Clean Code ch2 / ch3 summary

Clean Code by Robert Cecil Martin. Prentice Hall

chapter 2 : Meaningful Names

이장은 코드를 짤때 variable, function, class 등의 name을 어떻게 만들어야하는지를 알려준다.

세부 topic으로 나누어서 책에서는 rule들을 설명해주고 있다. rule은 다음과 같다.

1. ***Use Intention-Revealing Names***

   name은 intent를 드러내기 쉽다.

   variable,function,class의 이름은 그게 왜 존재하는지, 무엇을 하는지 또 어떻게 쓰이는지에 대해 알려주어야한다.

   예를 들어 지나간 시간을 나타내기 위해 int d를 쓰기보다는 int elapsedTimeInDays, int daysSinceCreation등과 같은 변수를 써야한다.

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

   예를 들어 이런 코드는 theList에 무엇이 들어가는지 알기 힘들고 return 되는 것이 어떤 것인지도 알기 어렵다.

   이건 code의 simplicity가 아니라 **implicity**이다. (context에서 얼마나 explicit하지 않은지 나타내는 정도)

   만약 게임을 하면서 flagged된 cell을 찾는다고 하면

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

이런식으로 코드를 바꾸어서 더 **explicit**하게 할 수 있다.

2. ***Avoid Disinformation***

code의 meaning에 **false clues**를 줄 수 있는 것을 피해야한다.

- 특정한 자료형 ex.List가 아닌 것을 arrayList라는 변수로 선언한다거나,
- similar한 concept을 다르게 표시한다거나,
- lower-case of L이나 uppercase o을 variable에 넣어서

혼동을 야기해서는 안된다.

3. ***make meaningful distinctions***

number-series naming(a1,a2,,,,aN)대신 source와 destination을 배열 대입 연산에 이용하는 것이 좋다.

noise words는 redundant하고 difference를 reader들에게 제공하기 매우 어렵다. 따라서 이를 사용하는 것을 피해야한다.

ex> a, an and the

getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();

→ 이건 매우 잘못된 form이다!

4. Use pronounceable Names

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

위와 아래를 비교했을때 위의 genymdhms는 알아듣기 힘들다. **programming은 social activity이기에** 이걸 이해하기 쉽게 아래와 같이 pronounceable 한 name으로 바꾸는 것이 중요하다.

5. ***Use searchable Names***

single letter name과 numeric constants는 text에서 찾기 어렵다는 문제점이 있다. 따라서 여러번 쓰이는 변수나 상수라면 **search-friendly**한 name을 붙여야 한다.

6. ***Avoid Encodings***

HN (변수 및 함수의 이름 인자 앞에 데이터 타입을 명시하는 규칙)을 쓰면 **type을 바꾸기 어려우므로 하지말도록 한다.**

prefix member variable with m_ 을 쓰지 않아야한다.

7. ***Avoid Mental Mapping***

name을 **mentally translate**해서 다른 이름으로 바꾸면 안된다. 예를 들어, loop counter로 단일 문자를 쓰는 것은 traditional하기에 그대로 쓰는 것이 좋다.

8. ***class Names***

class 와 object는 **noun or noun phrase**가 되어야한다.

9. ***method Names***

verb or verb phrase가 되어야 한다.

10. ***Don’t be Cute***

culture-dependent한 jokes를 쓰면 안된다.

11. ***pick one word per Concept***

하나의 추상적인 개념에 대해 한 단어를 선택하고 고수해야한다.

현대의 IDE들에서는 **context-sensitive clues**를 제공한다. 예를 들어 특정 object에 대해 가능한 methods를 알려준다. 그래서 함수를 선언할때 **parameter에 name**을 적는것이 좋다.

12. ***Don’t Pun***

두가지 목적으로 동일한 단어를 사용하지 않아야 한다.

add가 서로 다른 상황에 쓰인다면 add를 여러 번쓰는 것보다 insert나 append를 대신 사용하는 것이 좋다.

13. ***Use Solution Domain Names & Use problem Domain Names***

솔루션과 문제 영역의 개념을 분리해야한다.

14. ***Add Meaningful Context***

function을 smaller하게 나누고, variable에게 명확한 **context**를 제공해야한다.

Listing 2-2
Variables have a context.

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

예를 들어 다음과 같이 함수를 나누면 number, verb, pluralModifier가 어떠한 context에서 쓰이는지 더 잘 이해할 수 있다.

15. ***Don’t add Gratuituous Context***

모든 것에 접두사를 붙이면 안된다.

good name을 고르는 것은 좋은 **descriptive skill과 공유되는 문화적 배경이** 필요하다. 만약 rename에 성공한다면 정말 큰 improvement를 이끌어낼 것이고 이를 위해 노력해야한다.

---

chapter 3 : Functions

function을 만드는 rule또한 ch2에서 name을 정하는 방법처럼 여러가지가 있다.

1. ***Small!***

함수는 작아야한다.

함수는 150줄, 20줄도 길고, **3~4줄**이면 충분하다. 이러한 함수를 설득력 있는 순서로 짜야하는 것이다.

if,else,while안에 있는 block은 한줄이어야 하고 그 한줄은 **함수 호출**이어야한다.

2. ***Do One Thing***

함수는 한가지 일을 잘해야한다. 무조건 **한가지 일**만 해야한다.

어떠한 함수는 abstraction의 여러 단계를 담고 있을 수 있고 이는 바람직하지 않은 것이다.

한가지 일만을 하는 함수를 명확히 정의하자면, 그 함수에서 다른 기능을 구현한 다른 함수를 추출할 수 없을 때의 상태이다.

3. ***One level of Abstracion per function***

함수는 mixing levels of abstraction이면 혼란을 야기한다. 함수안에 있는 statement들은 **same level of abstraction**이어야한다.

4. ***Reading Code From Top to Bottom: The Stepdown Rule***

각각의 함수가 일정한 level의 추상화를 유지하는 것 뿐만 아니라,

프로그램을 다음과 같이 **set of TO paragraph** 로 읽을 수 있도록 코드를 짜야한다.

To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.

To include the setups, we include the suite setup if this is a suite, then we include the regular setup.

To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.

To search the parent. . .

5. ***Switch Statements***

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

이 코드의 가장 큰 문제점은 same한 structure를 가진 unlimited number of functions가 있다는 것이다.

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

이런식으로 **switch statement를 써서 polymorphic objects**를 만들어낼 수 있다. 시스템의 나머지 부분에서는 이를 볼 수 없다는 특징이 있다.

또한, 모든 circumstance는 **unique**하다.

6. ***Use Descriptive Names***

함수가 더 작고 더 집중될 수록 이름을 길게 설명해서 쓰는 것이 좋다.

할 수 있는 한 **가장 묘사적인 이름**을 쓰는 것이 좋다. 예를 들어 SetupAndTeardownPages 이런식으로 descriptive name을 쓰면 module의 design이 정확해진다.

7. ***Function Arguments***

함수의 인자 수는 **적을수록** 좋다.

- Common Monadic Forms

하나의 인자만을 전달하는 이유는, 해당 인수에 대한 질문을 다른 것으로 변형하여 반환할 수 있다.

- Flag arguments

boolean을 함수로 전달하는 것은 정말 좋지 않은 방법이다.

- Dyadic function

단항함수보다 이해하기 어렵다. 주로 monadic forms로 바꿀 수 있다.

- Triads

세개의 인수는 dyad보다 이해하기 어렵기에 만들기 전에 신중하게 고려해야한다.

- Argument Objects

만약 두개, 세개 이상의 argument가 쓰일때 이러한 argument들은 class로 wrapping되어야 하는 경우가 있다.

Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);

→ 예를 들어 이런 함수에서 variables의 group은 함께 전달되어야 한다.

- verbs and Keywords

함수와 인수는 좋은 verb/noun쌍을 형성해야한다.

8. ***command Query seperation***

command를 query와 구분해야지 모호하거나 readability가 감소하는 상황을 막을 수 있다.

9. ***Prefer Exceptions to Returning Error Codes***

error handling을 구분하기 위해서 throws **Exception** code를 쓰는 것이 더 좋다.
Try/Catch block은 최대한 제거해야한다.

10. ***Don’t Repeat!***

duplication은 evil이다.

앞선 ch2와 이번 ch3의 rule들을 통해 코드의 이름들은 좀 더 잘 지어지고, 기능들이 짧아지며 잘 조직될 것이다.