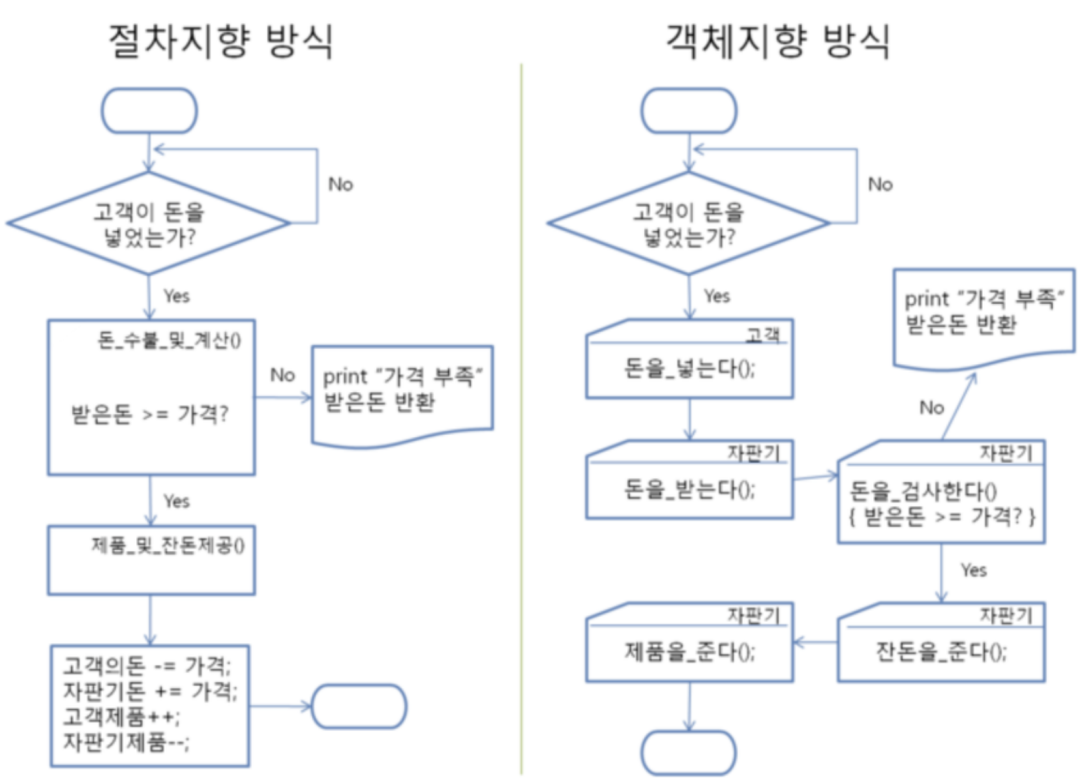
OOP

과목	소프트웨어공학
날짜	@2023년 3월 13일
발표자	윤선영

절차지향 VS 객체지향

	절차지향	객체지향
접근 방식	Top-down	Bottom-up
구현 관점	전체적인 기능 동작 고려 ⇒ 각 단계별로 기능 구현	필요한 속성의 객체를 설계 ⇒ 각 객체의 상호작용을 설계
구성 요소	함수	객체
접근 제어	없음 (모두 Public)	Public, Private, Protected
오버로딩, 다형성	불가능	함수, 생성자, 연산자 등을 오버로딩 가능
상속	불가능	가능
보안성	낮음	높음
데이터 공유	모든 함수가 가능	객체 간 멤버 함수로 가능
예시 언어	C	C++, Python, Java, Javascript



⇒ 절차 지향은 데이터를 중심으로 함수를 구현하고, 객체지향은 기능을 중심으로 메서드를 구현한다.

설계 방식의 차이

절차지향 프로그래밍은 프로그램의 순서와 흐름을 먼저 세우고, 필요한 자료구조와 함수들을 설계하는 방식이다.

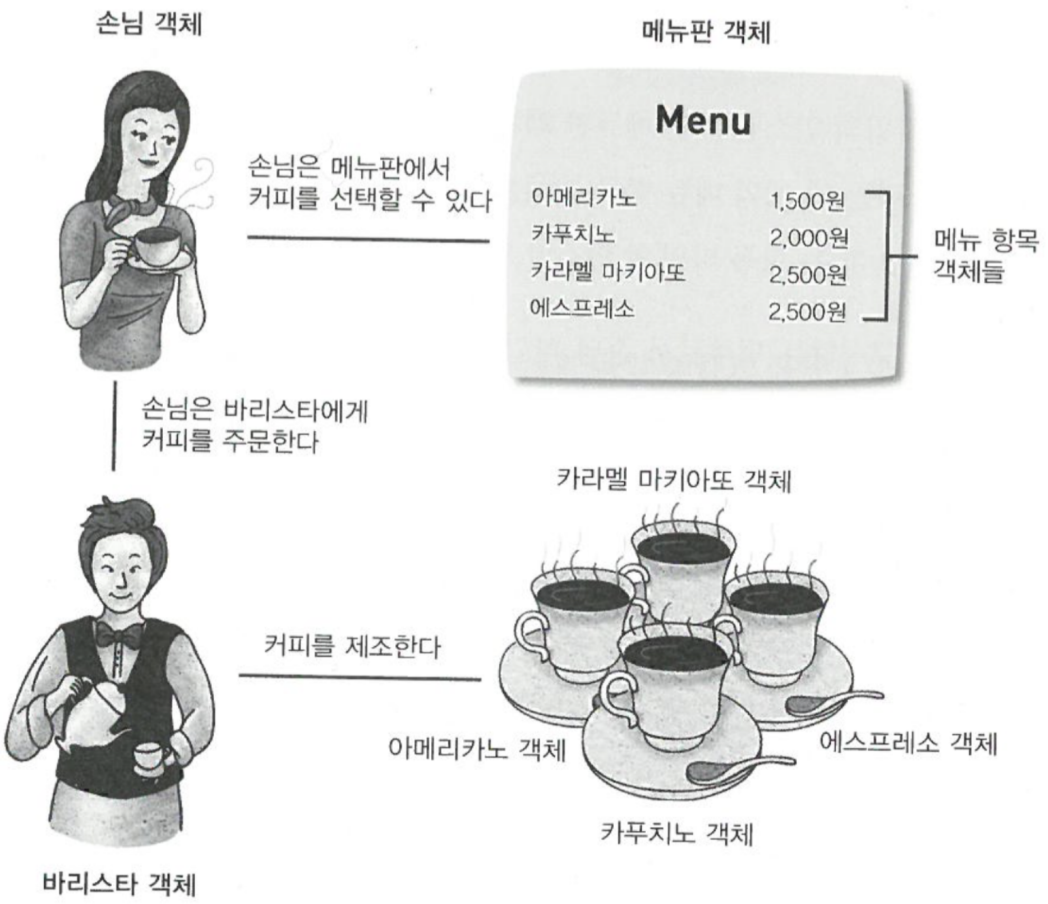
객체지향 프로그래밍은 자료구조와 이를 중심으로 한 모듈들을 먼저 설계하고, 이들의 실행 순서와 흐름을 조합하는 방식이다.

목적

절차지향 프로그래밍은 실행순서, 즉 절차가 중심이 된다.

객체지향 프로그래밍은 필요한 객체들의 종류와 속성들이 더 중심이 된다.

카페를 객체지향적으로 생각하기



[출처] 우아한형제들 기술 블로그, 책 '객체지향의 사실과 오해'

각 객체 간 관계를 파악하기

- 메뉴판 ↔ 손님 : 손님은 메뉴판에서 메뉴를 선택한다.
- 바리스타 ↔ 손님 : 손님은 바리스타에게 커피를 주문한다.
- 바리스타 ↔ 커피 : 바리스타는 커피를 만든다.

객체들을 분류하기

- 손님 객체는 '손님 타입'의 인스턴스
- 바리스타 객체는 '바리스타 타입'의 인스턴스
- 4가지 커피 모두 '커피 타입'의 인스턴스
- 메뉴판 객체는 '메뉴판 타입'의 인스턴스
- 4가지 메뉴 항목 객체들 모두 동일한 '메뉴 항목 타입'의 인스턴스

- 메뉴판 객체는 4개의 메뉴 항목 객체를 포함

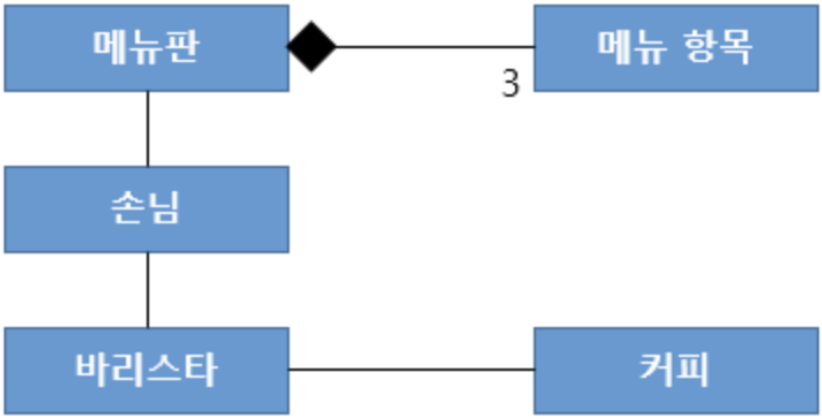
도메인 모델

도메인이란 소프트웨어로 해결하고자 하는 문제 영역을 의미한다. 여기서 도메인은 '카페'가 될 것이다.

'카페'라는 도메인은 '주문', '제조', '서빙' 등의 하위 도메인으로 쪼개질 수 있다.

도메인 모델은 특정 도메인은 개념적으로 표현한 것이다. 도메인을 모델링하는 방법은 여러 가지가 있는데,

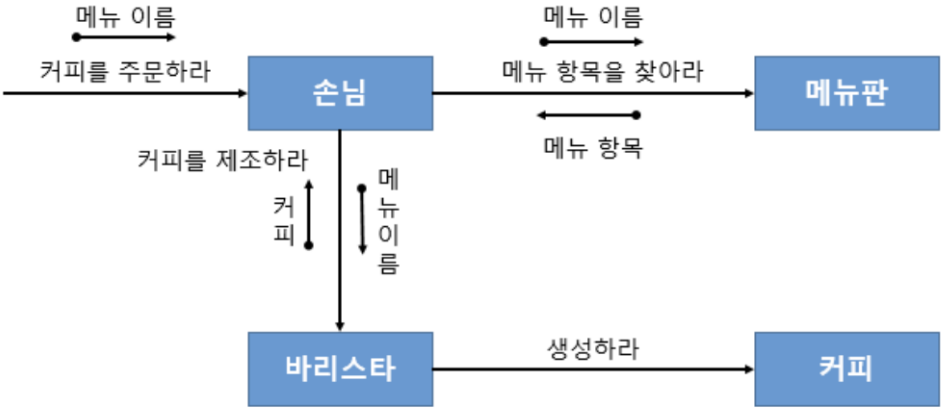
아래는 카페와 관련된 객체들을 타입과 관계를 이용해 추상화한 도메인 모델이다.



메뉴 항목은 메뉴판에 포함(합성)되는 관계이다. 이 때, 마름모 기호로 표현하며, 1:N 관계임을 알 수 있다.

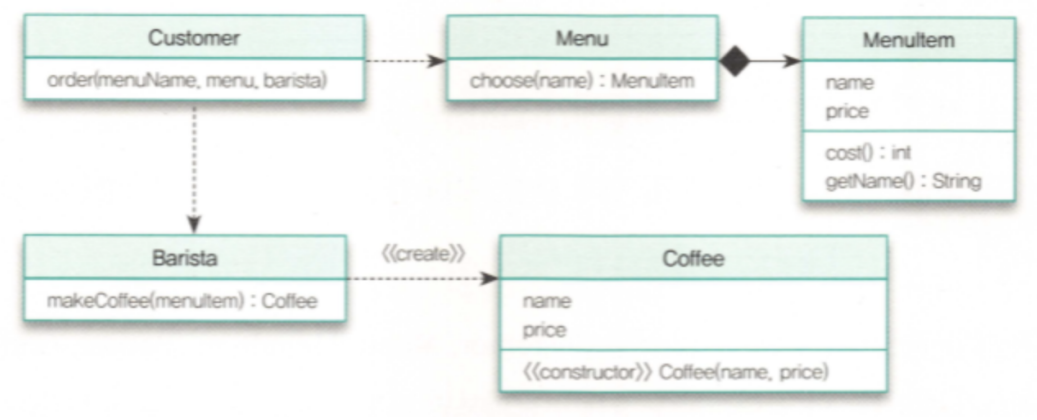
손님과 메뉴판은 포함(합성) 관계는 아니지만, 서로 알고 있어야 할 연관 관계이다.

객체지향적으로 설계하기



- 훌륭한 객체를 설계하기보다, 훌륭한 협력 관계를 설계하라.
- 객체가 메시지를 선택하는 것이 아닌, 메시지가 객체를 선택하도록 해라.
- 메시지를 선택하고, 이를 수신하기에 적합한 객체를 선택해라.
- 예를 들어, `커피를 주문하라` 라는 메시지는 손님이 수신하여야 한다.
- 손님 객체는 `커피를 주문하라` 라는 책임을 할당 받았다. 손님이 할당된 책임을 수행하는 도중에 스스로 할 수 없는 것들은 다른 객체에게 도움을 요청해야 한다.
- 손님은 주문을 하기 위해 `메뉴 항목을 찾아라` 라는 요청 또한 해야한다. 이는 메뉴판 객체가 해줄 수 있다. 손님은 메뉴판을 통해 메뉴 항목을 얻었고, `커피를 제조` 해 달라고 요청한다.
- 이 메시지는 바리스타만이 할 수 있다. 손님은 메뉴 항목을 메시지의 인자로 바리스타에게 전달하고 반환값으로 제조된 커피를 받아야 한다.
- 바리스타가 커피를 만들면 커피 주문의 협력은 끝난다.

클래스 다이어그램



오퍼레이션 순서

1. `손님`은 `메뉴판`에게 메뉴 항목 이름에 해당하는 `메뉴 항목`을 요청
2. `메뉴 항목`을 받아 `바리스타`에게 원하는 `커피`를 제조하도록 요청

메뉴 항목

```
public class MenuItem{
  private String name;
  private int price

  public MenuItem(String name, int price){
    this.name = name;
    this.price = price;
  }

  public int cost() {
    return price;
  }

  public String getName() {
    return name;
  }
}
```

커피

```
class Coffee {
  private String name;
  private int price;

  public Coffee(MenuItem menuItem){
    this.name = menuItem.getName();
    this.price = menuItem.cost();
  }
}
```

메뉴판

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

  public Menu(List<MenuItem> items){
```

```
    this.items = items;
  }

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

바리스타

```
class Barista {
  public Coffee makeCoffee(MenuItem menuItem){
    Coffee coffee = new Coffee(menuItem);
    return coffee;
  }
}
```

손님

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

객체지향 프로그래밍의 4가지 특징

1. 캡슐화

데이터와 코드의 형태를 외부로부터 알 수 없게 하고, 데이터의 구조와 역할, 기능을 하나의 캡슐 형태로 만드는 방법

1) 멤버 변수 앞에 접근 제어자 private를 붙인다. (private : 자기 클래스에서만 접근할 수 있는 것)

2) 멤버 변수에 값을 넣고 꺼내 올 수 있는 메서드를 만든다. (접두어 set/get을 사용해 메서드를 만든다.)

응집도를 높이고, 결합도는 낮추어야 한다. 외부에서 접근할 필요 없는 것들은 private으로 제한한다.

ex) 마트에서 구매자는 상품명과 가격만 알면될 뿐, 뒤에 원재료와 재질각 종 다양한 정보는 몰라도 된다 .

2. 다형성

하나의 메소드나 클래스가 다양한 방법으로 동작하는 것

- 일반적으로 오버라이딩이나 오버로딩을 의미
- **오버로딩** : 같은 동작을 해야 하는 메소드를 작성해야 하는데 매개변수가 다른 경우, 같은 이름의 메서드를 여러개 정의하고, 매개변수의 유형과 개수를 다르게 하여 다양한 유형의 호출에 응답할 수 있게 하는 것.
- **오버라이딩** : 조상클래스로부터 상속받은 메서드의 내용을 변경하는 것

3. 상속화

상위 클래스의 자료와 연산을 하위 클래스가 물려받아 이용할 수 있게 하는 것

- 여러 클래스의 공통 특징을 뽑아 상위 클래스로 만든 것.
- 코드의 재사용성을 높여줌.
- 현실세계의 relationship을 보다 직관적인 방법으로 따라한다.

4. 추상화

공통적인 특성(변수, 메소드)들을 묶어 표현하는 것

- 예를 들어, 음식 이라는 추상화 집단을 만든다면, 음식이 가진 공통적인 상태, 동작 등을 묶어 하나의 집합으로 표현할 수 있다.
- 추상화를 통해 정의된 자료형을 추상 자료형이라고 한다.
- 추상 자료형은 자료형의 data와 operation을 캡슐화한 것으로, 접근 제어를 통해 이를 은닉할 수 있다.

  추상 자료형 class

  class를 실제로 구현한 것 ⇒ instance

  class 내의 data ⇒ member variable

  class 내의 operation (function) ⇒ method

OOP의 장점

- 객체를 중심으로 프로그래밍하므로 사람의 관점에서 프로그램을 이해하고 파악하기 쉽다
- 재사용성, 확장성, 융통성이 높다

객체지향의 5가지 원칙 (SOLID)

SOLID 객체 지향 원칙을 적용하면 **코드를 확장**하고 **유지 보수 관리**하기가 더 쉬워지며, 불필요한 **복잡성을 제거**해 리팩토링에 소요되는 시간을 줄임으로써 프로젝트 **개발의 생산성**을 높일 수 있다.

S	SRP, Single Responsibility Principle 단일 책임 원칙	객체는 단 하나의 책임만 가져야 한다
O	OCP, Open Close Prinicple 개방 폐쇄 원칙	기존의 코드를 변경하지 않으면서 기능을 추가할 수 있도록 설계가 되어야 한다
L	LSP, Liskov Substitution Principle리스코프 치환 원칙	일반화 관계에 대한 이야기며, 자식 클래스는 최소한 자신의 부모 클래스 에서 가능한 행위는 수행할 수 있어야 한다
I	ISP, Interface Segregation Principle 인터페이스 분리 원칙	인터페이스를 클라이언트에 특화되도록 분리시키라는 설계 원칙
D	DIP, Dependency Inversion Principle 의존 역전 원칙	고수준 모듈은 저수준 모듈의 구현에 의존해서는 안된다. 의존 관계를 맺을 때 변화하기 쉬운 것 또는 자주 변화하는 것보다는 변화하기 어려운 것, 거의 변화가 없는 것에 의존하라는 것이다

S

청소기 클래스는 청소 메소드만 잘 구현하면 되지 화분에 물을 주고 드라이 까지 할 책임은 없다.

물론 다재다능한 청소기는 좋아 보이지만, 만일 청소기가 고장나면 다른 기능까지도 사용을 못해지기 때문이다.

즉, 청소기는 청소만 잘하면 된다는 책임만 가지면 된다. 하나의 클래스로 너무 많은 일을 하지 말고 딱 한 가지

책임만 수행하라는 뜻으로 보면 된다.

O

기능 추가 요청이 오면 **클래스를 확장을 통해 손쉽게 구현**하면서, **확장에 따른 클래스 수정은 최소화**하도록 프로

그램을 작성해야 하는 설계 기법이다. **추상화** 사용을 통해 관계를 구축하라는 말이다.

L

다형성 원리를 이용하기 위한 원칙이다.

자바에선 대표적으로 Collection 인터페이스를 LSP의 예로 들수있다.

Collection 타입의 객체에서 자료형을 LinkedList에서 전혀 다른 자료형 HashSet으로 바꿔도

add() 메서드를 실행하는데 있어 원래 의도대로 작동되기 때문이다.

한마디로 다형성 이용을 위해 부모 타입으로 메서드를 실행해도 의도대로 실행되도록 구성을 해줘야 하는

원칙이라 이해하면 된다.

I

SRP 원칙의 목표는 클래스 분리를 통하여 이루어진다면, ISP 원칙은 인터페이스 분리를 통해 설계하는 원칙.

`운전자가 자동차를 운전한다` 라는 명제를 객체간 관계로 비유하면

자동차에 대한 인터페이스, 운전자에 대한 인터페이스를 각각 분리하는 것이다.

```
interface Person {
  void cook();
  void plate();
  void order();
  void pickup();
  void eat();
}
```

```
interface Cook {
  void cook();
  void plate();
}

interface Customer {
  void order();
  void pickup();
  void eat();
}
```

D

DIP 원칙은 어떤 Class를 참조해서 사용해야하는 상황이 생긴다면, 그 Class를 직접 참조하는 것이 아니라 그 **대상의 상위 요소(추상 클래스 or 인터페이스)로 참조**하라는 원칙이다.