2019/11/7

-Clean Code I-

- Simplicity is hard

- Use Intention-Revealing Name : 코드를 이해하기 쉽게 naming을 해야한다. 시간낭비가 아니다.
  - 일관적으로
  - 읽을 수 있게
  - class 명사, method는 동사로 시작
    - 코드를 읽을때 이야기가 나온다.

- 함수 코드 크기가 작아야 한다.
  - 예) Rule of 30
  - 한가지 일만 하게 하라
    - 이름짓기 쉬움
    - 두개 이상일을 하면 이름으로 표현
    - 함수이름은 길어도 된다. 이해하기 쉽게 한다면 이해하기 쉬운 이름 > (better) 짧은 이름
  - argument 갯수를 최소화
    - 많을 경우, ex) Named Argument
      => error를 막을 수 있음
  - No side effect
    - side effect 가 있으면 이름에 표기
  - Command Query Separation
    - Do one thing rule 의 일부
    - 함수 이름으로 표시
  - prefer Exception to returning Error Codes
    - Exception handling 코드작성에 절반 이상의 시간을 쓸 수 있다.
    - 반드시 Exception 으로 구현

2019/9/24

- Clean Code II -

* Comment 는 우리의 생각을 코드로 표현할 수 없는 경우에 만 적는 것이다.
  - 항상 작성해야 하는 것이 아니다.

· 함수나 변수의 이름을 잘 정하면 comment 를 지을 필요가 없을 수 있다.

· Code로 표현할 수 없는 comment
  1. 의도 : ~하려고 ex) in order to, so that
  2. 이유 : ~ 때문에 ex) because, since

  ⇒ 코드만 읽고 파악하기 어려운 설계정보
  * 고급정보

  - Comment 에 키워드를 강제해도 된다.

· Commented-Out Code : 가끔 유용한 경우도 있다.

· 원칙을 세워두되 상황에 맞는 최선의 결정을 할 수 있어야 한다.

· Invariant 로 comment 를 대신할 수 있다.
  - ex) assert{}
  - higher-order function 을 이용하여 간결하게 표현 가능
  - * loop invariant 를 작성하자 ⇒ 고급정보
  - Data 구조에 대한 설명을 invariant 로 대신
    ex) method로 작성후 assert 에서 호출

· Return Code 대신 Exception을 사용
  - 원칙이다.
  - 예외 : Exception의 종류가 너무 많은 경우

- Unchecked Exception 을 사용해야 한다.
  - 프로그래머가 계속 추적해야 한다.
    유저보수 비용이 크다.
    ex) lib 이바뀌면 호환성이 맞지 않을 수 있다.
  - Higher-order function 을 사용하면 checked Exception 을 쓸 수 없다. ex) 함수가 인자로 들어오는 경우
  - checked exception 을 남용하지 말자
  - * Exception 을 ignore 하면 안된다.

· Null을 사용하지 말자
  → Option type 을 쓰자
  - NPE를 줄일 수 있다.

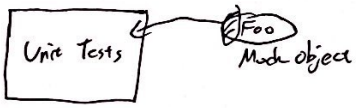
20/10/8

- Clean Code Ⅲ -

- Waterfall Model : 전통적인 개발모델
  - 초기단계로 돌아가면 cost가 크다.
  - 초기에 requirement를 찾아내기 어렵다.
  - 큰 회사에서 주로 이용

- Traditional Test-Last vs Test Driven Development (TDD)
  - TDD
    - Unit test를 만드는 과정 자체가 Design
    - Regression testing : 중간중간에 확인용도
    - Test가 자산으로 계속해서 쌓인다.

- Unit Tests : Class나 method 등 하나의 Component를 test 하기 위함
  - No dependencies on outside system



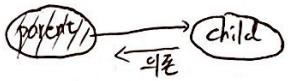
  - 단점 :
    1) 관리비용이 막대해진다.
    2) False positive → 시간허비
    3) 개발후 거의 useless → 거의항상 pass
    ☆4) Unit tests에 너무 의존하게 된다. (안좋은 습관)
    → Production code에 더 집중해야한다.
    → 조직에서 안좋은 문화

- test code를 production code 처럼 관리해야 한다.
  - integration, system test는 꼼꼼히 설계, 관리해야한다
  - Unit test는 버려도 된다. → Unit test mass 문제 발생, otherwise
    ⇒ Unit test는 간단해야 하므로 개발중간에 다시 작성하여도 큰 문제가 없다.

- Do one thing at a time, and do it well
  for tests

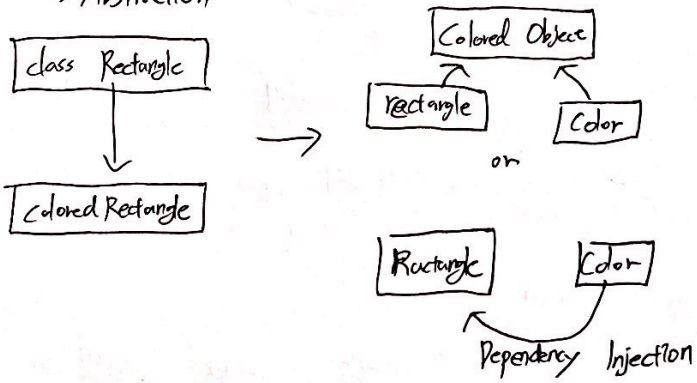
- class 설계는 굉장히 어려운 문제이다.

☆ Favor composition over inheritance ("effective Java")
  - Inheritance는 encapsulation을 위반한다.



  - Inheritance는 기본적으로 안좋은 것이다.
    except) 같은 package, programmer 의미상 적합할때

☆ Composability : CS 근본적 원리
  → Separation of concern
  → Divide and Conquer
  → Abstraction



20/10/15

-Mythical Man-Month I -

: Software engineering의 성경책

- 명언은 많지만 실천하는 사람은 드물다.

· The tar pit

- 프로그래밍 재밌는 이유 : making, complex, tractable medium

⇒ 하지만 tar pit에 빠질 수 있다.

· Software project가 실패하는 가장 큰 이유는 시간 부족이다.

· 프로그래머의 생산성은 ○ 모는 음수도 될 수 있다.

· all programmers are optimists (All will go well)

; Root of All Evil

- optimism에 대한 근거가 증명하지 않음을 받아드려야 한다.

⇒설계와 design 단계에서 optimism을 버려야 한다.

· Man-Month

- No communication among workers 일때만 통한다.

⇒ communication overhead를 줄이도록 중요하다

· optimism 때문에 testing을 등한시하게 된다.

⇒ Rule of thumb: planning $\frac{1}{3}$, coding $\frac{1}{6}$, component test $\frac{1}{4}$, system test $\frac{1}{4}$] $-\frac{1}{2}$

· 보통 optimism이 날아 있을 때 예상시간x2x120% 정도가 걸린다

Brooks's Law

"Adding manpower to a late software project make it later"

→ sequential constraint 때문

· Software project에서 가장 중요한 자원은 시간이다.

⇒ communication [traing cost (X), inter commou(✓)-줄이기

⇒ 창의적이지 않고 할줄하는 일에 쓰는 시간 줄이기 ex)문서화, 테스팅 자동화, 병렬화

20/10/22

-Mythical Man-Month II-

· The Surgical Team

- Productivity Variations : 경험이 많은 프로그래머들 끼리도 생산성 차이가 10배까지 난다.

- As few minds as possible : 가능한 작고 똑똑한 사람들로 이루어진 팀을 꾸리자.
  ex) 10명이상 (X)

- Mills's proposal :
  큰 팀을 작은 팀으로 나누고 각 팀은 surgical team 모델을 따른다.
  - Surgeon : chief programmer 최종결정권자
  - Copilot : Surgeon 대체
  - Toolsmith
  - Tester
  - etc...

  ⇒ Radically simpler communication pattern
  - 일종의 문화이다. 민주주의는 치명적 → Comm. overhead↑, 현실문제
    → Surgeon의 능력이 절대적으로 중요

Ex) Amazon
- 5~8 team
- design, build, test, deploy
- 팀간의 coordination problem 최소화

· Sotware project가 실패하는 이유는 자기자신의 무게 때문에 그렇다.
⇒ communication ↑, mistakes ↑
⇒ doc로 communication을 대체

· UML은 잘 안쓴다.
- overhead가 너무 크다.

· No silver Bullet

"There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity"

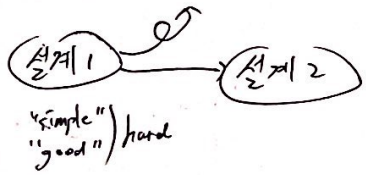
- Software Project = Werewolf

- Essential difficulties vs Accidental difficulties
  · 제거불가 · 제거 가능
  · 가장 어려움

20/11/5

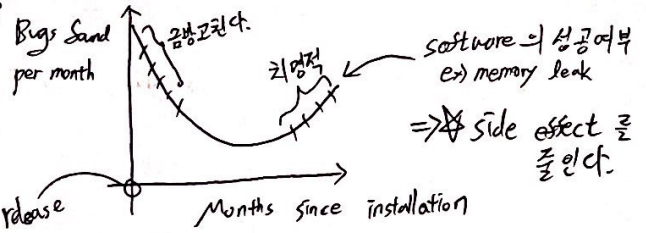
-Mythical Man-Month III-

- Software의 integrity를 위해서는 누군가가 모든 계획을 지배하여 조율해야 한다.
- the second is the most dangerous system a person ever designs ; the general tendency is to over-design it.
- 문서화를 통해 asynchronous comm. 를 해야한다.
- ☆ Representation is the essence of programming
  - Data를 어떻게 설계하느냐에 따라 algorithm은 달라질 수 있다.
- 중간에 설계를 버릴 수 있어야 한다는 것을 인정해야 한다.



- 유지보수 비용이 개발 비용보다 크다

☆



- Bug를 고치는 것이 새로운 버그를 만들 확률이 50%가 넘는다.
  ⇒ tracking이 가능해야 한다 ex) commit
- Side effect를 줄여야 software quality를 높일 수 있다. (치명적 버그를 고칠 수 있음)
  - immutable
  - functional programming

- System debugging에 공을 많이 들일 필요가 있다.
  ⇒ 개발코드 양의 50% 정도가 system test code 일 수 있다.
- Milestone을 만드는 것이 due를 맞추는데 중요하다.
  → 명확하게 계획을 짜서 프로그래머가 자기 자신을 속일 수 없게 한다.
- Document에서는 "overview"를 주는 게 중요하다.
- The tar pit of software engineering will continue to be sticky for a long time to come.
  (40년전, 지금 더 심해짐

20/11/12

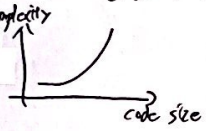
- Out of the tar pit -

- root causes of software problems
  1) Complexity
  2) Conformity
  3) Changeability
  4) Invisibility

★ Simplicity is hard

- program을 이해하는데 testing 보다는 informal reasoning이 더 중요하다. ⇒ simplicity is important

- Causes of Complexity
  1) State
  2) Control ex) sequencing, branching
  3) Volume



- Solutions
  - OOP
    ★ - object identity 문제 ex) equals, hash
  - Functional Programming
    - Referential transparency
      : same argument → same result
    - abstraction level ↑
  - Logic programming

⇒ F.P + L.P + state + control

- Essential Complexity vs Accidental complexity
  - ideal world
    - 나머지
    - 줄일 수 있다.

state → complexity : Ideal world

⇒ Minimize accidental state

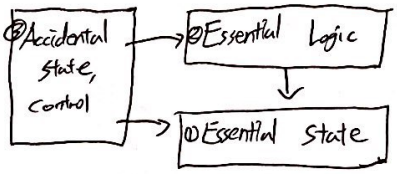
⇒ Classify data

Control은 전부 accidental

↓

- Required accidental complexity 를 찾아낸다. ex) cache : Real world

⇒ Ideal world의 essential complexity와 Real world에서 accidental complexity 빼고는 전부 버린다.



★ <Seperation>

- Complexity를 줄이기 위해 의식적인 노력이 필요하다.

20/11/17

- Concurrent programming I -

- process vs thread
- JVM == process
- happens-before Relation : visible of write effect
  1) Program Order : in the same thread
  2) Transitivity
  3) Volatile Fields
  4) Moritor Locking
  5) Thread - start
  6) Thread - Termination

20/11/24

- Concurrent Programming II -

- Future[T] : building block of concurrency

  ※ Future 생성은 즉시되지만 value는 나중에 나온다 (Separation of Concern)

  - Callbacks : Future가 완료되면 다른 thread에서 돌아가는 코드

- Fatal Exceptions in Futures are not caught
- prefer 'foreach'.
- Promise[T] : single-assignment variables
- Promise.future : future that start when promise complete
- Why Promise is powerful?

  Concern 1 : when the action should be taken
  Concern 2 : What the action should be taken

  → Separation of Concern

20/11/19

- Testing -

Scenario of error

1. 실제 bug가 있는경우
   ⇒ 처리로직 구현
2. 함수가 어떻게 쓰일지 명시하는 경우
   ⇒ ex) require {}
3. 함수가 어떻게 쓰일지 명확한 경우
   ⇒ ex) assert {}

· Invariant
  - Connecting two module
  - State transition
  - Collection
  - loop invariants
  - Thread

· Invariant가 critical mass에 도달하게 되면 굉장히 도움이 된다.
  - Combined strength grows

· Wrong change를 dependency Invariant로 쉽게 잡을 수 있다.

· Invariant + Integration Tests : 시너지가 있다.
  : Unit Test보다 낫다.

20/12/10

- Review -

Java ⟶ Scala - 언어의힘
-생산성↓

중요한 개념

1. 주석은 X , 주석 = why는
decision : 고급정보
주석에는 반드시 고급정보만 넣어야한다.

2. No silver bullet

3. "optimists"
leader가되었을 때 이런 사람심리를 기억할필요가 있다.

★4. Brooks's law
↳"communication overhead" (asynch / synch : overhead크다)
가장중요한 resource는 사람이 더하기 시간이다.

★5. No side-effect

6. Composition vs inheritance (X)

composability ← computational thinking의 기본
→dependency injection

7. Surgical model vs ~~인수추억~~

8. Simplicity is hard

9. equals(), hashcode() 구현이 매우 어렵다.
→ caseclass 사용

★10. Promise
가장 멋있는 abstraction
Separation of concern
"When" ⟷ "what"
분리

11. assert : invariant
↳고급 programming

Scala를 배웠으면 다른 어떤 언어도 배울수 있다.
except Haskell