[240313] 1주차

개인 생각 및 첨언은 색으로 표현

1장 깨끗한 코드

코드가 존재하리라

- 코드는 요구사항을 상세히 표현하는 수단, **요구사항을 표현하는 언어**

나쁜 코드

- 좋은 코드의 중요성과 르블랑의 법칙(LeBlanc's Law, '나중은 결코 오지 않는다')

나쁜 코드로 치르는 대가

- 쌓일수록 떨어지는 팀 생산성
- 인력 투입과 비례하지 않는 생산성 증가

  (Brook's Law, 지연되는 프로젝트에 인력 투입하면 더 느려짐)
- 깨끗한 코드란 한 가지를 잘 한다. 잘 쓴 문장처럼 읽힌다. 주의 깊게 짰다. 짐작했던 기능을 그대로 수행한다.

우리들 생각

- pass

우리는 저자다

- 내 코드를 읽는 독자와 소통할 책임을 기억하라.
- 코드는 읽는 비중이 훨씬 높다. 읽기 쉬운 코드가 중요하다.

보이스카우트 규칙

- 언제나 깨끗하게, 처음보다 더 깨끗하게 유지한다.

프리퀄과 원칙, 결론

- pass

2장 의미 있는 이름

들어가면서

- 소프트웨어에서 모든 곳에 이름이 쓰인다. 이름을 잘 지으면 여러모로 편하다.

의도를 분명히 밝혀라

- 존재 이유는? 수행 기능은? 사용 방법은?
  별도 주석이 필요하다면 의도를 분명히 드러내지 못했을 가능성
- int d → int daysSinceCreation
- 단순성이 아닌, 함축성이 문제 **독자가 무언가 알거라 가정하지 말것**
  theList ? →
  gameBoard
  index 0 ? →
  STATUS_VALUE
  상수 4 ? →
  FLAGGED
  반환값의 용도는? →
  return flaggedCells

  코드 맥락을 코드 자체에 명시적으로 드러내라
  primitive 값 대신 특정 역할을 위한 class로 개선
  List<int[]> → List<Cell>

그릇된 정보를 피하라

- 부적합한 약어 사용 자제
- 실제 List 가 아니라면 ~List와 같은 변수명 사용하지 않는다.
  (복수형 또는 group, bunch 등 적절한 단어 사용)

- 흡사한 이름 사용 주의
  XYZControllerForSpecificResult ↔ XYZControllerForHandleResult

의미 있게 구분하라

- 컴파일러나 인터프리터 통과 목적으로 코드 구현 시 문제 발생 확률 높음
  - a1, a2, ... 등 연속된 숫자
  - ~Info, ~Data, NameString, Customer↔CustomerObject 등 불용어 사용

발음하기 쉬운 이름을 사용하라

- 두뇌는 단어라는 개념에 전적으로 의지한다.
- 대화의 편의성

검색하기 쉬운 이름을 사용하라

- 숫자, 문자 하나 등은 검색하기 까다롭다.
- 단순한 메서드의 로컬 변수 이외에는 한 문자 사용하지 않는다.
- 이름 길이는 범위에 비례하도록 한다.
- 여러 곳에서 사용한다면 검색하기 쉬운 이름이 바람직하다.
  s → sumOfTask
  34 → NUMBER_OF_TASKS

인코딩을 피하라

- 정신적 부담 + 발음 어렵고 오타 발생 확률 높음
- 접두어 방식은 구식 코드의 징표

자신의 기억력을 자랑하지 마라

- 명료하게 작성하여 남들이 이해하는 코드가 좋은 코드

클래스 이름

- 명사구가 적합
- Manager, Info, Data와 같은 추상적인 단어와 동사는 자제

메서드 이름

- 동사나 동사구가 적합
- get, set, is 접두사는 javaBean표준에 따라 사용
- 생성자 정의 시 정적 팩토리 메서드
  new Product(3000) → Product.withPrice(3000)

기발한 이름은 피하라

- 의도를 명료하고 분명하고 솔직하게 표현

한 개념에 한 단어를 사용

- 안좋은 예시
  조회: get, fetch, retrieve, ...
  관리: controller, manager, driver, ...
- 일관성 있는 어휘 사용이 혼란을 줄임

말장난을 하지 마라

- 숫자 덧셈도 add, 문자열 붙임도 add, 리스트 추가도 add면 혼란
- insert, append 등 **의도를 밝힐 책임을 가지고** 문맥에 맞는 단어 사용

해법 영역에서 가져온 이름을 사용하라

- 코드 독자도 프로그래머: 전산, 수학, 알고리즘 용어 사용은 가능
  기술 개념에는 기술 이름이 적합
- 모든 이름을 domain 단어로 사용하는 정책은 현명하지 않음

문제 영역에서 가져온 이름을 사용하라

- 적합한 프로그래밍 용어가 없다면 문제 영역에서 단어 사용
  해당 분야 전문가에게 의미 질문 가능
- 관련이 깊은 맥락에서 이름을 가져와야 한다.

의미 있는 맥락을 추가하라

- 독자가 맥락을 유추하도록 하지 않는다.
- 함수를 쪼개고 흩어진 변수는 적절한 클래스로 구성해 확실한 맥락을 구성
- number → StatisticMessage.number

불필요한 맥락을 없애라

- 일반적으로 (의미가 분명한) 짧은 이름이 긴 이름보다 좋다.
- OkestroEmployee.class, OkestroProduct.class, ...
  불필요한 맥락을 만드는 'Okestro' 접두사
- customerAddress : 변수 이름으로 적합 ↔
  Address: 클래스 이름으로 적합

마치면서

- 좋은 이름을 선택하려면 뛰어난 설명 능력과 동일한 문화적 배경이 필요하다.
- 이름 개선은 단기적/장기적 장점이 있다.

3장 함수

작게 만들어라

- 작게, 더 작게
- 가로 150자 미만, 세로 20줄 미만
- 중첩 구조 자제, 들여쓰기 수준은 최대 2단

한 가지만 해라

- 추상화 수준이 하나인 단계만 수행한다.
- 의미 있는 이름으로 다른 함수를 추출할 수 있다면, 여러 작업을 하는 함수

함수 당 추상화 수준은 하나로

- 함수 내 모든 문장의 추상화 수준은 동일해야 함

- 근본 개념인지 세부 구현인지 헷갈리지 않아야 함
- 위에서 아래로 읽기 좋은 코드 (내려가기 규칙)

Switch 문

- Switch문을 완전히 피할 방법은 없음
- 저차원 클래스에 숨기고 다형성을 이용하여 반복 제거

서술적인 이름을 사용하라

- 길고 서술적인 이름이 낫다
  짧고 어려운 이름보다, 길고 서술적인 주석보다
- 설계가 명확해지고 코드 개선이 쉬워짐
- 모듈 내 함수 이름의 일관성을 지켜야함

함수 인수

- 이상적인 함수의 인수 개수: 최대 2개
  0개가 최선, 1개(단항)가 차선
- 3개는 가능한 피하고, 4개 이상은 특별한 이유 있어도 자제
- 인수가 많아질수록 테스트 부담
  가끔
  0개보단 1개가 테스트 낫지 않나? 의존성 외부로 추출
  현재 시간에 스케줄을 등록하는 메서드를 가정,
  registerSchedule() → registerSchedule(LocalDateTime.now());
- 단항 함수 예시
  - 질문을 던지는 함수 isEven(number)
  - 반환하는 함수 InputStream fileOpen(”MyFile”)
  - 이벤트 함수(외부 상태를 바꾸므로 역할이 명확히 드러나도록)
- 입력 인수를 변환하는 함수라면, 결과는 반환값으로 돌려준다.
  변환 함수에서 가급적 출력 인수를 사용하지 않는다.
- 플래그 인수 → 분기를 나타내는 우아하지 못한 함수
- 이항 함수 → 적절한 경우도 있지만, 가급적 피한다.

- assertEquals(expected, actual)의 예시

  assertThat(expected).isEqual(actual) 그래서 이걸 더 많이 쓰나
- 인수가 2개 이상 필요하다면 일부를 클래스 변수로 고려
- 함수 이름 또한 인수의 순서와 의도를 표현하도록
  - write(name) → writeField(name)
  - assertEquals(e, a) → assertExpectedEqualsActual(e, a)

부수 효과를 일으키지 마라

- 특정 상황에서만(부수 효과를 통과할 수 있는) 함수는 혼란을 일으킴
- 되도록 함수 이름에 명시하는게 좋은 방법
  '한 가지만 한다'는 규칙을 위반하지만.
- 객체지향적으로 구성하면 출력 인수 피하는게 좋다
  함수가 속한 객체 상태를 변경하는 방식 선택

명령과 조회를 분리하라

- 무언가를 수행(명령)하며 상태를 반환(조회)하면 의도하지 않은 혼란 발생
- CQ(R)S pattern이란게 있더라.. 명령인지 조회인지 따라 controller 분리, service 분리, DB 분리 등 어려운 내용

오류 코드보다 예외를 사용하라

- 오류 처리 코드를 위한 불필요한 코드 생성
- try - catch 블록은 해당 코드가 필요한 부분을 별도로 추출
- 정상 동작과 오류 처리 동작을 분리하면 코드 이해 및 수정 용이

반복하지 마라

- 중복은 악의 근원
  DB 정규형, 상속, AOP 등등 많은 기법이 중복 제거 또는 제어 목적 전략

구조적 프로그래밍

- 함수와 함수 내 블록은 입구와 출구가 하나여야 한다.
  one return
- 루프 내부 break, continue 자제
  - 하지만 함수를 작게 만든다면 굳이 그럴 필요 없음
  - 오히려 의도를 표현하기 더 쉬워진다.
- goto는 절대 금지

함수를 어떻게 짜죠?

- 글짓기와 유사하다. 꾸준히 다듬는다.

결론

- 프로그래밍의 기술은 언어 설계의 기술이다.
- 분명하고 정확한 언어로 깔끔하게 작성하기 위해 노력