[240320] 2주차

4장 주석

> 나쁜 코드에 주석을 달지 마라. 새로 짜라

- 주석을 엄격하게 관리하기 보다는 코드를 깔끔하고 표현력이 강하게 작성
- 코드만이 진실된 정보를 제공하는 유일한 출처이다.

주석은 나쁜 코드를 보완하지 못한다

- 코드에 주석을 추가하는 일반적인 이유

  코드 품질이 나쁘기 때문

- 표현력이 풍부하고 깔끔하며, 주석이 거의 없는 코드 > 복잡하고 어수선하고 주석이 많은 코드

코드로 의도를 표현하라

불분명한 Flag의 의미, 언제든 바뀔 수 있는 조건들

```
// bad 🥵
if ((e.flag & HOUR_FLAG) && (e.age > 65))

// good 🌟
if (e.isEligibleForFullBenefits())
```

좋은 주석

- 법적인 주석
  저작권, 소유권, 표준 라이선스
- 정보 제공
  반환 값(함수 이름을 명확하게 쓰는게 나음)
  복잡한 정규표현식 등

- 의도 설명
  저자의 결정에 깔린 의도를 설명

  조금 뜬금 없거나 문맥 없이 예측하기 어려운 (정렬 우선 순위 커스텀, 테스트를 위한 경쟁 조건 생성)
- 의미를 명료하게 밝힘
  정확히 달아야 한다. 주석이 올바른지 검증하기 쉽지 않다.
- 결과 경고
  일반적인 습관대로 작성하면 thread-safe 하지 않은 코드

  [240320] Thread Safety
- TODO
  앞으로 할 일을 주석으로 남겨두면 편하다. IDE도 TODO 찾아주는 기능 제공
- 중요성 강조
  중요해보이지 않지만 매우 중요한 무언가를 강조하기 위함
  공백 제거, 대소문자 구별 등
- 공개 API Javadocs
  설명이 잘 된 공개 API는 매우 유용하다. 하지만 항상 주석은 정확한 정보를 전달하도록 주의

나쁜 주석

- 주절거리는 주석
  정확한 의미를 찾기 위해서는 추가적인 노력을 들여야 하는, 대충 설명한 주석
- 같은 이야기 중복
  명확하게 읽히는 코드 내용을 그대로 풀어 쓰고, 가독성도 낮은 주석
- 오해할 여지
  의도는 좋았지만 엄밀하지 못해 오해의 소지가 있는 주석
- 의무적
  모든 함수에 적용되는 의무적인 주석: 가독성 및 오류 가능성 높임
- 이력 기록
  그냥 지우는 편이 낫다. 소스 코드 관리 시스템이 있음
- 있으나 마나
  생성자에 생성자라고 설명 달기, 아무 얘기나 써놓기
- 무서운 잡음
- 함수나 변수로 표현 가능

- 위치 표시
  남용하지 말고 아주 드물게 사용하는건 괜찮다.
- 닫는 괄호 옆
  차라리 함수를 깔끔하게 분리하자.
- 공로 또는 저자
- 주석으로 처리한 코드
  다른 사람도 지우기를 주저하고, 점점 쌓여간다. 버전 관리 시스템을 사용하자.
- HTML
- 전역 정보
  해당 함수 또는 클래스가 제어하지 못하는 정보는 불필요하다.
- 너무 많은 정보
- 모호한 관계
  코드와 주석은 관계가 명확해야 한다. 추가 정보가 필요한 주석은 bad
- 함수 헤더
- 비공개 코드 Javadocs

5장 형식 맞추기

형식을 맞추는 목적

당장 돌아가는 코드가 중요한 건 맞다.
하지만 오늘 코드의 가독성이 앞으로 바뀔 코드 품질에 영향을 미친다.

적절한 행 길이를 유지하라

- 500줄 미만 파일로도 대규모 시스템을 구축할 수 있다.(JUnit, FitNesse)
- 신문 기사처럼 위에서 아래로, 첫 부분은 고차원, 뒤로 갈수록 상세, 짧고 읽기 좋게
- 다른 개념은 빈 행으로 분리(선언문, 함수 사이 등)
- 밀접한 코드는 같은 파일에, 세로로 가까이 배치한다.

  이게 protected 변수를 피하는 이유? 잘 모르겠음
- 지역 변수는 사용 위치에 최대한 가깝게, 인스턴스 변수는 클래스 맨 처음에

- 호출하는 함수는 가깝게, 호출되는 함수를 뒤에 놓으면 가독성 자연스러움

가로 형식 맞추기

가로 정렬 굳이 사용하지 않는다. 정렬해야 할 만큼 멤버 변수가 많다면 클래스를 쪼개야 한다.

팀 규칙

팀에 속한다면 합의된 규칙에 따라라. 일관된 스타일이 독자에게 신뢰감을 준다.

6장 객체와 자료 구조

자료 추상화

- 구현을 감추려면 추상화가 필요하다. **사용자는 구현을 모른 채** 자료의 핵심을 조작할 수 있어야 한다.
- 아무 생각 없는 getter/setter는 나쁘다. 객체의 데이터를 표현할 좋은 방법을 고민해야 한다.

자료/객체 비대칭

- 객체는 추상화 뒤로 데이터를 숨기고 함수만 공개
- 자료구조는 데이터를 그대로 공개, 별다른 함수는 제공하지 않음
- 객체지향
  함수를 변경하지 않고 새로운 클래스를 추가하기 쉽다.
  새로운 함수를 추가하기 어렵다. 모든 클래스를 고쳐야 한다.

  인터페이스나 추상 클래스에 새로운 (추상) 메서드를 추가할 경우, 이를 구현하는 모든 클래스에 해당 메서드를 구현해야 한다. 기존 코드 변경 없이 인터페이스를 따르는 새로운 클래스를 구현하기는 쉽다.
- 절차지향
  자료 구조를 변경하지 않고 새로운 함수를 추가하기 쉽다.
  새로운 자료 구조를 추가하기 어렵다. 모든 함수를 고쳐야 한다.

  특정 자료 구조를 그대로 이용하므로, 함수를 추가하기는 쉽다. 특정 함수는 자료 구조의 디테일을 바로 사용하기 때문에, 새로운 자료구조를 다룰 필요가 생길 시 함수 내부를 다 변경해야 한다.

디미터 법칙

모듈은 자신이 조작하는 객체의 속사정을 몰라야 한다. 객체는 내부 구조를 공개하면 안된다.

- 클래스 C의 메서드 f는 다음과 같은 객체의 메서드만 호출해야 한다.
  - 클래스 C (본인이 속한 클래스)
  - f가 생성한 객체
  - f의 인수로 넘어온 객체
  - C 인스턴스 변수에 저장된 객체
- 기차 충돌 방지
  - getA().getB().getC()……
  - 최대 dot(.) 하나로만 접근 가능하도록
  - stream.map().filter().collect()
    → 기차 충돌이 아니다. 내부 데이터를 계속 타고 들어가는게 아니라 메서드가 이어질 수 있도록 stream을 반환하는 것.
- 자료구조라면 내부 데이터를 사용한다고 해도 상관 없다.
- 자료구조와 객체가 섞인 '잡종 구조'를 피하자.
- **객체에게 무언가를 말하라고 요청해야지 속을 드러내라고 하면 안된다.**
- c.getAbsolutePathOfDirectoryOption()
  → 절대경로를 받아서 어디에 쓰려고?
  → 필요한 데이터를 묻지 말고
  (캡슐화) 시켜라 (객체의 자율성)

  → c.createFileStream()

자료 전달 객체

DTO, 활성 레코드(save, find 등이 포함된 DTO)는 자료 구조로 본다.
비즈니스 로직이 있고, 내부 데이터를 숨기는 객체는 따로 생성한다.

DTO ↔ Domain

결론

새로운 자료 타입이 유연하게 필요하다 → 객체지향

새로운 동작이 유연하게 필요하다 → 자료구조, 절차적

7장 오류 처리

오류 코드보다 예외를 사용하라

- 오류 플래그 설정, 오류 코드 반환하면 호출하는 코드가 복잡해진다.
- 예외를 던지면 개념적으로 코드가 분리되어 품질이 향상된다.

Try-Catch-Finally 문부터 작성하라

- 예외를 일으키는 테스트 작성 시, 강제로 예외를 일으키는 케이스를 작성한다.
- try-catch 구조로 범위를 정의하고 리팩터링한다.
  try 문은 일종의 트랜잭션으로, 범위 정의 후 나머지 부분은 오류나 예외가 전혀 발생하지 않는다고 가정한다.
- try 블록의 트랜잭션 범위부터 구현하게 되므로 범위 내 트랜잭션 본질을 유지하기 쉬워진다.

미확인(unchecked) 예외를 사용하라

- checked 예외는 OCP를 위반한다.
  하위 단계 코드가 새로운 checked 예외를 던지면, 상위 코드가 모두 선언부를 변경해야 한다.
  (catch블록을 추가하거나 throws 선언부 추가)
- 중요한 라이브러리를 작성하지 않는 이상 checked 예외는 불필요하다.

예외에 의미를 제공하라

- 호출 스택으로는 부족하다. 오류 메시지에 정보를 담아 함께 던진다.(연산 이름, 실패 유형 등)

호출자를 고려해 예외 클래스를 정의하라

- 오류 정의 시 가장 중요한 관심사는 **오류를 잡아내는 방법**이어야 한다.
  오류가 발생한 컴포넌트, 오류 발생 유형 등
- 외부 API를 사용할 경우, wrapper 클래스가 유용하다.
  ACMEPort.open(): DeviseException, UnLockedException, ...

  → LocalPort.open()
- 외부 API 의존성을 줄이고, 테스트 코드 작성이 쉬워진다.

정상 흐름을 정의하라

- 특수 사례 패턴: 클래스를 만들거나 객체를 조작해 특수 사례를 처리한다.
  예외적인 상황을 처리할 필요가 없어진다.

```
(ex) 할인쿠폰 DiscountCoupon::getDiscountMoney(int originalMoney)
// 정상 사례
  DiscountCoupon coupon = createCoupon();
```

```
    int discounted = coupon.getDiscountMoney(int originalMoney)
-----------------------------------------------------------------
// 허술 사례 #1
  int discounted = -1;
    DiscountCoupon coupon = createCoupon();
    if (coupon == null)
      discounted = 0;
    else
      discounted = coupon.getDiscountMoney(int originalMoney)

// 허술 사례 #2
    try {
      DiscountCoupon coupon = createCoupon();
      discounted = coupon.getDiscountMoney(int originalMoney)
    } catch (CouponNotFoundException e) {
      discounted = 0;
    }

-----------------------------------------------------------------
// 특수 사례 : 할인 금액이 없는 특수 쿠폰 객체
class NoDiscountCoupon:
  int getDiscountMoney(int origin):
    return 0
```

null을 반환하지 마라

- 메서드에서 null을 반환하고 싶다면, 예외를 던지거나 특수 사례 객체를 반환하라
- 외부 API가 null을 던진다면, Wrapper 메서드를 구현해 예외를 던지거나 특수 사례 객체를 반환
- for(E e: employees) ← employees가 깔끔하게 empty List로 들어오면 된다.

null을 전달하지 마라

- 정상적인 인수로 null을 기대하는 메서드가 아니라면 최대한 피한다.
- assert 문을 사용하는 방법도 있다.

  VM 옵션에 따라 실행 안될 수 있다고 한다.

- null check → InvalidException
  NPE는 피하지만, 추가적인 예외 처리 필요하다.

  CustomExceptionHandler 사용하면 될듯
- Objects.requireNonNull
  NPE를 던지지만, 가독성이 좋다.

결론

깨끗한 코드는 읽기 좋고 안정성도 높아야 한다.

오류 처리를 프로그램 논리와 분리해야 독립적 추론이 가능하고 유지보수성이 높아진다.