Clean Code

Robert C.Martin

0. 들어가면서

장인 정신을 익히는 과정은 두 단계로 나뉜다. 바로 이론과 실전이다. 첫째, 장인에게 필요한 원칙, 패턴, 기법, 경험이라는 지식을 습득해야 한다. 둘째, 열심히 일하고 연습해 지식을 몸과 마음으로 체득해야 한다.

1. 깨끗한 코드

코드가 존재하리라

- 어느 수준에 이르면 코드의 도움 없이 요구사항을 상 세하게 표현하기란 불가능하다. 추상화도 불가능하다. 정확히 명시하는 수밖에 없다. 기계가 실행할 정도로 상세하게 요구사항을 명시하는 작업, 바로 이것이 프로그래밍이다. 이렇게 명시한 결과가 바로 코드다.
- 궁극적으로 코드는 요구사항을 표현하는 언어라는 사실을 명심한다. 요구사 항에 더욱 가까운 언어를 만들 수도 있고, 요구사항에서 정형 구조를 뽑아내는 도구를 만들 수도 있다. 하지만 어느 순간에는 정밀한 표현이 필요하다. 그 필요 성을 없앨 방법은 없다. 그러므로 코드도 항상 존재하리라.

나쁜 코드

- 우리 모두는 자신이 짠 쓰레기 코드를 쳐다보며 나중에 손보겠다고 생각한 경험이 있다. 우리 모두는 대충 짠 프로그램이 돌아간다는 사실에 안도감을 느끼며 그래도 안 돌아가는 프로그램보다 돌아가는 쓰레기가 좋다고 스스로를 위로한 경험이 있다. 다시 돌아와 나중에 정리하겠다고 다짐했었다. 물론 그때 그 시절 우리는 르블랑의 법칙(leblanc'sLaw)을 몰랐다. 나중은 결코 오지 않는다.

나쁜 코드로 치르는 대가

- 나쁜 코드는 개발 속도를 크게 떨어뜨린다.
- 나쁜 코드가 쌓일수록 팀 생산성은 떨어진다.

원대한 재설계의 꿈

- 한계에 이르러 '원대한 재설계'를 시작한다 → 유능한 인력으로 구성된 새로운 프로젝트 팀과 나머지 현재 시스템 유지보수팀이 경주를 시작한다 → 기존 시스템을 대체해야 하므로 변경을 모두 따라잡아야 한다 → 따라잡을 즈음엔 새로운 팀원들이 새 시스템을 설계하자고 한다. 현재시스템이 엉망이니까..

태도

- 사용자는 요구사항을 내놓으며 우리에게 현실성을 자문한다. 프로젝트 관리자는 일정을 잡으며 우리에게 도움을 청한다. 우리는 프로젝트를 계획하는 과정에 깊숙히 관여한다. 그러므로 프로젝트 실패는 우리에게도 커다란 책임이 있다. 특히 나쁜 코드가 초래하는 실패에는 더더욱 책임이 크다.
- 일정에 쫓기더라도 대다수 관리자는 좋은 코드를 원한다. 그들이 일정과 요구사항을 강력하게 밀어붙이는 이유는 그것이 그들의 책임이기 때문이다. 좋은 코드를 사수하는 일은 바로 우리 프로그래머들의 책임이다.

- 나쁜 코드의 위험을 이해하지 못하는 관리자 말을 그대로 따르는 행동은 전문가답지 못하다.

원초적 난제

- 나쁜 코드를 양산하면 기한을 맞추지 못한다. 오히려 엉망진창인 상태로 인해 속도가 곧바로 늦어 지고, 결국 기한을 놓친다. 기한을 맞추는 유일한 방법은, 그러니까 빨리 가는 유일한 방법은, 언제나 코드를 최대한 깨끗하게 유지하는 습관이다.

깨끗한 코드라는 예술?

- '코드 감각'이 있으면 좋은 코드와 나쁜 코드를 구분한다. 그뿐만이 아니다. 절제와 규율을 적용해 나쁜 코드를 좋은 코드로 바꾸는 전략도 파악한다.
- 깨끗한 코드를 작성하는 프로그래머는 빈 캔퍼스를 우아한 작품으로 바꿔가는 화가와 같다.

깨끗한 코드란?

> **비야네 스트롭스트룹 Bjarne Stroustrup** *C++ 창시자이자 The C ++ Programming Language 저자*
>
> 나는 우아하고 효율적인 코드를 좋아한다. 논리가 간단해야 버그가 숨어들지 못한다. 의존성을 최대한 줄여야 유지보수가 쉬워진다. 오류는 명백한 전략에 의거해 철저히 처리한다. 성능을 최적으로 유지해야 사람들이 원칙 없는 최적화로 코드를 망치려는 유혹에 빠지지 않는다. 깨끗한 코드는 한 가지를 제대로 한다.

- 깨끗한 코드는 보는 사람에게 즐거움을 선사해야 한다
- 나쁜 코드는 나쁜코드를 유혹한다

  → '실용주의 프로그래머' 의 '깨진 창문' ; 일단 창문이 깨지고 나면 쇠퇴하는 과정이 시작된다.
- 깨끗한 코드는 세세한 사항까지 꼼꼼하게 처리하는 코드다.
- 나쁜 코드는 너무 많은 일을 하려 애쓰다가 의도가 뒤섞이고 목적이 흐려진다. 깨끗한 코드는 한 가지에 '집중'한다.

> **그래디 부치 Grady Booch** *Object Oriented Analysis and Design with Application 저자*
>
> 깨끗한 코드는 단순하고 직접적이다. 깨끗한 코드는 잘 쓴 문장처럼 읽힌다. 깨끗한 코드는 결코 설계자의 의도를 숨기지 않는다. 오히려 명쾌한 추상화와 단순한 제어문으로 가득하다.

> **'큰' 데이브 토마스 big Dave Thomas** *OTI 창립자이자 이클립스 전략의 대부*
>
> 깨끗한 코드는 작성자가 아닌 사람도 읽기 쉽고 고치기 쉽다. 단위 테스트 케이스와 인수 테스트 케이스가 존재한다. 깨끗한 코드에는 의미 있는 이름이 붙는다. 특정 목적을 달성하는 방법은 (여러 가지가 아니라) 하나만 제공한다. 의존성은 최소이며 각 의존성을 명확히 정의한다. API 는 명확하며 최소로 줄였다. 언어에 따라 필요한 모든 정보를 코드만으로 명확히 표현할 수 없기에 코드는 문학적으로 표현해야 마땅하다.

- 작을수록 좋다.
- 인간이 읽기 좋은 코드를 작성하라.

> **마이클 페더스 Michael Feathers** *Working Effectively with Legacy Code 저자*
>
> 깨끗한 코드의 특징은 많지만 그 중에서도 모두를 아우르는 특징이 하나 있다. 깨끗한 코드는 언제나 누군가 주의 깊게 짰다는 느낌을 준다. 고치려고 살펴봐도 딱히 손 댈 곳이 없다. 작성자가 이미 모든 사항을 고려했으므로. 고칠 궁리를 하다보면 언제나 제자리로 돌아온다. 그리고는 누군가 남겨준 코드, 누군가 주의 깊게 짜놓은 작품에 감사를 느낀다.

- 깨끗한 코드는 주의 깊게 작성한 코드다. 누군가 시간을 들여 깔끔하고 단정하게 정리한 코드다. 세세한 사항까지 꼼꼼하게 신경 쓴 코드다. 주의를 기울인 코드다.

> **론 제프리스 Ron Jeffries** *Extreme Programming Installed와 Extreme Programming Adventure in C# 저자*

> 최근 들어 나는 켄트 벡이 제안한 단순한 코드 규칙으로 구현을 시작한다. (그리고 같은 규칙으로 구현을 거의 끝낸다.) 중요한 순으로 나열하자면 간단한 코드는
>
> - 모든 테스트를 통과한다.
> - 중복이 없다.
> - 시스템 내 모든 설계 아이디어를 표현한다.
> - 클래스, 메서드, 함수 등을 최대한 줄인다.
>
> 물론 나는 주로 중복에 집중한다. 같은 작업을 여러 차례 반복한다면 코드가 아이디어를 제대로 표현하지 못한다는 증거다. 나는 문제의 아이디어를 찾아내 좀 더 명확하게 표현하려 애쓴다.
> (...생략...)
> 중복 줄이기, 표현력 높이기, 초반부터 간단한 추상화 고려하기. 내게는 이 세 가지가 깨끗한 코드를 만드는 비결이다.

> **워드 커닝햄 Ward Cunningham** *Wiki 창시자, Fit 창시자, eXtreme Programming 공동 창시자, 디자인 패턴을 뒤에서 움직이는 전문가, Smalltalk와 객체 지향의 정신적 지도자, 코드를 사랑하는 프로그래머들의 대부*
>
> 코드를 읽으면서 짐작했던 기능을 각 루틴이 그대로 수행한다면 깨끗한 코드라 불러도 되겠다. 코드가 그 문제를 풀기 위한 언어처럼 보인다면 아름다운 코드라 불러도 되겠다.

- 프로그램을 단순하게 보이도록 만드는 열쇠는 언어가 아니다. 언어를 단순하게 보이도록 만드는 열쇠는 프로그래머다.

우리는 저자다

- 새 코드를 짜면서 우리는 끊임없이 기존 코드를 읽는다. 비율이 이렇게 높으므로 읽기 쉬운 코드가 매우 중요하다. 비록 읽기 쉬운 코드를 짜기가 쉽지는 않더라도 말이다. 하지만 기존 코드를 읽어야 새 코드를 짜므로 읽기 쉽게 만들면 사실은 짜기도 쉬워진다.
- 주변 코드를 읽기가 어려우면 새 코드를 짜기도 어렵다. 그러므로 급하다면, 서둘러 끝내려면, 쉽게 짜려면, 읽기 쉽게 만들면 된다.

보이스카우트 규칙

> 캠프장은 처음 왔을 때보다 더 깨끗하게 해놓고 떠나라.

- 체크아웃할 때보다 좀 더 깨끗한 코드를 체크인한다면 코드는 절대 나빠지지 않는다.

2. 의미있는 이름

의도를 분명히 밝혀라

- 좋은 이름을 지으려면 시간이 걸리지만 좋은 이름으로 절약하는 시간이 훨씬 더 많다. 그러므로 이름을 주의 깊게 살펴 더 나은 이름이 떠오르면 개선하기 바란다. 그러면 (자신을 포함해) 코드를 읽는 사람이 좀 더 행복해지리라.
- 따로 주석이 필요하다면 의도를 분명히 드러내지 못했다는 말이다.

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

- 문제는 코드의 단순성이 아니라 코드의 함축성이다. 다시 말해, 코드 맥락이 코드 자체에 명시적으로 드러나지 않는다.
- 각 개념에 이름만 붙여도 코드가 상당히 나아진다.

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[0] == 4)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

그릇된 정보를 피하라

- 프로그래머는 코드에 그릇된 단서를 남겨서는 안된다. 그릇된 단서는 코드 의미를 흐린다. 나름대로 널리 쓰이는 의미가 있는 단어를 다른 의미로 사용해도 안된다.
- 서로 흡사한 이름을 사용하지 않도록 주의한다.
- 유사한 개념은 유사한 표기법을 사용한다. 이것도 **정보**다. 일관성이 떨어지는 표기법은 **그릇된 정보**다.

의미 있게 구분하라

- 이름이 달라야 한다면 의미도 달라져야 한다.
- 의미가 불분명한 불용어(noise word) 를 피하라
    - e.g. ProductInfo, ProductData

발음하기 쉬운 이름을 사용하라

- genymdhms vs generationTimeStamp

검색하기 쉬운 이름을 사용하라

- **이름 길이는 범위 크기에 비례해야 한다.** 변수나 상수를 여러 곳에서 사용한다면 검색하기 쉬운 이름이 바람직하다.

인코딩을 피하라

- 헝가리식 표기법
- 멤버 변수 접두어
- 인터페이스 클래스와 구현 클래스: 인터페이스 이름보다 구현클래스 이름을 인코딩 하는 것이 낫다(주의를 흐트리고 과도한 정보를 제공한다)
    - IShapeFactory - ShapeFactory → ShapeFactory(interface) - ShapeFactoryImpl

자신의 기억력을 자랑하지 마라

- 똑똑한 프로그래머는 자신의 정신적 능력을 과시하고 싶어한다.
- 전문가 프로그래머는 명료함이 최고라는 사실을 이해하고, 자신의 능력을 사용해 남들이 이해하는 코드를 내놓는다.

클래스 이름

- 명사, 명사구
- Manager, Processor, Data, Info 등은 피하고, 동사는 사용하지 않는다.

메서드 이름

- 동사, 동사구
- 접근자, 변경자, 조건자는 앞에 get, set, is
- 생성자 중복정의할 때는 정적 팩토리 메서드를 사용한다. 메서드는 인수를 설명하는 이름을 사용한다

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);  // 더 낫다
Complex fulcrumPoint = new Complex(23.0);
```

기발한 이름은 피하라

- 재미난 이름보다 명료한 이름을 선택하라.
- 의도를 분영하고 솔직하게 표현하라.

한 개념에 한 단어를 사용하라

- 추상적인 개념 하나에 단어 하나를 선택해 이를 고수한다.
- 일관성 있는 어휘는 코드를 사용할 프로그래머가 반갑게 여길 선물이다.

말장난을 하지 마라

- 한 단어를 두가지 목적으로 사용하지 마라. 다른 개념에 같은 단어를 사용한다면 그것은 말장난에 불과하다.

해법 영역에서 가져온 이름을 사용하라

- 코드를 읽을 사람도 프로그래머이므로, 전산 용어, 알고리즘명, 패턴명, 수학 용어 사용도 괜찮다.
- 모든 이름을 문제 영역(domain)에서 가져오면, 같은 개념을 다른 이름으로 이해하던 동료들이 매번 고객에게 의미를 물어야 한다.
- 기술 개념에는 기술 이름이 가장 적합하다.

문제 영역에서 가져온 이름을 사용하라

- 적절한 '프로그래머 용어'가 없다면 문제 영역에서 이름을 가져온다.
- 문제 영역 개념과 관련이 깊은 코드라면 문제 영역에서 이름을 가져와야 한다.

의미있는 맥락을 추가하라

- 접두어를 추가해서 맥락을 분명하게 할 수 있다.
    - addrFirstName, addrLastName, addrState, addrCity - 변수가 좀 더 큰 구조에 속한다는 사실이 독자에게 분명해진다.
    - Address 클래스 생성하면 → 변수가 좀 더 큰 개념에 속한다는 사실이 컴파일러에게도 분명해진다.

불필요한 맥락을 없애라

- 일반적으로 짧은 이름이 긴 이름보다 좋다. 단, 의미가 분명한 경우에. 이름에 불필요한 맥락을 추가하지 않도록 주의한다.

3. 함수

의도를 분명히 표현하는 함수를 어떻게 구현할 수 있을까? 함수에 어떤 속성을 부여해야 처음 읽는 사람이 프로그램 내부를 직관적으로 파악할 수 있을까?

작게 만들어라!

- 함수를 만드는 첫째 규칙은 '작게'다. 둘째 규칙은 '더 작게' 다.

블록과 들여쓰기

- if 문 / else 문 / while 문 등에 들어가는 블록은 한 줄이어야 한다. 그러면 함수(enclosing function)가 작아질 뿐 아니라, 블록 안에서 호출하는 함수명이 적절하다면 코드를 이해하기도 쉬워진다.
- 즉, 중첩 구조가 생길만큼 함수가 커져서는 안된다는 뜻이다. 그러므로 함수에서 들여쓰기 수준은 1단이나 2단을 넘어서는 안된다.

한가지만 해라!

> 함수는 한 가지를 해야한다. 그 한 가지를 잘 해야 한다. 그 한 가지만 해야 한다.

- 지정된 함수 이름 아래에서 추상화 수준이 하나인 단계만 수행한다면 그 함수는 한가지 작업만 한다.
- 단순히 다른 표현이 아니라 의미 있는 이름으로 다른 함수를 추출할 수 있다면 그 함수는 여러 작업을 하는 셈이다.

함수 내 섹션

- 한 가지 작업만 하는 함수는 자연스럽게 섹션으로 나누기 어렵다.

함수 당 추상화 수준은 하나로!

- 함수 내 추상화 수준을 섞으면 코드를 읽는 사람이 헷갈린다. 특정 표현이 근본 개념인지 세부사항인지 구분하기 어려운 탓이다. 근본 개념과 세부사항이 섞이기 시작하면, 깨진 창문처럼 함수에 세부사항이 점점 더 추가된다.

위에서 아래로 코드 읽기: 내려가기 규칙

- 코드는 위에서 아래로 이야기처럼 읽혀야 한다. 한 함수 다음에는 추상화 수준이 한 단계 낮은 함수가 온다. 즉, 위에서 아래로 읽으면 함수 추상화 수준이 한 단계씩 낮아진다. → **내려가기** 규칙
- 각 함수는 다음 함수를 소개하고, 각 함수는 일정한 추상화 수준을 유지한다.

Switch 문

- switch 문을 저차원 클래스에 숨기고 반복하지 않는다. 다형성을 이용한다.

문제:

```
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);
    }
}
```

- 함수가 길다. 새 유형을 추가하면 더 길어진다.
- '한 가지' 작업만 수행하지 않는다.
- SRP 를 위반한다. 코드 변경 이유가 여럿.
- OCP 를 위반한다. 새 유형 추가할 때마다 코드 변경.
- 동일한 구조의 함수가 무한정 존재한다.

해결:

- switch 문을 추상 팩토리(abstract factory)에 숨기고, factory 에서 switch 문으로 파생 클래스의 인스턴스를 생성한다.

```
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);
        }
    }
}
```

서술적인 이름을 사용하라!

- 함수가 작고 단순할수록 서술적인 이름을 고르기도 쉬워진다.
- 서술적인 이름을 사용하면 개발자의 머릿속에서도 설계가 뚜렷해지므로 코드를 개선하기 쉬워진다.
- 일관성 있게, 모듈 내에서 함수 이름은 같은 문구, 명사, 동사를 사용한다.

함수 인수

- 이상적인 인수 개수는 0개. 다음이 단항, 이항. 삼항은 가능한 피하는게 좋다. 4개 이상은 특별한 이유가 필요하다. 이유가 있어도 사용하면 안된다. → 인수는 개념을 이해하기 어렵게 만든다.
- 테스트 관점에서 보면 인수는 더 어렵다. 인수 조합으로 함수를 검증.
- 출력 인수는 입력 인수보다 이해하기 어렵다.
- 최선은 인수가 없는 경우이며, 차선은 입력 인수가 1개 뿐인 경우이다.

많이 쓰는 단항 형식

- 인수에 질문을 던지는 경우. e.g. boolean fileExists("MyFile")
- 인수를 뭔가로 변환해 결과를 반환하는 경우. e.g. InputStream fileOpen("MyFile")
- 이벤트 함수
- 그 외는 피한다.

플래그 인수

- 추하다. 함수가 여러 가지를 한다는 의미이므로.

이항 함수

- 단항보다 이해하기 어렵다. 순서를 기억해야 한다. 가능하면 단항 함수로 바꾸도록 애써야 한다.

삼항 함수

- 순서, 주춤, 무시 x2

인수 객체

- 객체를 생성해 인수를 줄인다. 객체가 개념을 표현한다.

인수 목록

- 가변 인수

동사와 키워드

- 단항 함수는 함수와 인수가 동사/명사 쌍을 이뤄야 한다. e.g. writeField(name)
- 함수 이름에 키워드를 추가한다.
    - assertEquals → assertExpectedEqualsActual(expected, actual) ; 인수 순서를 기억할 필요가 없다.

부수 효과를 일으키지 마라!

- 시간적인 결합(temporal coupling)이나 순서 종속성(order dependency) 을 초래한다.
- 부수 효과로 숨겨진 경우 혼란이 커지므로, 만약 시간적인 결합이 필요하다면 함수 이름에 분명히 명시한다.

출력 인수

- 출력 인수는 피해야 한다. 함수에서 상태를 변경해야 한다면 함수가 속한 객체 상태를 변경하는 방식을 택한다.

명령과 조회를 분리하라!

- 함수는 뭔가를 수행하거나 뭔가에 답하거나 둘 중 하나만 해야 한다.

오류 코드보다 예외를 사용하라!

- 명령 함수에서 오류 코드를 반환하는 방식은 명령/조회 분리 규칙을 미묘하게 위반한다. 오류 코드를 처리하기 위해 중첩되는 코드를 야기한다.

```
if (deletePage(page) == E_OK) {
    ....
} else {
    ....
}
```

- 오류 코드 대신 예외를 사용하면 오류 처리 코드가 원래 코드에서 분리되므로 코드가 깔끔해진다.

```
try {
    deletePage(page);
} catch (Exception e) {
    ....
}
```

Try/Catch 블록 뽑아내기

- 코드 구조에 혼란을 일으키며, 정상 동작과 오류 처리 동작을 뒤섞는다. 별도 함수로 뽑아내는 편이 좋다. 코드를 이해하고 수정하기 쉬워진다.

오류 처리도 한 가지 작업이다.

- 오류 처리도 '한 가지' 작업에 속한다. 오류 처리 함수는 오류만 처리해야 한다.

Error.java 의존성 자석

- 오류 코드를 정의한 Error enum 은 의존성 자석(magnet) 이다. Error enum 이 변하면, Error enum 을 사용하는 클래스 전부를 다시 컴파일하고 재배치 해야 한다.
- 오류 코드 대신 예외를 사용하면 새 예외는 Exception 클래스에서 파생된다. 따라서 재컴파일/재배치 없이도 새 예외 클래스를 추가할 수 있다.

반복하지 마라!

- DRY
- 많은 원칙과 기법이 중복을 없애거나 제어할 목적으로 나왔다. RDB 의 정규 형식, 객체 지향 프로그래밍에서 상속, 구조적 프로그래밍, AOP, COP 등.

구조적 프로그래밍

- 에츠허르 데이크스트라(Edsger Dijkstra). 모든 함수와 함수 내 모든 블럭에 입구와 출구가 하나만 존재해야 한다. 즉, 리턴 문은 하나여야 한다. 루프안에 break, continue 를 사용해선 안되며, goto 는 절대 안된다.
- 구조적 프로그래밍은 함수가 아주 클 때만 상당한 이익을 제공한다.

결론

- 대가 프로그래머는 시스템을 (구현할) 프로그램이 아니라 (풀어갈) 이야기로 여긴다. 프로그래밍 언어라는 수단을 사용해 좀 더 풍부하고 좀 더 표현력이 강한 언어를 만들어 이야기를 풀어간다.
- 작성하는 함수가 분명하고 정확한 언어로 깔끔하게 같이 맞아 떨어져야 이야기를 풀어가기가 쉬워진다.

4. 주석

> 나쁜 코드에 주석을 달지마라. 새로 짜라.
>
> - 브라이언 W.커니핸, P.J. 플라우거

- 잘 달린 주석은 그 어떤 정보보다 유용하다. 경솔하고 근거없는 주석은 코드를 이해하기 어렵게 만든다. 오래되고 조잡한 주석은 거짓과 잘못된 정보를 퍼뜨려 해악을 미친다.
- 프로그래밍 언어를 치밀하게 사용해 의도를 표현할 능력이 있다면, 주석은 거의 필요하지 않으리라. 아니, 전혀 필요하지 않으리라.
- 코드로 의도를 표현하지 못해, 실패를 만회하기 위해 주석을 사용한다. 주석은 언제나 실패를 의미한다.
- 주석은 항상도 아니고 고의도 아니지만 너무 자주 거짓말을 한다. 오래될수록 코드에서 멀어진다. 프로그래머들이 주석을 유지하고 보수하기란 현실적으로 불가능하니까.
- 부정확한 주석은
  - 아예 없는 주석보다 훨씬 더 나쁘다.
  - 독자를 현혹하고 오도한다.
  - 결코 이뤄지지 않을 기대를 심어준다.
  - 더 이상 지킬 필요가 없는 규칙이나 지켜서는 안되는 규칙을 명시한다.
- 진실은 언제나 한 곳에만 존재한다. 바로 코드다.

주석은 나쁜 코드를 보완하지 못한다

- 표현력이 풍부하고 깔끔하며 주석이 거의 없는 코드가, 복잡하고 어수선하며 주석이 많이 달린 코드보다 훨씬 좋다. 자신이 저지른 난장판을 주석으로 설명하려 애쓰는 대신 그 난장판을 깨끗이 치우는데 시간을 보내라!

코드로 주석을 표현하라!

- 많은 경우 주석으로 달려는 설명을 함수로 만들어 표현해도 충분하다.

```
😨
// 직원에게 복지 혜택을 받을 자격이 있는지 검사한다
if ((employee.flags & HOURLY_FLAG) && (employee.age) > 65)

😀
if (employee.isEligibleForFullBenefits())
```

좋은 주석

정말 좋은 주석은, 주석을 달지 않을 방법을 찾아낸 주석이다!

- 법적인 주석

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// GNU General Public License 버전 2 이상을 따르는 조건으로 배포한다.
```

- 정보를 제공하는 주석

```
// kk:mm:ss EEE, MMM dd, yyyy 형식이다.
Pattern timeMatcher = Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

- 의도를 설명하는 주석
- 의미를 명료하게 밝히는 주석

```
public void testCompareTo() throws Exception {
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
  WikiPagePath ba = PathParser.parse("PageB.PageA");

  assertTrue(a.compareTo(a) == 0);    // a == a
  assertTrue(a.compareTo(b) != 0);    // a != b
  assertTrue(ab.compareTo(ab) == 0);  // ab == ab
  assertTrue(a.compareTo(b) == -1);   // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1);    // b > a
  assertTrue(ab.compareTo(aa) == 1);  // ab > aa
  assertTrue(bb.compareTo(ba) == 1);  // bb > ba
}
```

    - 주석이 올바른지 검증하기 쉽지 않으므로, 더 나은 방법이 없는지 고민하고 정확히 달도록 각별히 주의한다.
- 결과를 경고하는 주석

```
public static SimpleDateFormat makeStandardHttpDateFormat() {
  // SimpleDateFormat은 스레드에 안전하지 못하다.
  // 따라서 각 인스턴스를 독립적으로 생성해야 한다.
  SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
  df.setTimeZone(TimeZone.getTimeZone("GMT"));
  return df;
}
```

- TODO 주석
    - 주기적으로 점검해 없애도 괜찮은 주석은 없애라.
- 중요성을 강조하는 주석

```
String listItemContent = match.group(3).trim();
// 여기서 trim은 정말 중요하다. trim 함수는 문자열에서 시작 공백을 제거한다.
// 문자열에 시작 공백이 있으면 다른 문자열로 인식되기 때문이다.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

- 공개 API 에서 Javadocs

나쁜 주석

- 주절거리는 주석

```
public void loadProperties() {
  try {
    String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
    FileInputStream propertiesStream = new FileInputStream(propertiesPath);
    loadedProperties.load(propertiesStream);
  }
  catch(IOException e) {
    // 속성 파일이 없다면 기본값을 모두 메모리로 읽어 들였다는 의미다.
  }
}
```

    - 주석을 달기로 결정했다면 충분한 시간을 들여 최고의 주석을 달도록 노력한다.
    - 이해가 안되어 다른 모듈까지 뒤져야 하는 주석은 독자와 제대로 소통하지 못하는 주석이다.
- 같은 이야기를 중복하는 주석
    - 주석이 코드보다 더 많은 정보를 제공하지 못한다.

```
// ContainerBase.java (Tomcat)
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline, MBeanRegistration, Serializable {
  /**
```

```
     * 이 컴포넌트의 프로세서 지연값
     */
    protected int backgroundProcessorDelay = -1;

    /**
     * 이 컴포넌트를 지원하기 위한 생명주기 이벤트
     */
    protected LifecycleSupport lifecycle = new LifecycleSupport(this);

    /**
     * 이 컴포넌트를 위한 컨테이너 이벤트 Listener
     */
    protected ArrayList listeners = new ArrayList();

    /**
     * 컨테이너와 관련된 Loader 구현
     */
    protected Loader loader = null;

    /**
     * 컨테이너와 관련된 Logger 구현
     */
    protected Log logger = null;

    /**
     * 관련된 logger 이름
     */
    protected String logName = null;

    /**
     * 컨테이너와 관련된 Manager 구현
     */
    protected Manager manager = null;

    /**
     * 컨테이너와 관련된 Cluster
     */
    protected Cluster cluster = null;

    /**
     * 사람이 읽을 수 있는 컨테이너 이름
     */
    protected String name = null;
```

- 오해할 여지가 있는 주석
- 의무적으로 다는 주석
    - 코드를 복잡하게 만들며, 거짓말을 퍼뜨리고, 혼동과 무질서를 초래한다.

```
/**
 *
 * @param title CD 제목
 * @param author CD 저자
 * @param tracks CD 트랙 숫자
 * @param durationInMinutes CD 길이(단위: 분)
 */
public void addCD(String title, String author, int tracks, int durationInMinutes) {
```

- 이력을 기록하는 주석
  - 버전 관리 시스템에 맡겨라.
- 있으나 마나 한 주석

```
/**
 * 기본 생성자
 */
protected AnnualDateRule() {
}
/**
 * 월 중 일자를 반환한다.
 *
 * @return 월 중 일자
 */
public int getDayOfMonth() {
  return dayOfMonth;
}

...
} catch (Exception e) {
  try {
    response.add(ErrorResponder.makeExceptionString(e));
    response.closeAll();
  } catch(Exception e1) {
    // 이게 뭐야!
  }
}
```

  - 개발자가 주석을 무시하는 습관에 빠진다. 결국 코드가 바뀌면서 거짓말로 변한다.
  - 있으나 마나 한 주석을 달려는 유혹에서 벗어나 코드를 정리하라. 더 낫고, 행복한 프로그래머가 되는 지름길이다.
- 무서운 잡음
  - javadocs 도 때로는 잡음이다.
- 함수나 변수로 표현할 수 있다면 주석을 달지 마라
- 위치를 표시하는 주석

```
// Actions ////////////////////////////////
```

- 닫는 괄호에 다는 주석
  - 닫는 괄호에 주석을 달아야겠다는 생각이 든다면 대신에 함수를 줄이려 시도하자.
- 공로를 돌리거나 저자를 표시하는 주석

```
/* 홍길동이 추가함 */
```

- 주석으로 처리한 코드
  - 소스 코드 관리 시스템이 우리를 대신해 코드를 기억해 준다. 주석으로 처리할 필요가 없다.
- HTML 주석
  - 필요한 경우, 주석에 HTML 태그를 삽입해야할 책임은 프로그래머가 아니라 도구가 져야 한다.
- 전역 정보
  - 주석을 달아야 한다면 근처에 있는 코드만 기술해라. 시스템 전반의 정보를 기술하지 마라.

```
/**
 * 적합성 테스트가 동작하는 포트: 기본값은 <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
```

- 너무 많은 정보
- 모호한 관계
  - 주석과 주석이 설명하는 코드는 둘 사이 관계가 명확해야 한다.

```
/*
 * 모든 픽셀을 담을 만큼 충분한 배열로 시작한다(여기에 필터 바이트를 더한다).
 * 그리고 헤더 정보를 위해 200바이트를 더한다.
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

  - 주석 자체가 설명을 요구한다.
- 함수 헤더
  - 짧고 한 가지만 수행하며 이름을 잘 붙인 함수가 주석으로 헤더를 추가한 함수보다 훨씬 좋다.
- 비공개 코드에서 Javadocs

5. 형식 맞추기

프로그래머라면 형식을 깔끔하게 맞춰 코드를 짜야 한다.

형식을 맞추는 목적

- 코드 형식은 의사 소통의 일환이다. 의사소통은 전문 개발자의 일차적인 의무다.
- 오늘 구현한 기능이 다음 버전에서 바뀔 확률은 매우 높다. 그런데 오늘 구현한 코드의 가독성은 앞으로 바뀔 코드의 품질에 지대한 영향을 미친다. 맨 처음 잡아놓은 구현 스타일과 가독성 수준은 유지보수 용이성과 확장성에 계속 영향을 미친다.

적절한 행 길이를 유지하라

- 소스 길이는 얼마나 길어야 적당할까? 일반적으로 큰 파일보다 작은 파일이 이해하기 쉽다.

신문 기사처럼 작성하라

- 이름은 간단하면서도 설명이 가능하게 짓는다. 이름만 보고도 올바른 모듈을 살펴보고 있는지 아닌지를 판단할 정도로 신경써서 짓는다.
- 소스 파일 첫 부분은 고차원 개념과 알고리즘을 설명한다. 아래로 내려갈수록 의도를 세세하게 묘사한다. 마지막에는 가장 저차원 함수와 세부 내역이 나온다.
- 한 면을 채우는 기사는 거의 없다. 신문을 읽을 만한 이유는 여기에 있다.

개념은 빈 행으로 분리하라

- 각 행은 수식이나 절을 나타내고, 일련의 행 묶음은 완결된 생각 하나를 표현한다. 생각 사이는 빈 행을 넣어 분리해야 마땅하다.

세로 밀집도

- 줄바꿈이 개념을 분리한다면 세로 밀집도는 연관성을 의미한다. 즉, 서로 밀집한 코드 행은 세로로 가까이 놓아야 한다는 뜻이다.

수직 거리

- 서로 밀집한 개념은 세로로 가까이 둬야 한다. 두 개념이 서로 다른 파일에 속한다면 규칙은 통하지 않는다. 하지만 타당한 근거가 없다면 서로 밀접한 개념은 한 파일에 속해야 마땅하다. 이게 바로 protected 변수를 피해야 하는 이유 중 하나다.
- 같은 파일에 속할 정도로 밀접한 두 개념은 세로 거리로 연관성을 표현한다. 여기서 연관성이란 한 개념을 이해하는 데 다른 개념이 중요한 정도다. 연관성이 깊은 두 개념이 멀리 떨어져 있으면 코드를 읽는 사람이 소스 파일과 클래스를 여기저기 뒤지게 된다.
- **변수 선언**
    - 변수는 사용하는 위치에 최대한 가까이 선언한다.
- **인스턴스 변수**
    - 클래스 맨 처음에 선언한다. 변수 간에 세로로 거리를 두지 않는다. 잘 설계된 클래스는 많은(혹은 대다수) 클래스 메서드가 인스턴스 변수를 사용하기 때문이다.
- **종속 함수**
    - 한 함수가 다른 함수를 호출한다면 두 함수는 세로로 가까이 배치한다. 또한 가능하다면 호출하는 함수를 호출되는 함수보다 먼저 배치한다. 그러면 프로그램이 자연스럽게 읽힌다.
- **개념적 유사성**
    - 어떤 코드는 서로 끌어당긴다. 개념적인 친화도가 높기 때문이다. 친화도가 높을수록 코드를 가까이 배치한다.

세로 순서

- 일반적으로 함수 호출 종속성은 아래 방향으로 유지한다. 즉, 호출되는 함수를 나중에 배치한다. 그러면 소스 코드 모듈이 고차원에서 저차원으로 자연스럽게 내려간다.

가로 형식 맞추기

- 프로그래머는 명백하게 짧은 행을 선호한다.
- 120자가 넘으면 솔직히 주의부족이다.

가로 공백과 밀집도

- 가로로는 공백을 사용해 밀접한 개념과 느슨한 개념을 표현한다.
- 할당 연산자는 강조를 위해 앞뒤 공백
- 함수명에 이어지는 괄호 사이는 밀접하므로 붙여서. (공백을 넣으면 별개로 보인다.)
- 연산자 우선 순위 강조를 위해

가로 정렬

```
public class FitNesseExpediter implements ResponseSender {
  private    Socket       socket;
  private    InputStream  input;
  private    OutputStream output;
  private    Request      request;
  private    Response     response;
  protected long          requestParingTimeLimit;
  .....
}
```

- 코드의 엉뚱한 부분을 강조해 진짜 의도가 가려지므로 유용하지 않다.
- 정렬이 필요할 정도로 목록이 길다면 문제는 목록 길이지 정렬 부족이 문제가 아니다.

들여쓰기

- 소스 파일은 윤곽도(outline) 계층과 비슷하다.
  파일 전체, 파일 내 개별 클래스, 클래스 내 각 메서드에 적용되는 정보와, 블록 내 블록에 재귀적으로 적용되는 정보가 있다.
  계층에서 각 수준은 이름을 선언하는 범위이자 선언문과 실행문을 해석하는 범위다.
- 이렇듯 범위(scope)로 이뤄진 계층을 표현하기 위해 우리는 코드를 들여쓴다.
- 프로그래머는 이런 들여쓰기 체계에 크게 의존한다.
- **들여쓰기 무시하기**
  - 한 행의 범위를 뭉뚱그린 코드를 피한다.

```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "....";

  public CommentWidget(ParentWidget parent, String text) { super(parent, text); }
  public String render() throws Exception { return ""; }
}
```

가짜 범위

- 빈 while 이나 for 문은 피해라. 피하지 못할 때는 빈 블록을 올바로 들여쓰고 괄호로 감싼다.
- 세미콜론(;)은 새 행에다 제대로 들여써서 넣어준다. 그렇게 하지 않으면 눈에 띄지 않는다.

```
while (dis.read(buf, 0, readBufferSize) != 1)
;
```

팀 규칙

- 프로그래머라면 각자 선호하는 규칙이 있다. 하지만 **팀에 속한다면 자신이 선호해야 할 규칙은 바로 팀 규칙**이다.
- 좋은 소프트웨어 시스템은 읽기 쉬운 문서로 이뤄진다는 사실을 기억하기 바란다. 스타일이 일관적이고 매끄러워야 한다. 한 소스 파일에서 봤던 형식이 다른 소스 파일에도 쓰이리라는 신뢰감을 독자에게 줘야 한다.

밥 아저씨의 형식 규칙

- 코드 자체가 최고의 구현 표준 문서가 되도록

6. 객체와 자료 구조

자료 추상화

```
// 구현을 노출한다
public class Point {
  public double x;
  public double y;
}
```

```
// 구현을 숨긴다
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
}
```

- 인터페이스는 자료 구조를 명백하게 표현한다. 메서드가 접근 정책을 강제한다.
- 변수 사이에 함수를 넣는다고 구현이 저절로 감춰지지는 않는다. 구현을 감추려면 추상화가 필요하다. 형식 논리에 치우쳐 조회 함수, 설정 함수로 변수를 다룬다고 클래스가 되지는 않는다. 그보다는 **추상 인터페이스를 제공해 사용자가 구현을 모른 채 자료의 핵심을 조작할 수 있어야 진정한 의미의 클래스**다.

```
// 연료 상태를 구체적인 숫자값으로 알려준다. 변수 조회.
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

```
// 연료 상태를 백분률이라는 추상적인 개념으로 알려준다. 정보가 어디서 오는지 드러나지 않는다.
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

- **자료를 세세하게 공개하기보다는 추상적인 개념으로 표현하는 편이 좋다.** 인터페이스나 조회/설정 함수만으로 추상화가 이뤄지지 않는다. 객체가 포함하는 자료를 표현할 가장 좋은 방법을 심각하게 고민해야 한다. 아무 생각 없이 조회/설정 함수를 추가하는 방법이 가장 나쁘다.

자료/객체 비대칭

- **객체는 추상화 뒤로 자료를 숨긴 채 자료를 다루는 함수만 공개한다. 자료구조는 자료를 그대로 공개하며 별다른 함수를 제공하지 않는다.**

```
public class Square {
  public Point topLeft;
  public double side;
}
public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}
public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.1415926535;

  public double area(Object shape) throws NoSuchShapeException {
    if (shape instanceof Square) {
      Square s = (Square) shape;
      return s.side * s.side;
    } else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle) shape;
      return r.height * r.width;
    } else if (shape instanceof Circle) {
      Circle c = (Circle) shape;
      return PI * c.radius * c.radius;
    } else {
```

```
      throw new NoSuchShapeException();
    }
  }
}
```

- 함수를 추가하고 싶다면? 도형 클래스는 아무 영향을 받지 않는다.
- 새 도형을 추가하고 싶다면? Geometry 클래스에 속한 함수를 모두 고쳐야 한다.

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side * side;
  }
}
public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.1415926535;

  public double area() {
    return PI * radius * radius;
  }
}
```

- 새 도형을 추가해도 기존 함수에 아무 영향을 미치지 않는다.
- 새 함수를 추가하고 싶다면 도형 클래스 전부를 고쳐야 한다.

> (자료 구조를 사용하는) 절차적인 코드는 기존 자료 구조를 변경하지 않으면서 새 함수를 추가하기 쉽다. 반면, 객체 지향 코드는 기존 함수를 변경하지 않으면서 새 클래스를 추가하기 쉽다.
>
> 반대로,
>
> 절차적인 코드는 새로운 자료 구조를 추가하기 어렵다. 그러려면 모든 함수를 고쳐야 한다. 객체 지향 코드는 새로운 함수를 추가하기 어렵다. 그러려면 모든 클래스를 고쳐야 한다.

- 새로운 자료 타입이 필요한 시스템은 클래스와 객체 지향 기법이 적합하다. 반면, 새로운 함수가 필요한 경우는 절차적인 코드와 자료 구조가 좀 더 적합하다.
- 모든 것이 객체란 생각은 미신이다. 때로는 단순한 자료 구조와 절차적인 코드가 가장 적합한 상황도 있다.