 두 객체는 독립적으로 제거될 수 있다.

### 패키지

관련된 클래스 집합을 하나의 논리적인 단위로 묶는 구성 요소를 패키지(package) 또는 모듈(module)이라고 한다.

패키지를 이용하면 시스템의 전체적인 구조를 이해하기 위해 한 번에 고려해야 하는 요소의 수를 줄일 수 있다.

함께 협력하는 응집도 높은 클래스 집합을 하나의 패키지 내부로 모으면 코드를 이해하기 위해 패키지 경계를 넘나들 필요가 적어진다.

합성 관계가 내부에 포함된 객체들의 존재를 감춤으로써 내부 구조를 추상화하는 것처럼 패키지는 내부에 포함된 클래스들을 감춤으로써 시스템 구조를 추상화한다.