7️⃣

# 함께 모으기

> 💡 코드와 모델을 밀접하게 연관시키는 것은 코드에 의미를 부여하고 모델을 적절하게 한다.
> - 에릭 에반스

**객체지향 설계 안에 존재하는 세 가지 상호 연관된 관점**

- 개념 관점
  - 설계는 도메인 안에 존재하는 개념과 개념들 사이의 관계를 표현함
  - 도메인이란 사용자들이 관심을 가지고 있는 특정 분야나 주제를 말함
  - 소프트웨어는 도메인에 존재하는 문제를 해결하기 위해 개발됨
  - 실제 도매인의 규칙과 제약을 최대한 유사하게 반영하는 것이 핵심
- 명세 관점
  - 사용자의 영역인 도메인을 벗어나 개발자의 영역인 소프트웨어로 초점이 옮겨짐
  - 도메인의 개념이 아니라 실제로 소프트웨어 안에서 살아 숨쉬는 객체들의 책임에 초점을 맞추게 됨
  - 객체의 인터페이스를 바라보게 됨
- 구현 관점
  - 실제 작업을 수행하는 코드와 연관됨
  - 객체들이 책임을 수행하는 데 필요한 동작하는 코드를 작성하는 것에 초점을 맞추게 됨

**클래스와 관점의 관계**

- 클래스는 세 가지 관점이라는 안경을 통해 설계와 관련된 다양한 측면을 드러낼 수 있음
- 클래스가 은유하는 개념 = 도메인 관점
- 클래스의 공용 인터페이스 = 명세 관점
- 클래스의 속성과 메서드 = 구현 관점
- 클래스는 세 가지 관점을 모두 수용할 수 있도록 개념, 인터페이스, 구현을 함께 드러내야 함
- 코드 안에서 세 가지 관점을 쉽게 식별할 수 있도록 분리해야 함

```
7장의 목표
1. 도메인 모델에서 시작해서 최종 코드까지의 구현 과정
2. 구현 클래스를 개념 관점, 명세 관점, 구현 관점에서 바라본다는 것이 무엇을 의
```

# 도메인

**커피 전문점이라는 세상**

- 도메인
  - 커피 전문점
- 객체
  - 손님
  - 메뉴
  - 메뉴판
  - 바리스타
  - 커피
- 인스턴스
  - 동적인 객체를 정적인 타입으로 추상화해서 복잡성을 낮춤

    - 손님 타입: 손님
    - 커피 타입: 아메리카노, 에스프레스, 카라멜 마키아또
- 객체들 간의 관계
    - 포함/합성 관계
        - 특정 항목이 다른 한쪽의 항목에 포함된다는 뜻
    - 연관 관계
        - 선
        - 한 타입의 인스턴스가 다른 타입의 인스턴스를 포함하지 않지만 서로 알고 있어야 함

**도메인 모델**

- 소프트웨어가 대상으로 하는 영역인 도메인을 단순화해서 표현한 모델

# 설계하고 구현하기

**객체지향의 목표**

- 훌륭한 객체를 설계하는 것이 아니라 `훌륭한 협력을 설계하는 것`

**협력 설계**

- 협력을 설계할 때는 객체가 메시지를 선택하는 것이 아니라 메시지가 객체를 선택하게 해야 함
- 메시지를 먼저 선택하고 그 후에 메시지를 수신하기에 적절한 객체를 선택해야 함
- 메시지를 수신할 객체는 메시지를 처리할 책임을 맡게 되고 객체가 수신하는 메시지는 객체가 외부에 제공하는 공용 인터페이스에 포함됨

**커피 전문점 설계**

- '커피를 주문하라'라는 메시지를 수신할 객체는 무엇인가?

    - 어떤 객체가 커피를 주문할 책임을 저야 하는가? = 손님
    - 메시지를 처리할 객체 = 손님 타입의 인스턴스
    - 손님 객체는 커피를 주문할 책임을 할당받음
- 손님 객체는 메뉴 항목에 대해서 알지 못하기 때문에 메뉴 항목에 대한 요청을 외부로 전송해야 됨
    - '메뉴 이름'이라는 인자를 포함해 메시지를 전송함
    - 메시지를 수신한 객체는 '메뉴 이름'에 대응되는 '메뉴 항목'을 반환해야 함
    - 메뉴 항목을 찾을 책임은 메뉴 항목을 가장 잘 알고 있는 메뉴판 객체에게 할당함
- 손님 객체는 메뉴 항목을 얻어서 커피 제조에 대한 요청을 외부로 전송해야 됨
    - '메뉴 항목'을 인자로 전달하고 반환값으로 제조된 커피를 받아야 함
    - 커피 제조에 대한 책임은 바리스타 객체에게 할당함
- 바리스타 객체가 본인이 알고있는 정보로 커피를 제조하여 손님 객체에게 반환하면서 커피 만드는 것이 끝남

⇒ 협력에 필요한 객체의 종류와 책임, 주고받아야 하는 메시지에 대한 대략적인 윤곽이 잡혔음

⇒ 메시지를 정제함으로써 각 객체의 인터페이스를 구현 가능할 정도로 상세하게 정제하는 작업이 필요함

**인터페이스 정리하기**

- 각 객체를 협력이라는 문맥에서 떼어내어 수신 가능한 메시지만 추려내면 객체의 인터페이스가 됨
- 객체가 어떤 메시지를 수신할 수 있다는 것은 그 객체의 인터페이스 안에 메시지에 해당하는 오퍼레이션이 존재한다는 것을 의미
    - 손님 객체의 인터페이스: '커피를 주문하라'라는 오퍼레이션 포함
    - 메뉴판 객체의 인터페이스: '메뉴 항목을 찾아라'라는 오퍼레이션 포함
- 객체들의 협력은 실행 시간에 컴퓨터 안에 일어나는 상황을 동적으로 묘사한 모델
    - 소프트웨어의 구현은 동적인 객체가 아닌 정적인 타입을 이용해 이뤄짐

    - 객체들을 포괄하는 타입을 정의한 후 식별된 오퍼레이션을 타입의 인터페이스에 추가해야 함
- 객체의 타입을 구현하는 일반적인 방법 = 클래스
    - 인터페이스에 포함된 오퍼레이션은 외부에서 접근 가능하도록 public으로 선언해야 됨

```
class Customer { public void order(String menuName) {} }
class MenuItem {}
class Menu { public MenuItem choose(String name) {} }
class Barista { public Coffe makeCoffe(MenuItem menuItem) {}
class Coffee { public Coffee(MenuItem memuItem) {} }
```

**구현하기**

- 오퍼레이션을 수행하는 방법을 메서드로 구현
- Customer가 어떻게 Menu 객체와 Barista 객체에 접근할 수 있을까?
    - 객체에 대한 참조를 얻어야 함
    - order() 메서드의 인자로 Menu와 Barista 객체를 전달받는 방법으로 참조 문제 해결

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

- 참조를 얻어 각 책임에 맞게 구현하기

# 코드와 세 가지 관점

**코드는 세 가지 관점을 모두 제공해야 한다**

- 개념 관점
  - 소프트웨어 클래스가 도메인 개념의 특성을 최대한 수용하면 변경을 관리하기 쉽고 유지보수성을 향상시킬 수 있음
  - 소프트웨어 클래스와 도메인 클래스 사이의 간격이 좁으면 좁을수록 기능을 변경하기 위해 뒤적거려야 하는 코드의 양도 점점 줄어듦
- 명세 관점
  - 클래스의 인터페이스를 바라봄
  - 공용 인터페이스는 외부의 객체가 해당 객체의 접근할 수 있는 유일한 부분
  - 인터페이스를 수정하면 해당 객체와 협력에 하는 모든 객체에 영향을 미침
  - 최대한 변화에 안정적인 인터페이스를 만들기 위해서는 인터페이스를 통해 구현과 관련된 세부 사항이 드라나지 않게 해야 함
- 구현 관점
  - 클래스의 내부 구현을 바라봄
  - 클래스의 메소드와 속성은 구현에 속하며 공용 인터페이스의 일부가 아님
  - 메서드의 구현과 속성의 변경은 원칙적으로 외부의 객체에게 영향을 미쳐서는 안 됨
  - 메서드와 속성이 철저하게 클래스 내부로 캡슐화되어야 함
  - 외부의 클래스는 자신이 협력하는 다른 클래스의 비밀 때문에 우왕좌왕해서는 안됨

**도메인 개념을 참조하는 이유**

- 소프트웨어는 항상 변함
- 때문에 설계는 변경을 위해 존재함
- 여러 개의 클래스로 기능을 분할하고 클래스 안에서 인터페이스와 구현을 분리하는 이유는 변경이 발생했을 때 코드를 좀 더 수월하게 수정하길 원하기 때문임
- 소프트웨어 클래스가 도메인 개념을 따르면 변화에 쉽게 대응할 수 있음

# 인터페이스와 구현을 분리하라!

- 클래스를 봤을 때 클래스를 명세 관점과 구현 관점으로 나눠볼 수 있어야 함
- 캡슐화를 위반해서 구현을 인터페이스 밖으로 노출해서는 안 됨
- 인터페이스와 구현을 명확하게 분리하지 않고 흐릿하게 섞어놓아서도 안 됨
- 세 가지 관점 모두에서 클래스를 바라볼 수 있으려면 훌륭한 설계가 뒷받침돼야 함