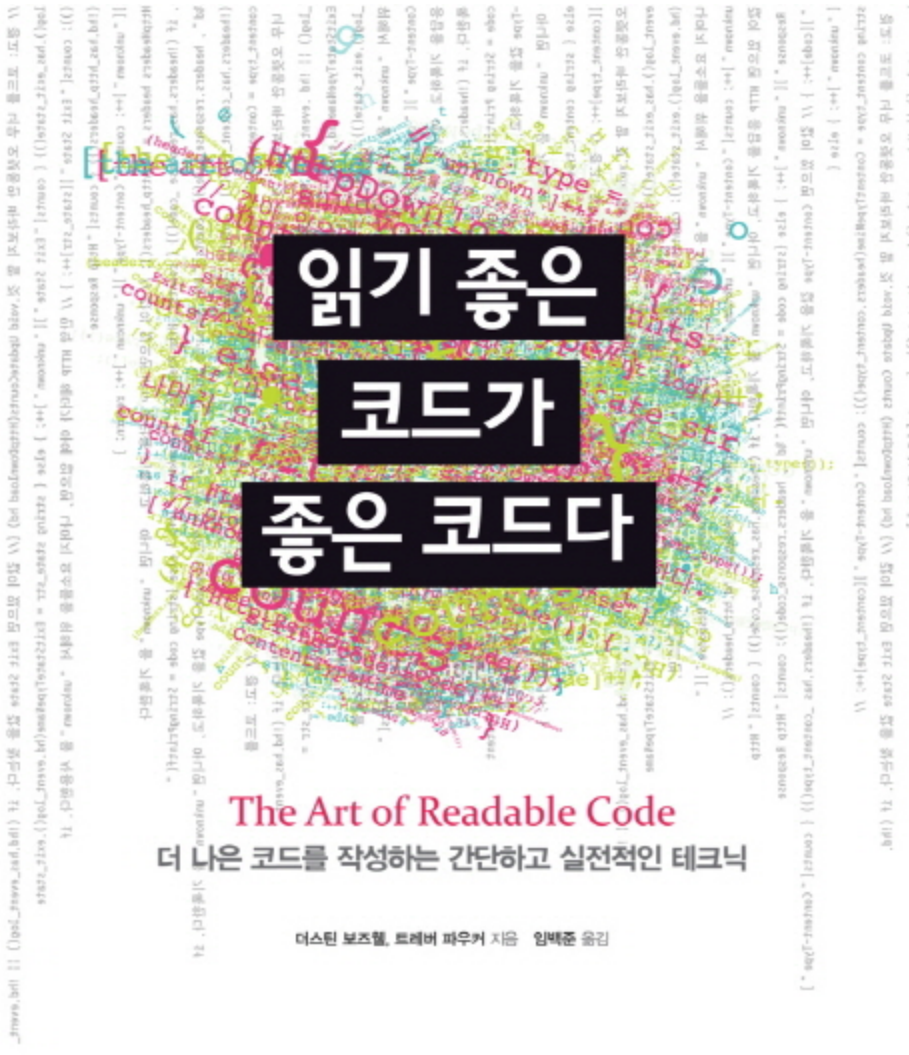
책 소개 읽기 좋은 코드가 좋은 코드다



읽기 좋은 코드를 고민하게 된 계기

- 작년 초 영화서비스 할 때 부터 고민
- 코드리뷰
- 함수명, 변수명의 의미가 모호하다는 피드백
- 로직이 이해하기 힘들다는 피드백
- 코드 작성 방법에 문제가 있다는 것을 인지
- 개선하자!! 읽기 좋은 코드 작성 방법을 학습하자!!

우연히 책을 알게 됨...

- 읽기 좋은 코드에 대한 리서치 중
- 제 코드는 이 책을 읽기 전과 후로 나눕니다.
- 코드리뷰에 평가는 확실히 달라졌음
- 그래서 소개를 하게 되었습니다.

이제 `본론` 으로 들어가겠습니다.

먼저 코드는 왜 이해하기 쉬워야 할까요?

- 우리는 코드를 작성하는 시간보다 **코드를 보고만 있는 시간**을 대부분 차지
- 우리에게는 시간은 유한함
- 때문에 제한시간에 요구사항을 개발하는 게 하나의 목표
- 최소한 미래의 내 자신 또는 다른사람이
    - 코드를 이해하는 데 들이는 시간 최소화

코드를 완전히 이해한다는 것은 무엇을 의미할까요??

이 책에서는 **세가지**로 정의합니다.

- 첫째, 코드를 자유롭게 수정이 가능하다.
- 둘째, 버그를 짚어내는 것이 가능하다.
- 셋째, 수정된 내용이 작성한 다른 부분의 코드와 어떻게 상호작용하는지를 알 수 있어야 한다.

코드를 이해한다는 것은 이 세가지를 할 수 있을 때를 의미합니다.

이 책에서 알려주고 싶은 것

개념들을 모아보면 세가지로 분류를 할 수 있습니다.

1. 네이밍
2. 논리 정렬
3. 하위 문제 추출

1. 먼저 `네이밍` 부터 알아보도록하겠습니다.

- 변수명, 함수명, 클래스명 모든 이름은 **일종의 설명문**입니다.
- 즉, 전달하고 싶은 설명을 이름을 통해 전달합니다.

```
class MovieComponent {
  async render () {
    const movies = await this.fetchMovies()
    document.body.innerHTML =
      movies.map(movie => `<div>${movie}</div>`)
  }
  fetchMovies () {
    return Promise.resolve(['스파이더맨', '아이언맨'])
  }
}
```

1_1. 보편적인 단어를 피하고 구체적인 단어를 선택

- 구체적인 단어를 통해 정확한 의도한 정보를 전달해야 함
- 어디에서 가져오는 지
  - `getPage()` => `fetchPage(), downloadPage()`
- 무엇을 수행하는 지
  - `stop()` => `kill(), resume(), pause()`
- 무엇을 반환하는 지
  - `size()` => `height(), nodesLength(), memoryBytes()`

1_2. 단위를 포함하는 것들

- 시간의 양이나 바이트의 수와 같이 측정치를 포함하면
- 단위를 포함하는 것이 좋습니다.

```
// Not Cool
const start = new Date().getTime()
...
const end = new Date().getTime() - start
console.log(`Load time was: ${end} seconds`) // Wrong!!

// Cool
const startMs = new Date().getTime()
...
const endMs = new Date().getTime() - startMs
console.log(`Load time was: ${endMs / 1000} seconds`)
```

1_3. 다른 중요한 속성 포함하기

- 위험한 요소
- 나중에 잘못 이해했을 때 심각한 버그를 만들 가능성 있는 것들
- 패스워드가 암호화 안되있다면 : `password` => `plainTextPassword`
- URL Encoded 데이터라면 : `data` => `dataURLEnc`
- 이스케이프 처리가 되어야 한다면 : `comment` => `unescapedComment`

2. 이번에는 `논리 정렬` 입니다.

- 논리 정렬은 **읽기 쉽게 흐름제어**를 만드는 것
- 조건, 루프, 흐름을 통제하는 선언문은 코드를 복잡하게 만드는 원인
- 코드를 읽을 때 다시 되돌아가서 코드를 읽지 않아도 되게 만들어야 함

2_1. 조건문에서 인수의 순서

```
// 가독성 좋음
if (length >= 10) {}

// 가독성 낮음
if (10 <= length) {}
```

- 왼쪽 : 질문을 받는 표현
- 오른쪽 : 비교대상
- 이러한 가이드 라인은 영어 어순과 일치한다.

2_2. if/else 블록의 순서

```
// Not Cool
if (a !== b) {
} else {
}

// Cool
if (a === b) {
} else {
}
```

- `if/else`를 사용하는 경우 부정이 아닌 긍정을 먼저 다루는 게 좋다
- 첫번째 블록을 생각할 때 한번 부정한 값을 생각해야 함으로 긍정부터 다루는 게 좋음
- 생각보다 이런 코드가 많음...

2_3. 중첩을 최소화하기

- 코드의 중첩이 심할 수록 코드를 읽는 사람의 마음속에 존재하는 정신적 스택에 추가적인 조건이 입력됨
- 함수 중간에 반환하기로 중첩을 제거 할 수 있음

```
// Not Cool
if (userResult === SUCCESS) {
  if (permissionResult !== SUCCESS) {
    reply.writeError('error reading permissions')
  } else {
    reply.writeError('')
  }
} else {
  reply.writeError(userResult)
}

reply.done()
```

2_3_1. 반환하기로 중첩을 제거

```
// Cool
if (userResult !== SUCCESS) {
  reply.writeError(userResult)
  reply.done()
  return
}

if (permissionResult !== SUCCESS) {
  reply.writeError('error reading permissions')
  reply.done()
  return
}

reply.writeError('')
reply.done()
```

2_4. 거대한 구문 나누기

- 개별적인 표현은 그렇게 크지 않지만, 모두 한곳에 있어 읽기 힘들게 함
- 표현하는 많은 부분이 동일
- 동일한 부분을 **요약 변수로 추출**해서 함수의 앞부분에 놓아둘 수 있음

```
// Not Cool
const updateHighlight = messageNum => {
  if ($(`#vote_value${messageNum}`).html() === "Up") {
    $(`#thumbs_up${messageNum}`).addClass("highlighted");
    $(`#thumbs_down${messageNum}`).removeClass("highlight
  } else if ($(`#vote_value${messageNum}`).html() === "Do
    $(`#thumbs_up${messageNum}`).removeClass("highlighted
    $(`#thumbs_down${messageNum}`).addClass("highlihgted"
  } else {
    $(`#thumbs_up${messageNum}`).removeClass("highlighted
    $(`#thumbs_down${messageNum}`).removeClass("highlight
  }
}
```

2_4_1. 요약 변수로 추출

- 우리 뇌는 자연스럽게 **그룹**과 **계층 구조**를 따라서 동작
- 선언문과 조건을 분리

```
// Cool
const updateHighlight = messageNum => {
  const thumbsUp = $(`#thumbs_up${messageNum}`)
  const thumbsDown = $(`#thumbs_down${messageNum}`)
  const voteValueHtml = $(`#vote_value${messageNum}`).htm
  const ACTIVE_CLASS = "highlighted"

  if (voteValueHtml === "Up") {
    thumbsUp.addClass(ACTIVE_CLASS);
  } else {
    thumbsUp.removeClass(ACTIVE_CLASS);
  }
  if (voteValueHtml === "Down") {
    thumbsDown.addClass(ACTIVE_CLASS);
  } else {
    thumbsDown.removeClass(ACTIVE_CLASS);
  }
}
```

3. 이번에는 `하위 문제 추출` 입니다.

- 사람는 한번에 서너개 일만 생각할 수 있다고 합니다.
- 즉, 코드의 표현이 커지면 이해하기 더 어렵습니다.

하위 문제를 추출하는 방법

- 함수 목적은 무엇인가?
- 함수내의 라인마다, 이 라인은 필요하지만 직접적으로 상관없는 하위문제를 해결하는 가?

함수의 목적과 라인은 직접적으로 관련되지 않으면 하위 문제로 추출합니다.

3_1. 설명 변수

- 커다란 표현을 쪼개는 가장 쉬운 방법은 작은 하위 표현을 담을 추가 변수를 만드는 것
- 하위표현의 의미를 설명하므로 **설명 변수**라고도 함

```
// Not Cool
if (line.split(':')[0] === "root") {}

// Cool
const username = line.split(':')[0]
if (username === "root") {}
```

3_2. 거대한 함수 추출

- 상위수준 목적은 주어진 점과 가장 가까운 장소를 찾는 것
- 코사인의 특별법칙 공식을 사용하는 부분은 하위문제로 추출 가능

```
const findClosestLocation = (lat, lng, array) => {
  let closest
  let closestDist = Number.MAX_VALUE

  for (let i = 0, len = array.length; i < len; i++) {
    const latRad = radians(lat)
    const lngRad = radians(lng)
    const lat2Rad = radians(array[i].latitude)
    const lng2Rad = radians(array[i].longitude)

    const dist = Math.acos(Math.sin(latRad) * Math.sin(la

    if (dist < closestDist) {
      closest = array[i]
      closestDist = dist
    }
  }
  return closest
}
```

3_2_1. 직접 상관없는 하위문제

루프의 내부에 있는 코드는 대부분 주요 목적과 직접 상관없는 하위문제를 다룸

```
const sphericalDistance = (lat1, lng1, lat2, lng2) => {
  const latRad = radians(lat1)
  const lngRad = radians(lng1)
  const lat2Rad = radians(lat2)
  const lng2Rad = radians(lng2)

  return Math.acos(
    Math.sin(latRad) * Math.sin(lat2Rad) +
    Math.cos(latRad) * Math.cos(lat2Rad) *
    Math.cos(lng2Rad - lngRad)
  )
}
```

3_2_2. 하위 문제 추출 결과

이제 원래 코드는 이렇게 변한다.

```
const findClosestLocation = (lat, lng, array) => {
  let closest
  let closestDist = Number.MAX_VALUE

  for (let i = 0, len = array.length; i < len; i++) {
    const dist = sphericalDistance(lat, lng, array[i].lat

    if (dist < closestDist) {
      closest = array[i]
      closestDist = dist
    }
  }
  return closest
}
```

- 밀도 높은 기하 공식에 방해받지 않음
- 상위수준의 목적에 집중
- 전반적으로 코드의 가독성이 높아짐

3_3. 기존의 인터페이스를 단순화하기

- 라이브러리가 깔끔한 인터페이스를 제공하면 누구나 좋아함
- 자신이 사용하는 인터페이스가 깔끔하지 않다면
- `Wrapper`로 보완할 수 있음

3_3_1. 예) 자바스크립트가 브라우저 쿠키

- 자바스크립트가 브라우저 쿠키를 다루는 방식은 이상적이지 않다.
- 개념적으로 보면 **쿠키**는 **이름/값** 짝으로 이루어진다.
- 브라우저가 제공하는 인터페이스는 다음과 같은 문법으로 된 하나의 `document.cookie` 를 사용한다.

```
name1=value1; name2=value2; ...
```

3_3_2. 필요한 쿠키 찾으려면...

필요한 쿠키를 찾으려면 이 거대한 문자열의 구문분석을 직접 수행해야 한다.

다음은 `max_results` 라는 이름을 가진 쿠키의 값을 읽는 코드이다.

```
let maxResults
const cookies = document.cookie.split(';')
for (let i = 0, len = cookies.length; i < len; i++) {
  const cookie = cookies[i].replace(/^[ ]+/, '')
  if (cookie.indexOf('max_results') === 0) {
    maxResults = Number(cookie.substring(12, cookie.lengt
  }
}
```

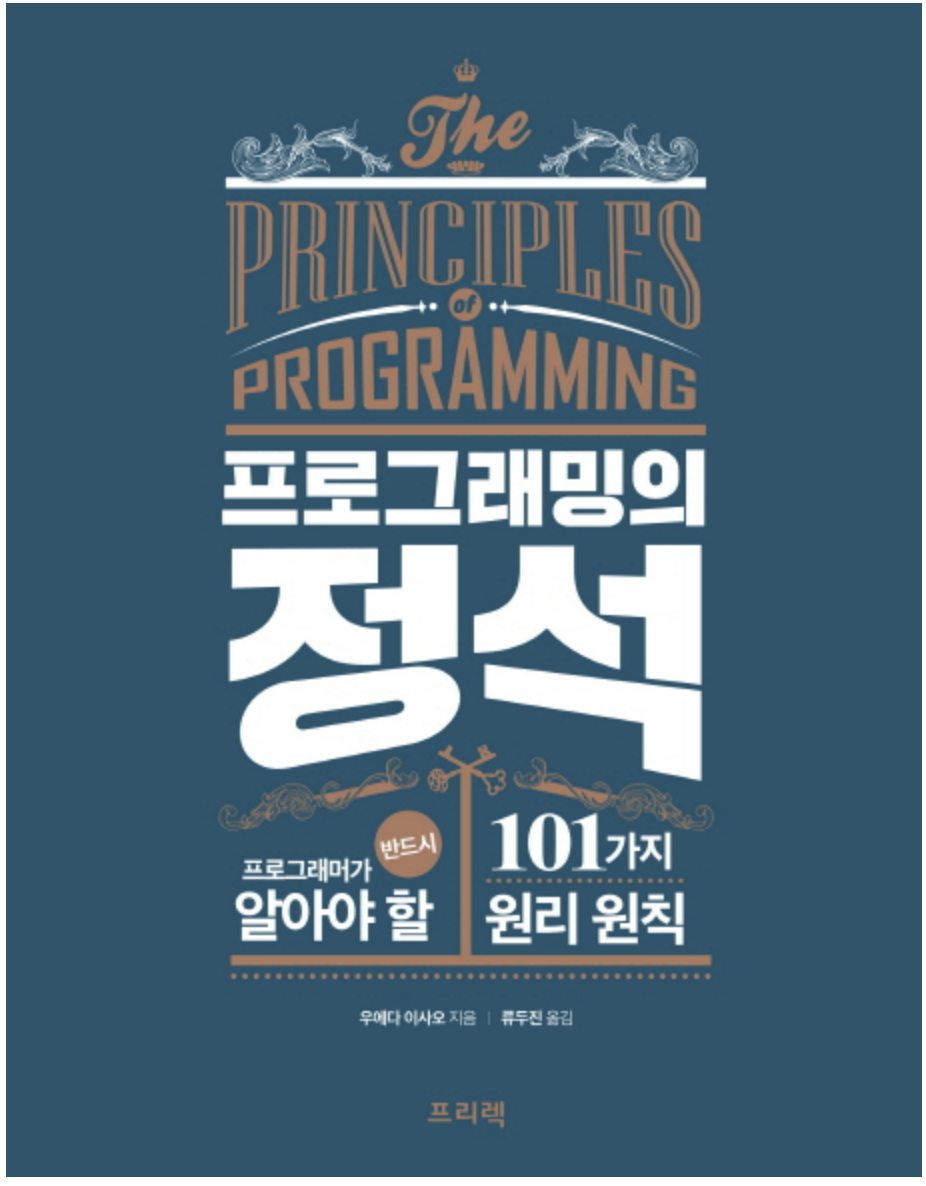
3_3_3. 이상적인 인터페이스 만들기

- 이상적이지 않은 인터페이스를 그냥 받아들일 이유가 없다
- 이런 인터페이스가 있으면 언제나 이를 둘러싸는 함수를 작성하여 지저분한 내부를 감출 수 있다.

```
const getCookie = name => {
  const cookies = document.cookie.split(';')
  for (let i = 0, len = cookies.length; i < len; i++) {
    const cookie = cookies[i].replace(/^[ ]+/, '')
    if (cookie.indexOf(name) === 0) {
      return cookie.substring(name.length + 1, cookie.leng
    }
  }
}
const maxResults = Number(getCookie('max_results'))
```

마지막!

2. 원칙: 프로그래밍의 가이드라인



2.5 SLAP(Single Level of Abstraction Principle)

- 단일 수준 추상화 원칙
- 코드를 작성할 때 **높은 수준**과 **낮은 수준**의 추상화 개념을 분리
- 최고 수준부터 중간 수준의 처리가 책의 **목차**가 되고
- 최저 수준의 처리가 책의 **본문 내용**이 됨
- 결과적으로 추상화 수준을 일치시킨 코드는 훌륭한 책

```
function 고수준() { 중수준1(); 중수준2(); } // 수준1의 목차
function 중수준1() { 저수준1(); 저수준2(); } // 수준2의 목자-1
function 저수준1() { }
function 저수준2() { }
function 중수준2() { 저수준3(); }
function 저수준3() { }
```

2.7 명명이 중요하다(Naming is important)

- 명명 방침 : **이름 가역성**
  - **이름이란 명명의 기반이 된 내용의 설명문을 복원할 수 있어야 한다**
- 이 방침을 충족하려면 **루프백 확인**을 수행해야 한다.
  - 기존은 내용의 설명문 => 이름
  - 반대로 이름 => 설명문
  - `설명->이름->설명` 의 순으로 한 바퀴 돌아서 원래로 돌아왔을 때
    - 설명이 일치하면 좋은 이름
    - 일치하지 않으면 주의가 필요

```
const sum = (a, b) => a + b
```

EoF