실용주의 프로그래머

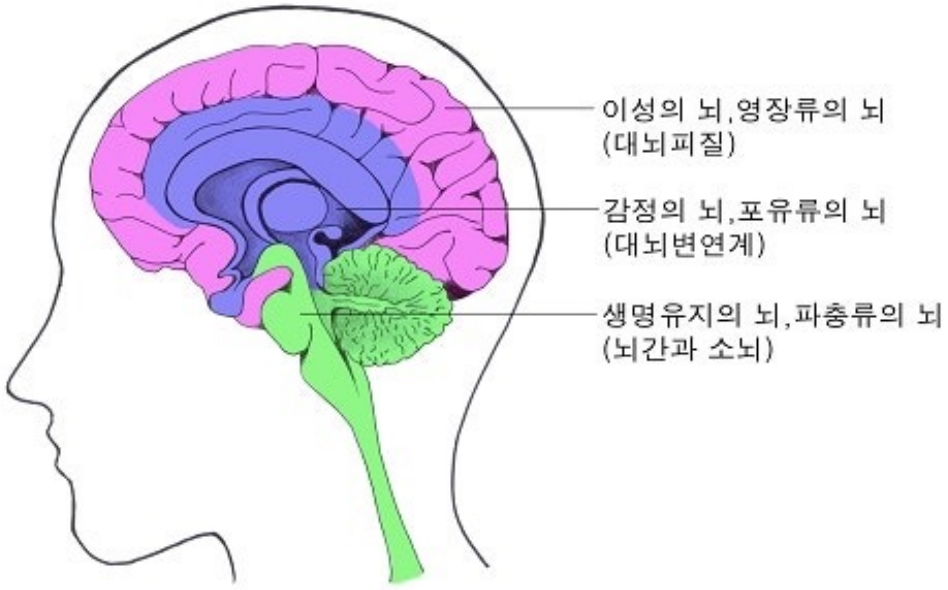
7장. 코딩하는 동안

Topic 37
파충류의 뇌에 귀 기울이기

Tip 61. 여러분 내면의 파충류에게 귀 기울여라

파충류의 뇌?

회의를 할 때 보면 흥분한 사람은 공격적이거나 도망가려고 함.
이성의 뇌가 멈추고 생존의 뇌(파충류의 뇌)가 작동한 것
그러므로 누군가를 설득하려면 영장류의 뇌에 말을 걸어야 한다고 함



코딩하면서 스스로 느끼는 불안감, 초조함 혹은 더뎌지는 작업 속도는
내면의 파충류가 신호를 보내는 것.

우리는 이 신호를 무시하지 말고 내면의 파충류에게 귀 기울여야 함

HOW?

1. 일단 하고 있는 일을 멈추자! 생각이 뇌에 스며들도록
2. 문제를 표면으로 끄집어내자 (코드에 대한 그림을 그리거나, 동료에게 설명)
3. 그런데도 안되면? 프로토타이핑으로 만들자

1. 포스트잇에 '프로토타이핑 중'이라고 모니터 옆에 붙여두기
2. 프로토타입은 원래 실패하고, 성공하더라도 버리는 것이라고 상기
3. 텅 빈 에디터 화면에 하고 싶은 것, 배우고 싶은 것을 주석으로 표현
4. 코딩 시작

요즘에는 ai 덕분에 프로토타이핑에 드는 비용도 작을 듯...

Topic 38
우연에 맡기는 프로그래밍

잘 돌아가네? 타닥타닥 -> 갑자기 안 됨 -> 구석구석 찾아도 안보임
why? 애초에 코드가 왜 잘 돌아가는지도 몰랐기 때문

Tip 62. 우연에 맡기는 프로그래밍을 하지 말라.

우리가 접하는 우연들
구현에서 생기는 우연 -> 의도된 호출 방식이 아니지만 일단 돌아가니까 냅두기
비슷하다는 착각 -> 비슷한 기능이니까 이것만 바꾸고 갖다쓰자 = 스노우볼
유령패턴 -> 그냥 코드의 오류일수도 있는데 패턴과 인과관계를 의심해서 쉐도우 복싱
상황에서 생기는 우연 -> 무의식적으로 이런 상황에서 돌아갈 것이라고 암묵적으로 가정하기

so 문서화된 동작에만 의존하자. 그럴 수 없다면 추측을 문서로 상세히 남기기

의도적으로 프로그래밍 하자

•언제나 내가 지금 무엇을 하고 있는지 알아야 한다.

•더 경험이 적은 프로그래머에게 코드를 상세히 설명할 수 있어야 한다.

•자신도 잘 모르는 코드를 만들지 말라. 우연의 함정에 빠질 가능성이 높다. 왜 동작하는지 모르면 왜 실패하는지도 알 리가 없다.

- 만약 마감기한 내에 잘 사용해보지 못한 라이브러리를 써야 한다면? 어떻게 해야할까...

•계획을 세우고 그것을 바탕으로 진행하라

•신뢰할 수 있는 것에만 기대라. 가정에 의존하지 말라. 무언가 신뢰할 수 있을지 판단하기 어렵다면 일단 최악의 상황을 가정하라

•가정을 기록으로 남겨라.

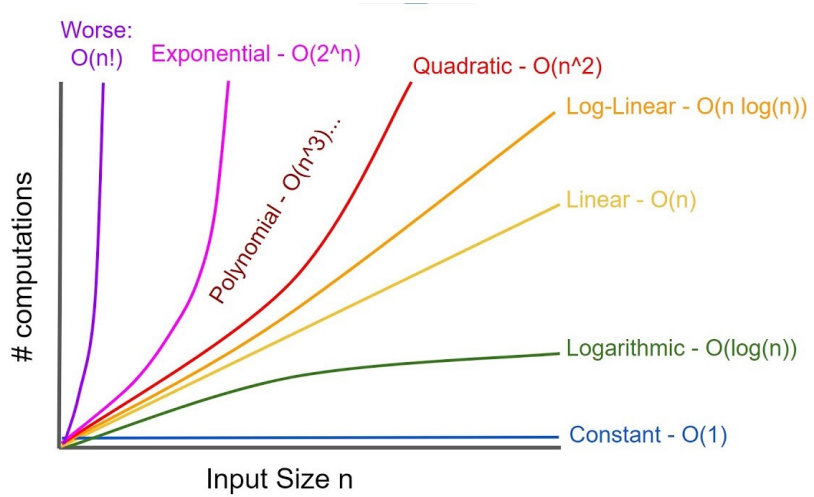
•코드뿐 아니라 우리가 세운 가정도 테스트해 보아야 한다. 단정문을 작성하라

•노력을 기울일 대상의 우선순위를 정하라.

•과거의 노예가 되지 말라. 기존 코드가 앞으로 짤 코드를 지배하도록 놓아두지 말라. 언제나 리팩터링할 자세가 되어 있어야 한다.

Topic 39
알고리즘의 속도

Tip 63. 사용하는 알고리즘의 차수를 추정하라.



- Guess a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$ and verify your answer using the substitution method.

① 트리 생각하기

$T(n) = T(n/2) + n^2$

$T(n/2) = T(n/4) + (\frac{n}{2})^2$

$T(n/4) = T(n/8) + (\frac{n}{4})^2$

$T(n/8) = T(n/16) + (\frac{n}{8})^2$

⋮

$T(1) = \Theta(1)$

② recursion tree 연산 총 비용은

$n^2 + \frac{n^2}{4} + \frac{n^2}{16} + \frac{n^2}{256} + \cdots$

$= (1 + \frac{1}{4} + \frac{1}{4^2} + \frac{1}{4^3} + \cdots) n^2$

↓ 등비

$\leq (\frac{1}{1-\frac{1}{4}}) n^2 = O(n^2)$ //

③ 치환법

$T(n) \leq \frac{cn^2}{4} + n^2$

$= (\frac{c}{4} + 1) n^2 \leq cn^2$

$\rightarrow \frac{c}{4} + 1 \leq c$

$\rightarrow 1 \leq \frac{3}{4}c$

$\rightarrow \frac{4}{3} \leq c$ //

입력 데이터 집합이 작을 때는 수행시간이 선형적으로 늘어나다가도,
수백만 개의 레코드를 입력하면 스레싱이 발생해 수행시간이 폭증하기도 한다.

Tip 64. 여러분의 추정을 테스트하라.

코드 프로파일러를 사용해 알고리즘이 돌아갈 때 각 단계의 실행 횟수를 세고
입력값 크기별 실행 횟수를 그래프로 그려보자

프론트엔드 개발의 특성상 알고리즘을 써야하는 경우가 굉장히 드물다.

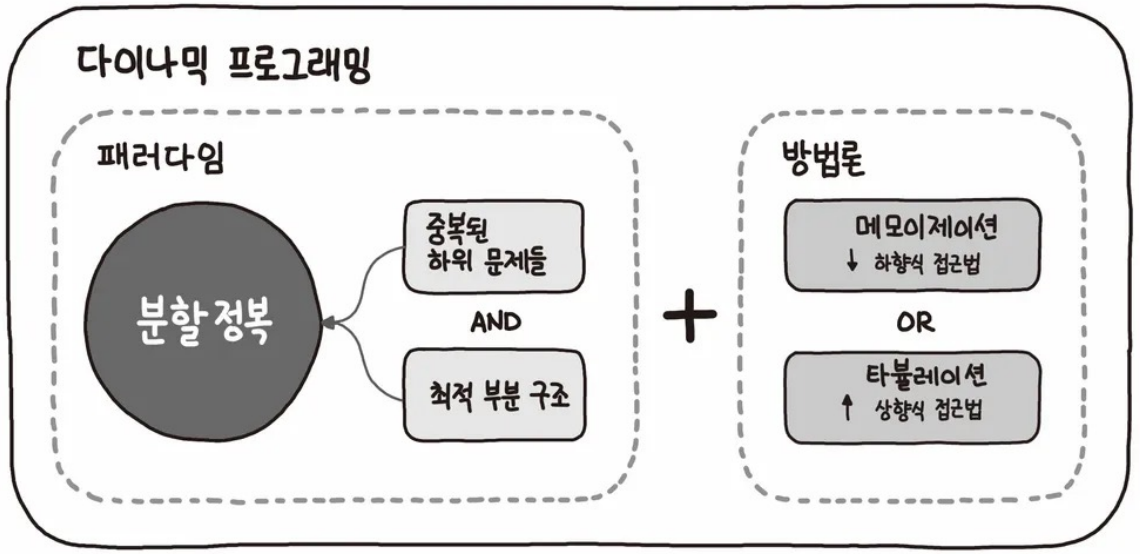
그럼 알고리즘을 왜 공부해야 하나요?

조금 더 컴퓨터스럽게 생각하는 방법을 통해서 최적화 하는 방식을 배우기 위해서!

- 테오의 프론트엔드 -

부모 컴포넌트가 리렌더링되니
변경 사항이 없는 자식 컴포넌트도 리렌더링 -> 비효율

DP의 개념을 적용한 useMemo
중복적인 계산을 제거하여 전체적인 실행속도를 빠르게 한다.
= 큰 문제를 해결하기 위해 작은 문제를 호출



Topic 40
리팩터링

리팩터링이란 밖으로 드러나는 동작은 그대로 유지한 채 내부 구조를 변경함으로써 이미 존재하는 코드를 재구성하는 체계적인 기법이다.

1. 이 활동은 체계적이며 아무렇게나 하는 것이 아니다.
   → 정확한 목적을 가지고 접근, ETC로 구현하는 이유
2. 밖으로 드러나는 동작은 바뀌지 않는다. 기능을 추가하는 작업이 아니다.
   → 자동화된 단위 테스트가 필요

언제 해야할까

- 중복
- 직교적이지 않은 설계
- 더 이상 유효하지 않는 지식
- 성능
- 테스트 통과

어떻게 해야할까

- 리팩터링과 기능 추가를 동시에 하지말라
- 리팩터링을 시작하기 전 든든한 테스트가 있는지 먼저 확인하라
- 단계를 작게 나누어서 신중하게 작업하라.

Tip 65. 일찍 리팩터링하고, 자주 리팩터링하라.

다시 코드를 여는 것은 고통스러운 작업일 수 있다.

특히 일정 압박으로 리팩터링을 하지 않기도 한다.

리팩터링이 필요한 코드는 일종의 종양이다. 작을 때 수정하자

Topic 41
테스트로 코딩하기

테스트가 코딩을 주도한다

Tip 66. 테스트는 버그를 찾기 위한 것이 아니다.

우리 메서드의 테스트 작성에 대해 생각함으로써 코드의 작성자가 아니라 사용자인 것처럼

메서드를 **외부의 시선으로 보게되었다**.

무언가를 테스트하기 좋게 만들면 **결합도도 낮아진다.**

Tip 67. 테스트가 코드의 첫 번째 사용이다

TDD: 목표가 어디인지 알아야 한다

TDD의 기본 주기

1. 추가하고 싶은 작은 기능 하나를 결정
2. 그 기능이 구현되었을 때 통과하게 될 테스트를 하나 작성
3. 테스트를 실행, 방금 추가한 테스트 딱 하나만 실패해야 함.
4. 실패하는 테스트를 통과시킬 수 있는 최소한의 코드만 작성. 모든 테스트가 통과하는지 확인
5. 코드를 리팩터링. 개선 후에도 테스트가 계속 통과하는지 확인

주의! TDD의 노예가 되지 않도록 해야 함 (과도한 시간투자, 많은 중복 테스트, 상향식 설계)

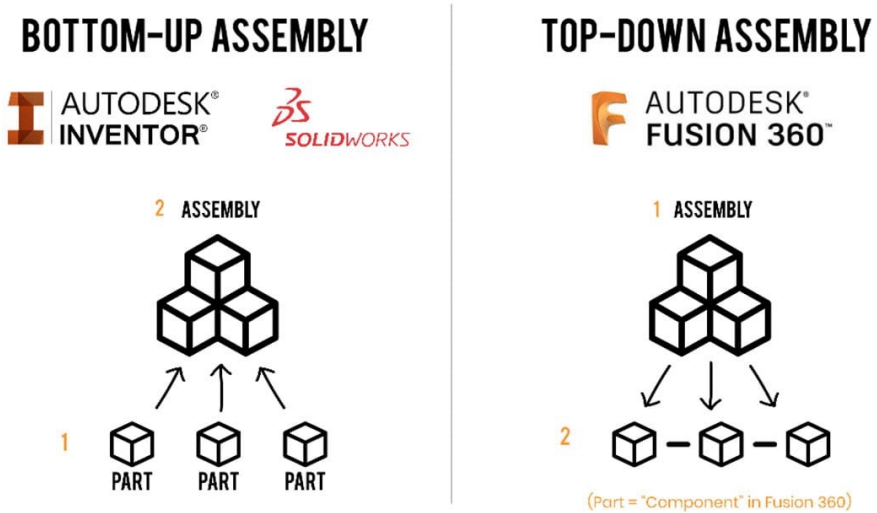
⇒ 큰 그림을 살피는 것을 잊지 말라.

Tip 68. 상향식이나 하향식이 아니라 끝에서 끝까지(end to end) 만들어라.

하향식 설계는 전체 요구 사항을 시작할 때 다 알고 있다고 가정하지만 사실은 알 수 없음

상향식 설계는 추상화 계층을 쌓다 보면 결국에는 하나의 최상위 해결 계층에 도착할 것이라고 가정.
하지만 목표가 어디인지 모르는데 어떻게 각 계층의 기능을 결정할 수 있을까?

두 방식 모두 뜻대로 되지 않음
점진적으로 한쪽 끝과 다른 쪽 끝을 잇는 조그만 기능 조각들을 만들고, 그 과정에서 문제에 대하여 배우기



계약을 지키는지 테스트하기

테스트는 우리에게 두 가지를 알려준다.

1.코드가 계약을 지키는지

2.코드로 표현된 계약의 의미가 우리가 생각하는 것과 일치하는지

여러개의 다른 모듈에 의존하는 경우에는? 그 모듈의 하위 컴포넌트들을 모두 검증한 후에야 해당 모듈 테스트 가능

번잡할 수 있지만 이후에 벌어질지 모를 재앙을 피하려고 노력하는 것

Tip 69. 테스트할 수 있도록 설계하라.

테스트 접점 만들기

아무리 테스트를 잘 갖추었어도 모든 버그를 발견할 수는 없다.

로깅을 잘하거나 에러 리포트 등을 잘 활용하자.

테스트 문화

최악의 선택은 '나중에 테스트' = '사실 테스트하지 않음'

제대로 된 테스트 문화를 가졌다면 모든 테스트가 언제나 통과해야 한다.

Tip 70. 여러분의 소프트웨어를 테스트하라. 그러지 않으면 사용자가 테스트하게 된다.

Topic 42
속성 기반 테스트

Tip 71. 속성 기반 테스트로 가정을 검증하라.

```
public class PassingGrade {
    public boolean passed(float grade) {
        if (grade < 1.0 || grade > 10.0) throw new IllegalArgumentException();
        return grade >= 5.0;
    }
}
```

기존 테스트 방식이라면? 조건문에 해당하는 input들을 넣어보면서 기대하는 결과값이 출력되는지를 확인

속성 기반 테스트? fail이 리턴되는 패턴, true가 리턴되는 패턴, exception이 발생되는 패턴
각각이 발생할만한 입력값들에 대해 범위를 제한해주고 랜덤한 파라미터들을 전달해서 테스트

Topic 43
바깥에서는 안전에 주의하라

Tip 72. 단순함을 유지하고 공격 표면을 최소화하라.

기본 보안 원칙

1.공격 표면을 최소화하라.

2.최소 권한 원칙

3.안전한 기본값

4.민감 정보를 암호화하라.

5.보안 업데이트를 적용하라.

Topic 44
이름 짓기

Tip 74. 이름을 잘 지어라. 필요하면 이름을 바꿔라.

스트루프 효과 - 색깔을 말하는 것보다 이름을 부르는 것이 잘 읽힘.
그러니까 이름을 지을 때는 표현하고 싶은 것을 더 명확하게 다듬기 위해 노력하자.

각 언어 사용자들이 선호하는 케이스를 무시하지 말기. (ex. 스네이크/카멜 케이스)

팀 내에서 특별한 의미가 있는 용어들을 비롯하여 고유의 어휘를 모든 사람이 뜻을 알고 일관성 있게 사용해야 한다.
→ 팀 내 의사소통 중요. 짝 프로그래밍.

잘못된 이름을 바꿀 수없는 상황이라면 더 큰 문제. ETC 위반.

이름을 바꾸기 쉽게 만들자.

7장 끝