Clean Code: ch.2 and 3

ch.2: Meaningful Names

이름을 잘 짓는 방법

- 그것의 사용 목적, 역할을 드러내자: 훨씬 빠른 코드 이해가 가능하다.

  예_ val a = 22 / **val age = 22**

- 착각하기 쉬운 이름은 사용하지 않는다: keyword로 등록된 단어를 쓰거나 기능이 다른데 비슷한 이름으로 짓는 등의 행동은 프로그래머에게 혼란을 준다

  예_ AcountList → 근데 실제 리스트 타입이 아닌 경우

  예_ 1과 I을 함께 쓰는 경우

- 다른 개념이라면 확실히 구분되는 용어를 사용하자

  예_ val Product / def ProductInfo → 이름이 철자만 다르고 의미는 같다

- 발음하기 쉬운 단어를 사용하자: 읽기 쉬워야 다른 개발자와 소통이 편해진다

- 검색하기 쉬운 단어 사용: e나 숫자 1개는 찾기 어렵다.

- interface class와 concrete class의 이름은 인코딩을 이용한다

  : interface class는 그냥 이름을 사용하고 concrete class에 C(이름) 이런식으로 이름을 구분한다.

- class name과 method name

  : class name은 명사 또는 명사 구 / method name은 동사 또는 동사구

  예_ class Account { def getName(…) }

- 추상적인 개념 하나당 단어 하나를 사용하자(일관성)

  : 예를 들어 클래스마다 정보를 지우는 매서드가 있을 때, delete, remove, erase 등등 같은 의미의 여러 단어를 사용하면 혼란을 키운다.

  - 그러나 맥락이 다른데 하나의 단어를 사용한다고 고집하면 안된다.
    예_ def add(a, b) = a + b / def add (a) = append “a” in a list

- 불필요한 단어는 사용하지 않는다.

  예_ 사용자 이메일 정보를 저장하는 클래스 이름: CustomerAddress / **Address**

💡 1. 변수의 역할과 맥락이 잘 드러나게 작성할 것!
2. 헷갈리거나 불필요한 단어는 사용하지 말 것!

ch.3: Functions

함수 잘 만드는 방법

- 함수는 작게 만들수록 유리하다: 중첩 구조가 들어가거나 여러줄이 되지 않도록 노력하자
- 하나의 함수는 한가지 기능을 수행한다

  : 함수의 이름 아래에서 추상화 수준이 하나인 경우 한가지 기능을 가진 것이다.

  : 더이상 작은 함수로 쪼갤 수 없으면 한가지 기능을 수행한다.
- 함수 내에서는 모두 동일한 abstraction level을 갖도록 해야한다.

  예_ .getHtml() : high abstraction level

  PathParser.render(pagepath) : middle abstraction level

  .append("\n") : low abstraction level
  - 하나의 abstraction level로 구성할 수 없는 경우

    : 위에서 아래로 내려가면서 abstraction level이 한번에 한 단계씩 낮아지도록 작성
- 서술적인 이름을 사용하기: 길어도 서술적인 함수 이름을 사용

  예_ 함수 이름으로 testableHtml / **SetupTeardownIncluder**
- 인수 개수를 적게!
- Side effect를 주의하기: 예상치 못한 오류가 발생할 수 있다.

  예_ 아래 코드는 password를 check하는 함수에서 session도 초기화한다. 따라서 session이 초기화되면 안되는 상황에 호출될 경우 오류를 만든다.

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

- Command와 Query를 분리하자: 함수는 무언가 수행하거나 무언가에 답하거나 둘 중 하나의 기능만 해야한다. 두 가지 경우가 모두 argument에 있다면 실제로 값을 넣었을 때, 의미를 제대로 해석하기 어렵다.

  예_ 위 if문은 username을 unclebob으로 바꾸라는 건지, username값을 unclebob으로 설정하라는 건지 혼동된다.

  ```
  public boolean set(String attribute, String value);
  if(set("username","unclebob")) ...

  if(attributeExixts("username")) {
    setAttribute("username","unclebob")
  }
  ```

- Use Exception(not Error Code): false, -1과 같은 Error code가 반환되면 코드상 바로 밑에서 에러를 다루는 코드를 작성하지만 Exception을 사용하면 기존 코드와 오류 코드가 깔끔하게 분리된다.
  - Try/Catch를 함수로 뽑아내기

    ```
    public void delete(Page page) {
      try {
        deletePageAndAllReferences(page);
      }
      catch (Exception e) {
        logError(e);
      }
    }
    ```

  - 오류를 처리하는 함수도 오류 처리 작업만 해야한다.

- 중복되는 코드를 없애기

Switch문

기본적으로 switch문은 여러 경우의 수를 처리하는 문장이므로 “한 가지” 일만 하는 것이 불가능하다. 그래서 최대한 적게 사용하는 방식으로 구성해야 한다.

polymorphism을 이용하는 것을 추천한다.

예_ abstract factory pattern을 사용함

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:


        return new SalariedEmployee(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

▼ abstract factory pattern이란?

[Design Pattern] 2. Creational Pattern

object의 instantiation을 추상화하는 방법입니다.즉, instance를 만들 때, 어떻게 하면 재사용과 변경에 유용한 구조로 만들 수 있을까에 대한 고민의 결과로 나온 pattern이라고 볼 수 있습니다.

https://euidong.github.io/posts/design-pattern-2

함수 인수: Argument

함수는 인수가 적을수록 좋다: 인수는 함수의 의미를 이해하기 어렵게 한다. 코드를 읽는 시점에 알 필요 없는 구체적인 인수 정보를 해석해야 하기 때문이다.

- Common monadic Form(단항 인수)
  - argument에 질문을 하는 경우: 리턴값이 boolean
  - parameter를 다른 형태로 변환되어 리턴되는 경우
  - event function: input argument만 있고 output argument는 없는 경우
  - 금지되는 경우: void includeSetupPageInto(StringBuffer pageText)

    → output argument를 return value 형태가 아닌 입력 인수로 받음
- Flag
  - argument로 flag를 넘기는 것은 함수 내에서 두가지 이상의 일을 한다는 의미: 추천하지 않음
- Dyadic Function(이항 함수)
  - argument 간 자연적인 순서가 없다면 dyadic function은 argument를 혼동해서 집어넣을 수 있다
- Triads(삼항 함수)
  - 신중히 고려해 꼭 필요한 경우에만 사용
- Argument Objects
  - 인수가 2, 3개 필요하다면 해당 인수들을 class 묶어보자. 개념이 더 명확하게 드러날 수 있다

    예_

    ```
    Circle makeCircle(double x, double y, double radius)
    Circle makeCircle(Point center, double radius)
    ```
- Argument Lists

    - argument 개수가 mutable한 경우, 함수는 보통 이항 함수다
    - 하나의 argument가 List 형태로 여러 argument를 통합해 관리한다
- Verbs and Keywords
    - 함수(동사)와 인수(명사)가 어우러지는 이름을 작성하자

```
write(name)
writeField(name)
```

작성한 코드를 여러 번 다듬고 보완해 위에서 말한 함수를 얻을 수 있다.