실용주의 프로그래머

1장. 실용주의 철학

2장. 실용주의 접근법

실용주의 개발자가 되려면

1.어떤 일을 하면서 자신이 무엇을 하고 있는지 생각해야함

2.개발 과정에서 내리는 모든 결정을 끊임없이 비판적으로 평가해야함

Tip1. 자신의 기예(craft)에 관심을 가져라.

실용주의 개발자가 갖는 여러 특징 중 공통적이면서 기본

- 새로운 것에 빨리 적응
- 호기심
- 비판적인 사고
- 현실주의자
- 다방면에 능숙

Tip3. 자신의 당신에게는 agency(자율성)가 있다.

- 당신에게는 스스로의 행동을 직접 결정할 수 있는 힘이 있음
- 현재 조직 내에서 일하는 방식을 바꾸는 등 조직의 변화를 꾀하기 잘 안되면 이직~
- 기술에 뒤쳐지는 기분이 든다면 여가 시간을 쪼개서 재미있어 보이는 것을 공부하기
- 우선 주도적으로 행동하고 기회를 잡아라

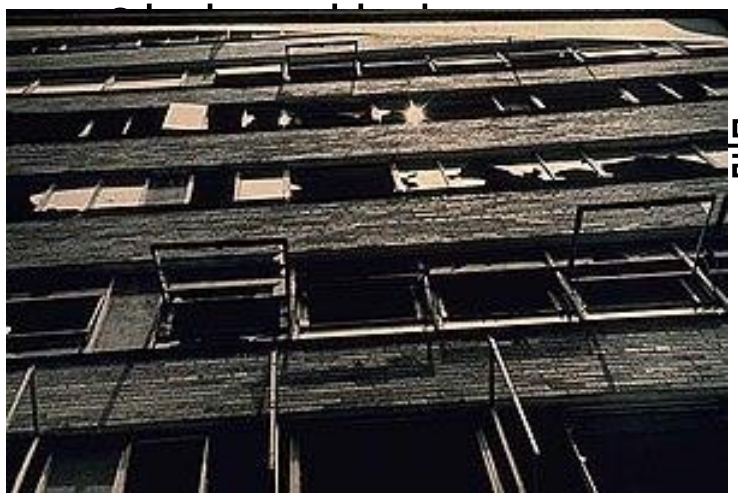
Tip4. 어설픈 변명 말고 대안을 제시하라.

- 개인적으로 최선을 다하는 것이 전부가 아니다.

  통제할 수 없는 위험 요소가 있지 않은지 상황을 분석해야 한다.
- 고양이가 내 소스 코드를 삼켰어요
  - 다른 사람 혹은 다른 무언가를 비난하거나 변명을 만들어 내지 말라

Tip5. 깨진 창문을 내버려 두지 말라.

- 소프트웨어의 무질서도가 증가할 때 우리는 이를 '소프트웨어의 부패'라고 일컫는다.
  - 원영적 사고: 기술 부채 <- 매일 하는 합리화...
- 나쁜 설계, 잘못된 결정, 혹은 형편없는 코드 등이 모두 깨진 창문이다.
- 발견하자마자 고칠 수 없다면 주석처리, 더미 데이터와 같은 판자로 덮는 것만



깨진 창문 이론:
깨진 유리창 하나를 방치하면 [illegible] 가 확산되기 시작한다는 이론

https://ko.wikipedia.org/wiki/%EA%B9%A8%EC%A7%84_%EC%9C%A0%EB%A6%AC%EC%B0%BD_%EC%9D%B4%EB%A1%A0

Tip6. 변화의 촉매가 되라

- 무엇을 해야 하는지, 어떻게 해야 하는지 정확히 아는 경우가 있다.
- 하지만 일에 착수하려고 허락을 구하는 때부터, 뭔가가 지연되거나 일이 복잡해지기 시작한다.

  ⇒ 시작 피로, 한번에 크게 바꾸려고 하면 좌절됨.
- 큰 무리 없이 요구할 수 있을 만한 것을 찾아라. 그리고 그걸 잘 개발하라 → MVP
- 미래를 살짝이라도 보여 주면 사람들은 도와주기 위해 모여들 것이다.

Tip8. 품질을 요구 사항으로 만들어라

- 오늘의 훌륭한 소프트웨어는 많은 경우 환상에 불과한 내일의 완벽한 소프트웨어보다 낫다.
- 사용자에게 뭔가 직접 만져볼 수 있는 것을 일찍 준다면,
  피드백을 통해 종국에는 더 나은 해결책에 도달할 수 있을 것이다.
- 완벽하게 훌륭한 프로그램을 과도하게 장식하거나 지나칠 정도로 다듬느라 망치지 말라.
  완벽해지기란 불가능
- 버전 관리를 잘 해야겠다..(물론 제대로 해본 적이 없음)

Tip9. 지식 포트폴리오에 주기적으로 투자하라

- 새로운 것을 배우는 능력은 가장 중요한 전략 자산임
- 포트폴리오 만들기
  - 주기적인 투자
  - 다각화
  - 리스크 관리
  - 싸게 사서 비싸게 팔기
  - 기술 검토 및 재조정
- 지식 자산 쌓기
  - 매년 새로운 언어 배우기
  - 기술 서적 읽기
  - 수업 듣기, 지역 모임 참여
  - 트렌드 알아보기
- 당장 프로젝트에 적용하지 않더라도 사고 확장

Tip12. 무엇을 말하는가와 어떻게 말하는가는 모두 중요하다.

- 개발자로서 우리는 여러 입장에서 소통해야 한다.
  - 최종 사용자와도, 기계와도, 다음 세대의 개발자들에게도...
- How?
  - 청중을 알기, 피드백 모으기
  - 무엇을 말할지 미리 계획하고 개요를 작성하기. 듣는 사람을 고려하기
  - 말하는 시점도 적절하게
  - 문서를 만들 때 내용에만 집중하지 말기. 멋져 보이게 하기
  - 청중을 참여시키기

Tip14. 좋은 설계는 나쁜 설계보다 바꾸기 쉽다.

- ETC(Easier to Change)원칙을 따르기
  - 결합도를 줄이면 → 관심사 분리 → 각각이 바꾸기 쉬워짐
  - 단일 책임 원칙 SRP → 요구 사항이 바뀌더라도 모듈 하나만 바꿔서 반영 가능
  - 왜 이름짓기가 중요? → 코드 읽기 쉬워짐 → 코드를 바꾸려면 코드를 읽어야 함

  - 의식적으로 ETC라는 가치를 적용하려고 노력해야함.

Tip15. DRY: 반복하지 말라 Don’t Repeat Yourself

- 지식은 고정적이지 않고 이해는 날마다 바뀜 -> 늘 유지보수 모드에 있음
- 명세와 프로세스, 개발하는 프로그램 안에 지식을 중복해서 넣는다면?

  -> 유지보수가 어려워짐
- DRY 원칙: 모든 지식은 시스템 내에서 단 한 번만, 애매하지 않고, 권위 있게 표현되어야 한다.

Tip15. DRY: 반복하지 말라 Don’t Repeat Yourself

- 코드의 중복?

```
def print_balance(account)
  printf "Debits: %10.2f\n", account.debits
  printf "Credits: %10.2f\n", account.credits
  if account.fees < 0
    printf "Fees: %10.2f-\n", -account.fees
  else
    printf "Fees: %10.2f\n", account.fees
  end
  printf "            ----------\n"
  if account.balance < 0
    printf "Balance: %10.2f-\n", account.balance
  else
    printf "Balance: %10.2f\n", account.balance
  end
end
```

```
def validate_age(value):
  validate_type(value, :integer)
  validate_min_integer(value, 1)

def validate_quantity(value):
  validate_type(value, :integer)
  validate_min_integer(value, 1)
```

Tip15. DRY: 반복하지 말라 Don’t Repeat Yourself

- 문서화 중복

```
// 사용자의 나이를 1 증가시킴
function increaseAge(user) {
    user.age += 1;
}
```

Tip15. DRY: 반복하지 말라 Don't Repeat Yourself

- 데이터 중복

```
class Line {
  Point start;
  Point end;
  double length;
}
```

start와 end를 알면 length는 자동으로 결정

비용이 많이 드는 연산을 여러 번 수행하지 않기 위해 그냥 저장하면 안되나?
→ 대신 바깥 세상에 DRY 원칙 위배를 노출하지 말자

```
class Line {
  private Point start;
  private Point end;
  private double length;

  public Lint(Point start, Point end){
    this.start = start;
    this.end = end;
    calculateLength();
  }

  void setStart(Point p) {this.start = p; calculateLength();}
  void setEnd(Point p) {this.end = p; calculateLength();}

  Point getStart() {return start;}
  Point getEnd() {return end;}

  double getLength() {return length;}

  private void calculateLength() {
    this.length = start.distanceTo(end);
  }
}
```

Tip18. 최종 결정이란 없다.

- 프로젝트 초기에는 최종 결정을 줄여야 함
  - 초기에 최선의 결정을 내리지 못 함
  - 중요한 결정은 쉽게 되돌릴 수 없음
- 결정은 돌에 새겨진 글씨가 아님
  - SW 개발 속도는 요구 사항, 사용자, HW 변화를 앞지를 수 없음
- 선택의 여지를 남겨둘 수 있는 구조 필요
  - 즉 변화에 유연하게 대응할 수 있는 역량 필요

Tip20. 목표물을 찾기 위해 예광탄을 써라.

- 즉 소프트웨어에서는 가장 복잡한 곳의 코드를 제일 먼저 짜보는 것, 테스트 해보는 것
- 예광탄 코드는 기능은 없지만 골격이 만들어지는 최초의 코드

  -> 일관성, 생산성 증대, 디버깅 및 테스트 속도 증가
- 예광탄은 지금 맞히고 있는 것이 무엇인지를 보여주는 것이지 꼭 목표물을 맞추는 것이 아님. 목표에 맞을 때까지 개발
- 프로토타입은 나중에 버리는 코드를 만들지만,

  예광탄 코드는 완결된 코드이며 골격의 일부가 된다

Tip21. 프로토타이핑으로 학습하라.

- 전체적으로 시스템이 어떻게 동작할지에 대한 감을 잡는 것이다.
- 프로토타입의 핵심: 생산한 코드가 아닌 이를 통해 배우는 교훈, 전체적으로 시스템이 어떻게 동작할지 아는 것
- 프로토타입을 만들 때 중요한 점은 증명을 한 후에는 프로토타입을 버려야 한다는 것
  - 나무 자동차를 타고 러시아워에 운전할 수는 없다.

Tip23. 추정으로 놀람을 피하라.

- 추정하는 법을 배우고 추정 능력을 계발하여 무언가의 규모를 직관적으로 짚을 정도가 되면, 추정 대상의 가능성을 가늠할 수 있음
- 초기 기능의 구현과 테스트를 마친 후, 이를 첫 번째 이터레이션의 끝으로 삼기
  - 주기의 경험을 바탕으로 이터레이션의 수와 각 이터레이션에서 무엇을 할지에 대한 처음의 추측을 다듬기