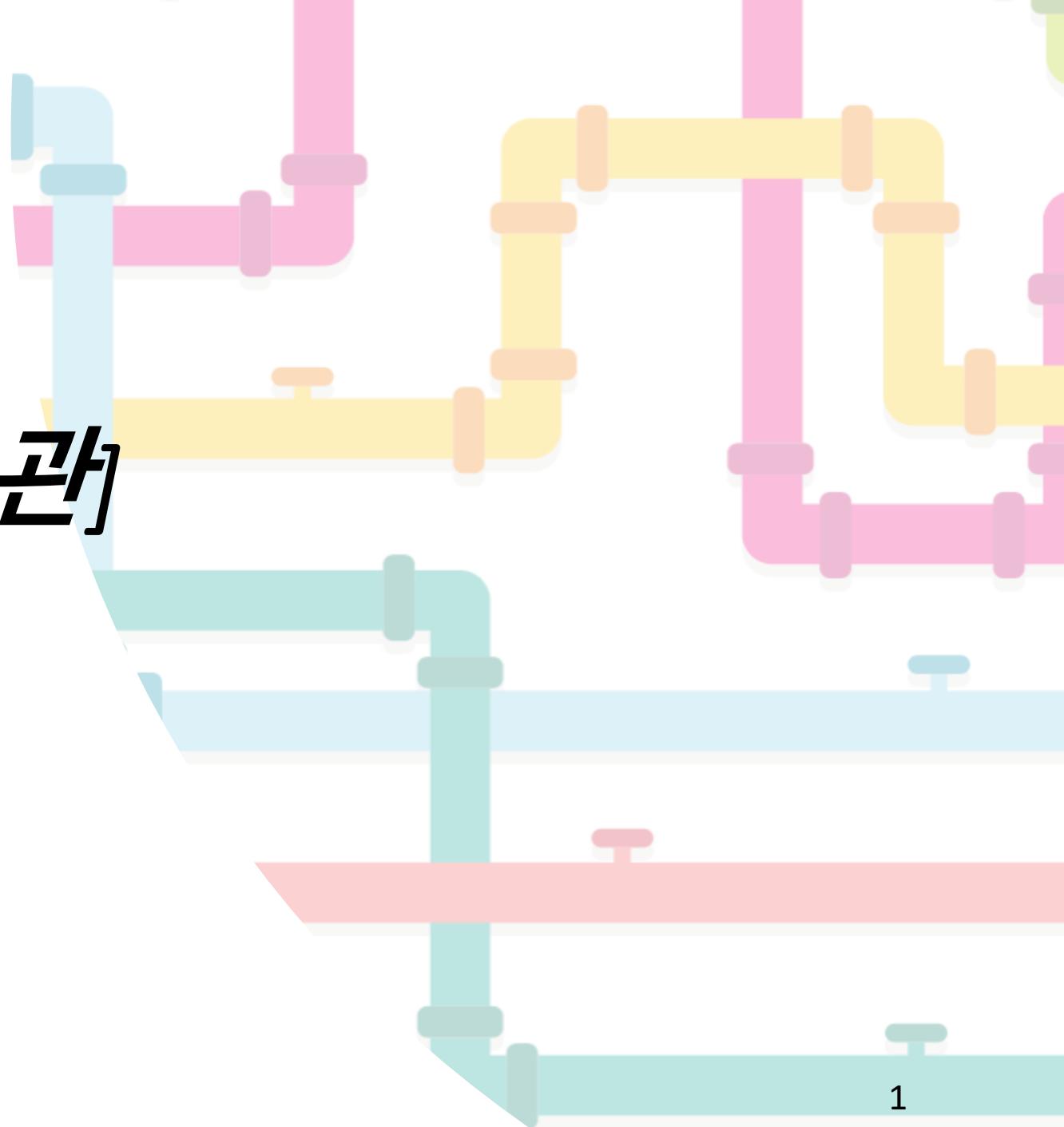

[올바른 Java 코딩습관]

Clean Code





Contents

1. Clean Code
2. Code Smell
3. Code Review
4. TDD
5. Clean Architecture



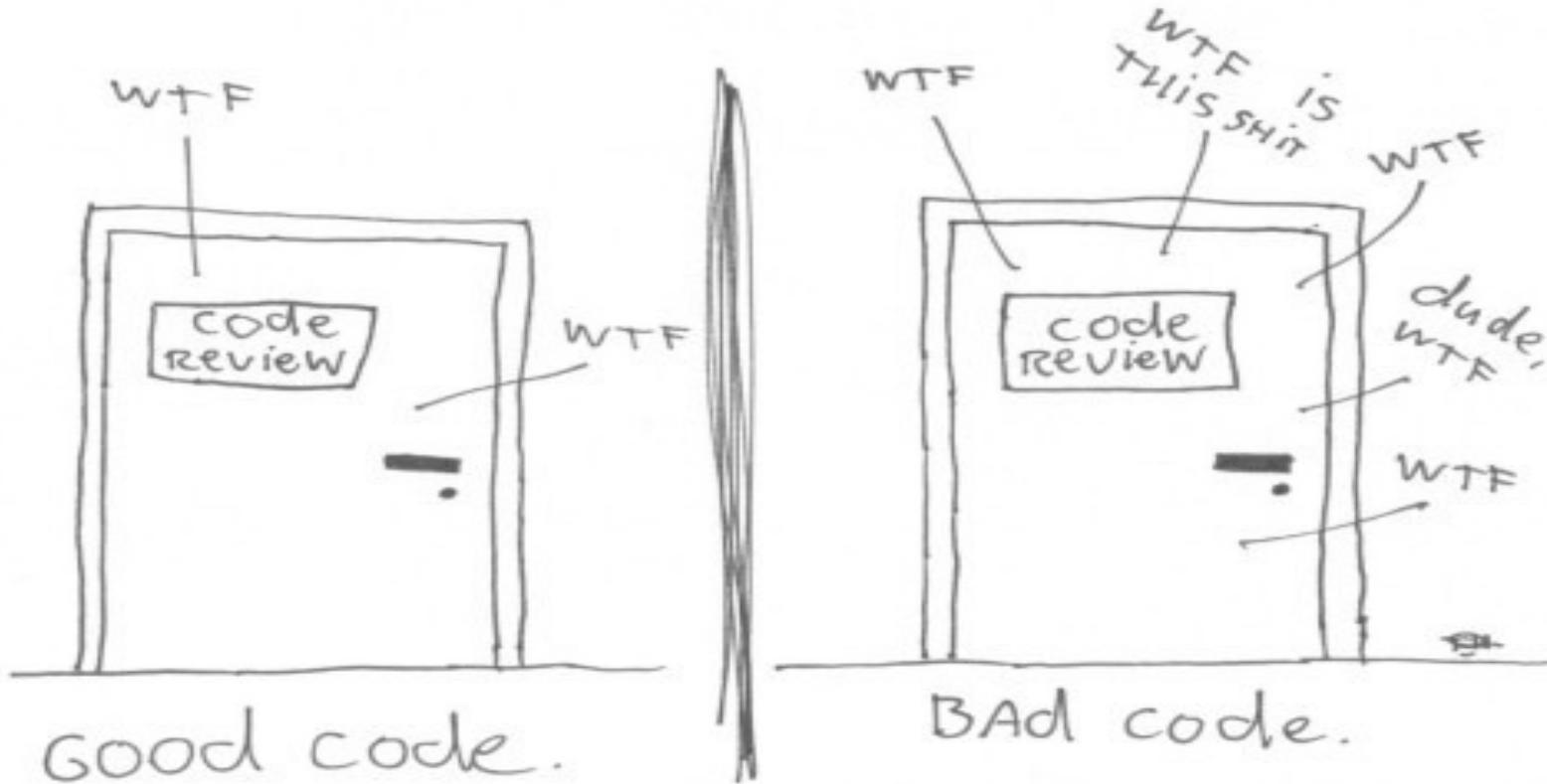
Clean Code

프로그래밍이란?

- “프로그래밍이란 기계가 실행할 정도로 상세하게 요구사항을 명시하는 작업이다.
- 따라서 아무리 코드를 자동으로 생성하는 시대가 온다고 하더라도 요구사항을 명시하는 작업이 사라지지 않는 한 프로그래밍은 없어지지 않는다.
- 다만 프로그래밍 언어의 추상화 수준은 점차 높아질 뿐이다.”

코드 품질을 측정하는 척도

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



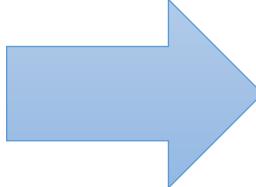
Bad vs Good



나쁜 코드

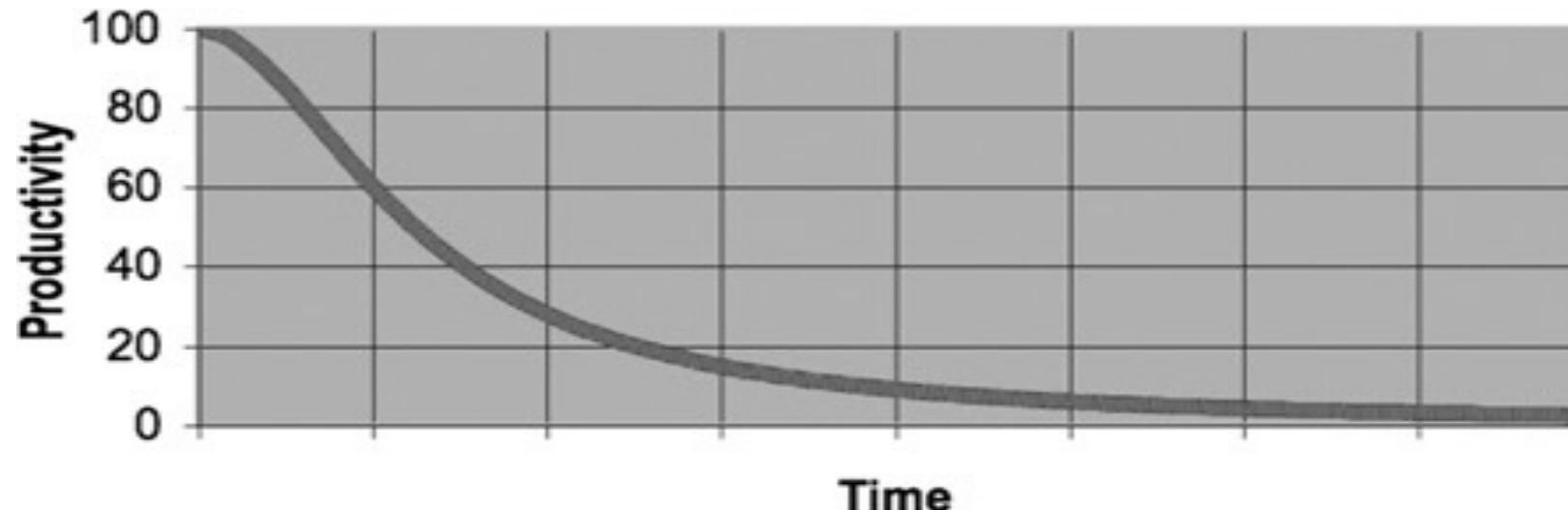
- 나쁜 코드에 발목이 잡혀 고생한 기억이 있는가?
- 왜 나쁜 코드를 짰는가?
 - 급해서? 서두르느라?
 - 제대로 짤 시간이 없다고 생각해서
 - 코드를 다듬느라 시간을 보냈다가 상사한테 욕 먹을까봐
 - 지겨워서 빨리 끝내려고
 - 다른 업무가 너무 밀려 후딱 해치우고 밀린 업무로 넘어가려고

나쁜 코드

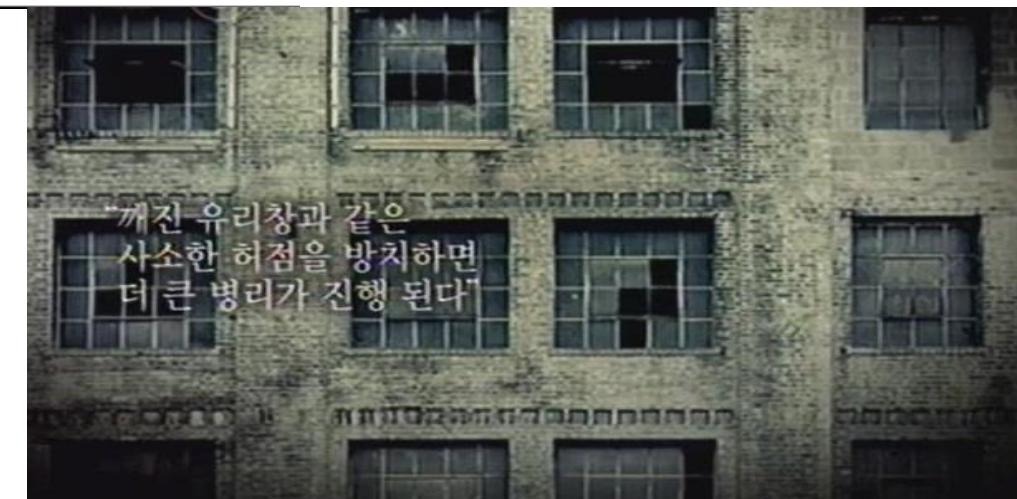
- 대충 짠 프로그램이 돌아간다 → Relief
 - 안 돌아가는 프로그램보다는 돌아가는 쓰레기가 좋다고 생각
 - 쓰레기 코드를 쳐다보며 나중에 꼭! 손보겠다고 생각 마라.
 - 르블랑의 법칙(LeBlanc's Law)
- 
- “Later equals never.”***

나쁜 코드로 치르는 대가

- 점진적인 생산성 저하



- 나쁜 코드는 개발 속도를 크게 떨어뜨린다.
- 깨진 창문을 그대로 두지 말자.



클린 코드라는 예술

- 클린 코드를 어떻게 작성할까?
 - 나쁜 코드 식별 능력이 **클린 코드 작성 능력**은 아님
 - “**Clean**”이라는 감각으로 자잘한 기법들을 적용하는 절제와 규율이 필요
 - “**코드 감각**”이 중요

가독성

- 코드는 이해하기 쉬워야 한다.
- 코드는 다른 사람이 그것을 이해하는 데 들이는 시간을 최소화하는 방식으로 작성되어야 한다.
- 1회용 코드는 되도록 피해야 한다. (스스로가 희생양이 될지도)
→ 참고: Perl 코드 (WORN: Write Once Read Never)

개발자 vs 관리자

- 일정과 요구사항으로 강력하게 밀어붙이는 관리자 핑계 대지 말고 일정에 쫓기더라도 좋은 코드를 만들려 하는 프로그래머가 되어라.
- 시간이 걸리니 손을 씻지 말고 수술해달라는 환자의 말을 듣고 손을 씻지 않으면, 책임은 의사에게 있다.
- 나쁜 코드의 위험을 이해하지 못하는 관리자의 말을 그대로 따르는 행동은 위험하다.

- 읽기 쉬운 코드가 매우 중요
- 읽기 쉬운 코드를 짜기가 쉽지 않더라도, 기존 코드를 읽어야 새 코드를 짜므로 읽기 쉽게 만들면 짜기도 쉬워진다.
- 주변 코드를 읽지 않으면 새 코드를 짜지 못한다.
- 급하다면, 서둘러 끝내려면, 쉽게 짜려면 읽기 쉽게 만들면 된다.

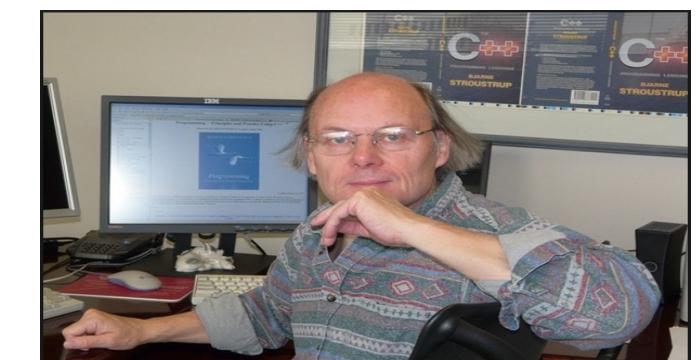
코드 읽기 : 코드 쓰기 = 10 : 1

클린 코드란?

비야네 스트롭스트롭 (C++ 창시자)

“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 업수 최적화로 코드를 망치려는 유혹에 빠지지 않는다.”

“깨끗한 코드는 한 가지를 제대로 한다.”



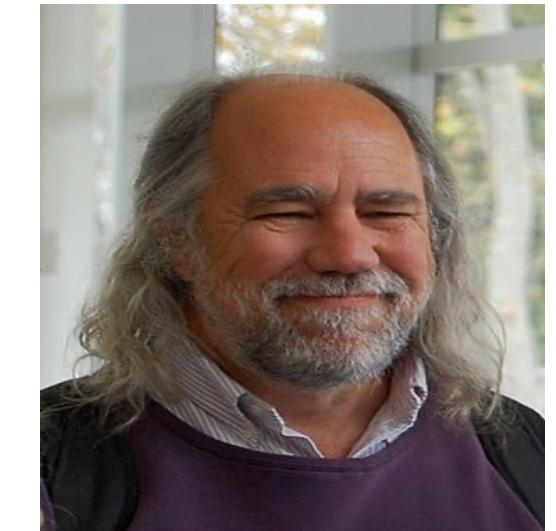
클린 코드란?

그레이디 부치 (객체지향 대가, UML 개발)

“ 깨끗한 코드는 단순하고 직관적이다. 깨끗한 코드는 잘 쓴 문장처럼 읽힌다.

깨끗한 코드는 결코 설계자의 의도를 숨기지 않는다.

오히려 명쾌한 추상화와 단순한 제어문으로 가득하다.”



클린 코드란?

데이브 토마스(실용주의 프로그래머)

“깨끗한 코드는 작성자가 아닌 사람도 읽기 쉽고 고치기 쉽다. 단위 테스트 케이스와 수용 테스트 케이스가 존재한다. 이런 이름은 의미가 있다. 특정 목적을 달성하는 방법은 (여러 가지가 아니라) 하나만 제공한다. 의존성은 최소이며 각 의존성을 명확히 정의한다. API는 명확하며 최소로 줄였다.

따로는 필요한 정보 전부를 코드만으로 명확하게 드러내기 어려우므로 언어에 따라 문학적 표현도 필요하다.”



클린 코드란?

마이클 페더즈 (“Working Effectively with Legacy Code” 저자)

깨끗한 코드의 특징은 많지만 그 중에서도 모두를 아우르는 특징이 하나 있다.

“깨끗한 코드는 언제나 누군가 주의 깊게 짚다는 느낌을 준다.”

고치려고 살펴봐도 딱히 손 댈 곳이 없다. 작성자가 이미 모든 사항을 고려했으므로,

고칠 궁리를 하다 보면 언제나 제자리로 돌아온다.

그리고는 누군가 남겨준 코드 누군가 주의 깊게 짜놓은 작품에 감사할 느낀다.

클린 코드란?

론 제프리 (“Extreme Programming Installed” 저자, 애자일 선언)

최근 들어 나는 [컨트 벡]이 제안한 간단한 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, 메소드, 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.

“중복 줄이기, 표현력 높이기, 초반부터 간단한 추상화 고려하기”

내게는 이 세가지가 깨끗한 코드를 만드는 비법이다.



클린 코드란?

워드 커닝햄(위키 창시자, 피트 창시자, 익스트림 프로그래밍 창시자)

코드를 읽으면서 짐작했던 기능을 각 루틴이 그대로 수행한다면 깨끗한
코드라 불러도 되겠다.

“코드가 그 문제를 풀기 위한 언어처럼 보인다면 아름다운 코드가 불러도
되겠다.”



클린 코드 → 리팩토링

클린코드 = 리팩토링 → 나쁜 코드를 좋은 코드로 바꾸는 것

```
// Private properties
private var blur_filter:BlurFilter;           // Blurs trails
private var screen_bmp:Bitmap;                // Captures screen to
private var background_mc:Sprite;             // Hold trails
private var tags_mc:Sprite;                   // Holds tags

// Initialization
public function Main()
{
    // Set up stage
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    // Add UI event listeners
    UI.addEventListener(UIEvent.CLICK, clicked);
    UI.addEventListener(UIEvent.DATA_LOADED, dataLoaded);

    // Add layers
    addChild(background_mc = new Sprite());
    addChild(tags_mc = new Sprite());

    // Create background bitmap
    background_mc.addChild(screen_bmp = new Bitmap());

    // Prevents offscreen content from rendering
    scrollRect = new Rectangle(0, 0, stage.stageWidth, stage.stageHeight);
    cacheAsBitmap = true;

    // Create blur filter
    blur_filter = new BlurFilter(3, 3, BitmapFilterQuality.LOW);
}
```

```
private var scr
protected var bg:Sprite
private var stuff:Sprite;
public function myprogram(){
    stage.align = "tl";
    stage.scaleMode ="noScale";

    UI.addEventListener("click",function(e){trace("please click!");
    UI.addEventListener("dataloaded",callback);

    addChild(bg=new Sprite());
    addChild(stuff=new Sprite());
    bg.addChild(scr=new Bitmap());
    scrollRect=new Rectangle(0,0,800,400);
    cacheAsBitmap= true;
    bf= new BlurFilter(3,3,1);

    stage.addEventListener("resize", callback2);
    callback2(null);

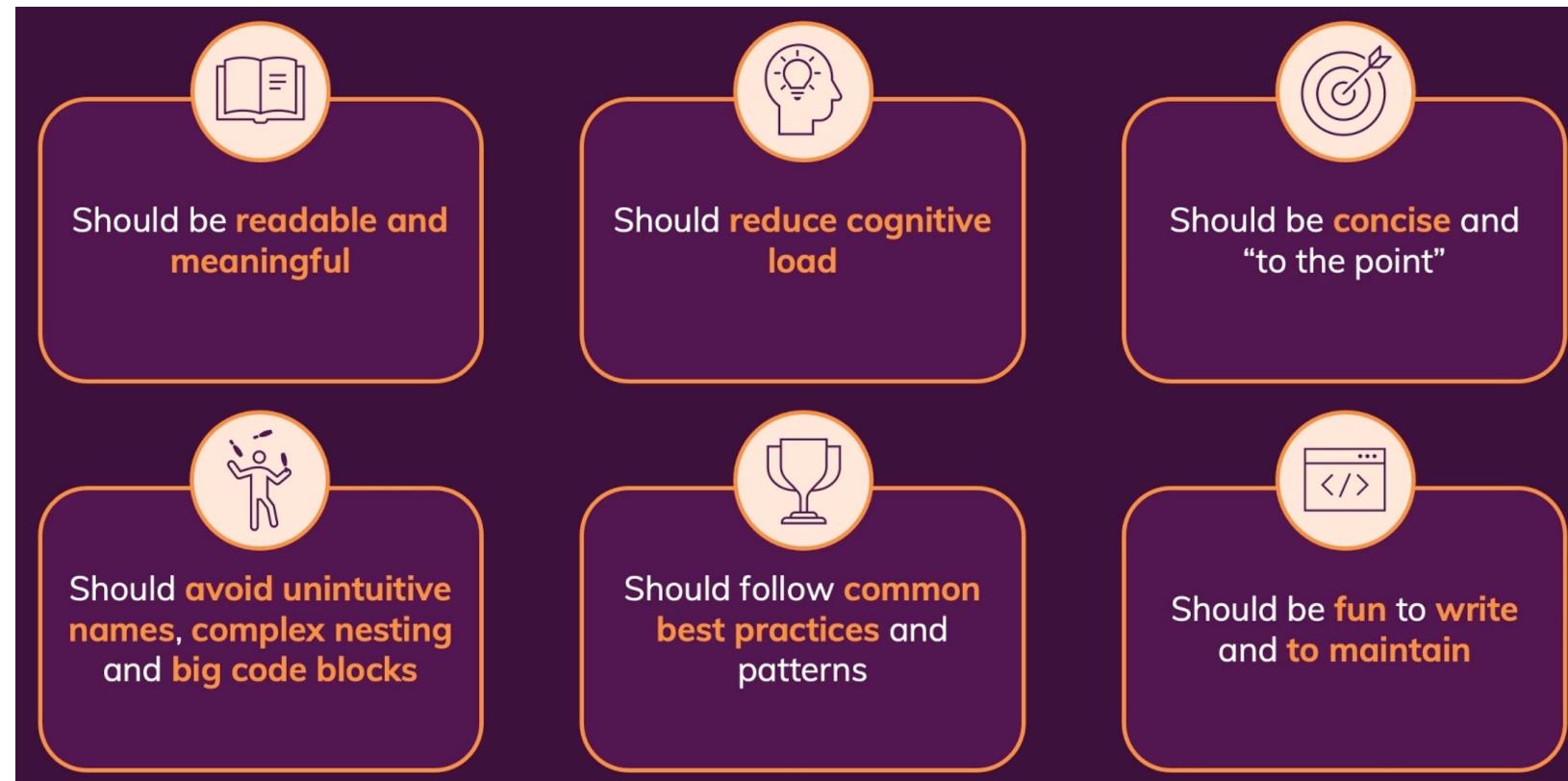
    if(Multitouch.supportsGestureEvents){
        trace("mlutitouch?")
        //trace("seriosuly?mt?")
        // ttrace("multitouch WTF?!?!")
        // trace("some errorr jere");
        Multitouch.inputMode = "gesture";
        addEventListener("gestureZoom", callback3);
    }
    // swfaddress is so hard! LOL!!111
    SWFAddress.addEventListener(SWFAddressEvent.INIT,myCallk
    SWFAddress.addEventListener(SWFAddressEvent.CHANGE,mySwf
```

클린 코드 정의

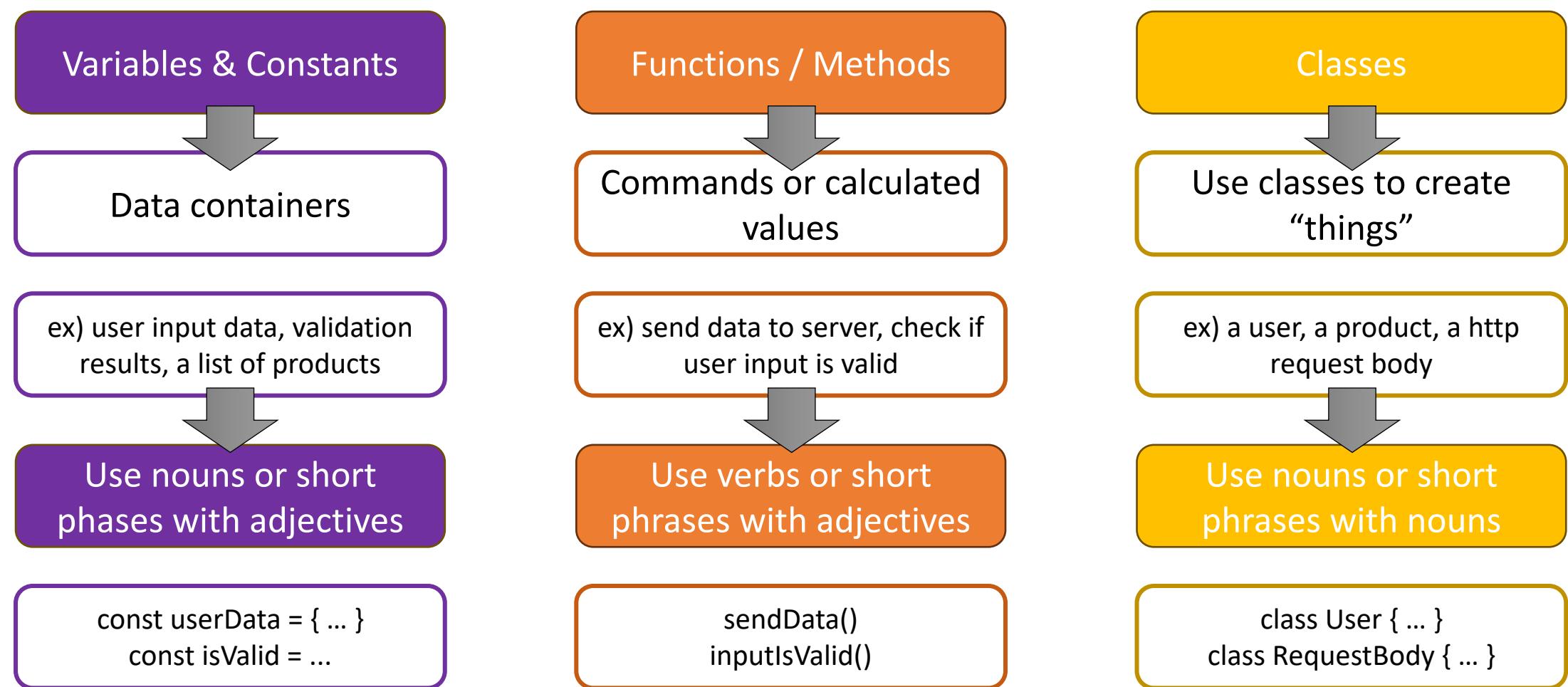
- SW를 보다 쉽게 이해할 수 있고, 적은 비용으로 수정할 수 있도록
겉으로 보이는 **동작의 변화 없이 내부 구조를 변경**하는 것
- 일련의 **리펙토링을 적용**하여 겉으로 보이는 동작의 변화 없이
소프트웨어의 구조를 바꾸는 것
- 다음 방법으로 동작의 변경 여부 검증
 - 테스팅
 - 최대한 개발Tool의 지원 기능 사용
 - 아주 조심히 진행

클린 코드 정의

- ***It's not about whether code works.***
- ***A vast majority of time is spent reading and understanding code.***
- ***Clean Code is Easy To Understand – Dirty Code Is Not***



1. 의미 있는 이름



1. 의미 있는 이름

이모지(Emoji)를 이름으로 사용한다면? (Swift)

```
class 💩💩💩💩
{
    func 💩💩💩💩(😎: Int, 🐱: Int) -> Int
    {
        return 😎 + 🐱
    }
}

var 🐔 = 3
var 😢 = 🐔 + 2

var 💩 = 💩💩💩💩()
println(💩.💩💩💩(🐔, 🐱:😢))
```

```
class 🐶🐶
{
    func 🐥😎(😎: Int, 🐱: Int) -> Int
    {
        return 😎 + 🐱
    }
}

var 🐔 = 3
var 😢 = 🐔 + 2

var 🐶 = 🐶🐶()
println(🐶.🐶🐶(🐔, 😢))
```

1. 의미 있는 이름

1) 의도가 분명하게 이름을 지으라

- 변수(또는 함수나 클래스)의 존재 이유는?
- 수행 기능은?
- 사용 방법은?

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

1. 의미 있는 이름

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

1. theList에 무엇이 들었는가?
2. theList에서 0번째 값이 어째서 중요한가?
3. 값 4는 무슨 의미인가?
4. 함수가 반환하는 리스트 list1을 어떻게 사용하는가?

1. 의미 있는 이름

- 지뢰찾기 게임을 만든다고 가정
- theList는 게임판 → gameBoard로 변경
- 게임판에서 각 칸은 단순 배열로 표현
- 배열에서 0번째 값은 칸 상태를 뜻함
- 값 4는 깃발이 꽂힌 상태

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();

    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);

    return flaggedCells;
}
```

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();

    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);

    return flaggedCells;
}
```

1. 의미 있는 이름

2) 의미 있게 구분

```
getActiveAccont();
getActiveAccounts();
getActiveAccontInfo();
```

3) 발음하기 쉬운 이름을 사용

- genymdhms
- “젠 와이 엠 디 에이취 엠 에스”
- “젠 야 무다 힘즈”

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";

    /* ... */
}

class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;
    private final String recodeId = "102";

    /* ... */
}
```

1. 의미 있는 이름

4) 클래스 이름

- 명사나 명사구가 적합, 동사는 피하도록
- **good ex)** Customer, WikiPage, Account, AddressParser
- **bad ex)** Manager, Processor, Data, Info

5) 메소드 이름

- 동사나 동사구
- ex) postPayment, deletePage, save
- 접근자, 변경자 조건자는 get, set, is를 붙인다.
- 생성자를 중복정의(Overload) 할 때는 정적 팩토리 메소드 사용

```
Complex fullcrumPoint = Complex.FromRealNumber(23.0);
```

vs.

```
Complex fullcrumPoint = new Complex(23.0);
```

1. 의미 있는 이름

6) 의미 있는 맥락을 추가하라

```
private void printGuessStatistics(char candidate, int count) {  
    String number;  
    String verb;  
    String pluralModifier;  
  
    if (count == 0) {  
        number = "no";  
        verb = "are";  
        pluralModifier = "s";  
    } else if (count == 1) {  
        number = "1";  
        verb = "is";  
        pluralModifier = "";  
    } else {  
        number = Integer.toString(count);  
        verb = "are";  
        pluralModifier = "s";  
    }  
  
    String guessMessage = String.format(  
        "There %s %s %s%s", verb, number, candidate, pluralModifier  
    );  
  
    print(guessMessage);  
}
```

1. 의미 있는 이름

6) 의미 있는 맥락을 추가하라

```
public class GuessStatisticsMessage {  
    private String number;  
    private String verb;  
    private String pluraModifier;  
  
    public String make(char candidate, int count) {  
        createPluraDependentMessageParts(count);  
        return String.format(  
            "There %s %s %s%s",  
            verb, number, candidate, pluraModifier  
        );  
    }  
  
    private void createPluraDependentMessageParts(int count) {  
        if (count == 0) {  
            thereAreNoLetters();  
        } else if (count == 1) {  
            thereIsOneLetter();  
        } else {  
            thereAreManyLetters(count);  
        }  
    }  
  
    private void thereAreManyLetters(int count) {  
        number = Integer.toString(count);  
        verb = "are";  
        pluraModifier = "s";  
    }  
  
    private void thereIsOneLetter() {  
        number = "1";  
        verb = "is";  
        pluraModifier = "";  
    }  
  
    private void thereAreNoLetters() {  
        number = "no";  
        verb = "are";  
        pluraModifier = "s";  
    }  
}
```

1. 의미 있는 이름

7) Name Casing

snake_case

camelCase

PascalCase

kebab-case

is_valid
send_response

isValid
sendResponse

AdminRole
UserRepository

<side-drawer>

ex) Python

ex) Java, Javascript

ex) Python, Java, Javascript

ex) HTML

Variables, functions,
methods

Variables, functions,
methods

Classes

Custom HTML Elements

1. 의미 있는 이름

8) Examples

What is stored?

Bad Names

Okay Names

Good Names

A user object (name,
email, age)

u
data

userData
person

user
customer

“u” and “data” could contain
anything

“userData” is a bit redundant,
“person” is too unspecific

“user” is descriptive,
“customer” is even more
specific

User input validation
result (true/false)

v
val

correct
validatedInput

isCorrect
isValid

“v” could be anything, “val”
could also stand for “value”

Both terms don’t
necessarily imply a
true/false value

Descriptive and value type
is clear

2. 함수

```
buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
    WikiPage teardown = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
        WikiPagePath teardownPath = wikiPage.getPageCrawler().getFullPath(teardown);
        String teardownPathName = PathParser.render(teardownPath);
        buffer.append("\n")
            .append("!include -teardown .")
            .append(teardownPathName)
            .append("\n");
    }

    if (includeSuiteSetup) {
        WikiPage suiteTeardown = PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE_TEARDOWN_NAME, wikiPage);
        if (suiteTeardown != null) {
            WikiPagePath pagePath = wikiPage.getPageCrawler().getFullPath(suiteTeardown);
            String pagePathName = PathParser.render(pagePath);
            buffer.append("!include -teardown .")
                .append(pagePathName)
                .append("\n");
        }
    }
}

pageData.setContent(buffer.toString());
return pageData.getHtml();
}
```

2. 함수

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup = PageCrawlerImpl.getInheritedPage(SuiteResponse.SUITE_SETUP_NAME, wikiPage);
            if (suiteSetup != null) {
                WikiPagePath pagePath = suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName)
                    .append("\n");
            }
        }
        WikiPage setup = PageCrawlerImpl.getInheritedPage("Setup", wikiPage);
        if (setup != null) {
            WikiPagePath setupPath = wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append("!include -setup .")
                .append(setupPathName)
                .append("\n");
        }
    }
}
```

2. 함수

```
public static String renderPageWithSetupsAndTeardown(
    PageData pageData, boolean isSuite
) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }

    return pageData.getHtml();
}
```

- 설정(setup) 페이지와 해제(teardown) 페이지를 테스트 페이지에 넣어 후 해당 테스트 페이지를 HTML로 렌더링하는 함수

2. 함수

1) 작게 만들어라!

```
public static String renderPageWithSetupsAndTeardown(
    PageData pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);

    return pageData.getHtml();
}
```

2. 함수

2) 한 가지만 해라

- 함수는 한 가지를 해야 한다. 그 한 가지를 잘 해야 한다. 그 한 가지만을 해야 한다.

1. 페이지가 테스트 페이지인지 판단한다.
2. 그렇다면 설정 페이지와 해제 페이지를 넣는다.
3. 페이지를 HTML로 렌더링한다.

`renderPageWithSetupsAndTeardowns`, 페이지가 테스트 페이지인지 확인한 후 테스트 페이지라면 설정 페이지와 해제 페이지를 넣는다. 테스트 페이지든 아니든 페이지를 HTML로 렌더링한다.

2. 함수

3) Switch 문

- Switch 문을 작게 만들기는 어렵지만, 다형성을 이용하여 Switch 문을 저차원 클래스에 숨기고 반복하지 않도록 한다.

```
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:
            new InvalidEmployeeType(e.type);
    }
}
```

1. 함수가 길다.
2. 한 가지 작업만 수행하지 않는다.
3. 동일한 함수가 무한정 존재할 수 있다.

2. 함수

3) Switch 문

```
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:  
                return new InvalidEmployeeType(r.type);  
        }  
    }  
}
```

2. 함수

4) 서술적인 이름을 사용하라!

- 길고 서술적인 이름이 짧고 어려운 이름보다 좋다.
- 이름을 붙일 때 일관성이 있어야 한다.
- ex) includeSetupAndTeardownPages, includeSetupPages, includeSuiteSetupPage 등

5) 함수 인수

- 함수에서 **이상적인 인수 개수는 0개**이다
- 3개는 가능한 피하는 편이 좋다.
- 4개 이상은 특별한 이유가 필요하다.
- **테스트 관점에서 인수가 없는 것이 훨씬 용이**하다.

3. 변수

변수는 내버려두면 계속 늘어난다

- 변수를 덜 사용하고 최대한 '**가볍게**' 만들어 가독성을 높인다.
- 실천 방안
 - 방해되는 변수를 제거: 결과를 즉시 처리하는 방식으로, 중간 결과값을 저장하는 변수를 제거
 - 각 변수의 범위를 최대한 작게 줄임: 각 변수 위치를 옮겨 변수가 나타나는 줄의 수를 최소화 → 안 보이면 멀어진다.
 - 값이 한 번만 할당되는 변수를 선호: const, final 등으로 값이 한 번만 할당되어 바뀌지 않는 변수는 이해하기 쉽다.

4. 코드 흐름 제어

코드 흐름을 읽기 쉽게 만들어야 논리가 명확해진다.

- 실천 방안
 - if/else 문의 블록 순서에 주의: 긍정적이고 쉽고 흥미로운 경우가 앞쪽에 위치
 - 삼항연산자 (? :)나 do/while, goto는 코드 가독성을 떨어뜨리므로 되도록 사용 금지
 - 중첩된 코드 블록의 흐름을 따라가려면 많은 집중이 필요 → 선형적인 코드 추가
 - 함수 중간에 반환하면 중첩을 피하고 코드를 더 깔끔하게 작성 가능 → 함수 앞부분에서 '보호 구문'으로 간단한 경우를 미리 처리하는 방안도 고려

5. 주석

- 1) 나쁜 코드에 주석을 달리 마라. 새로 짜라.
- 2) 잡음을 줄이고 의도를 명확하게
- 3) 가능하면 코드로 의도를 표현하라!

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

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

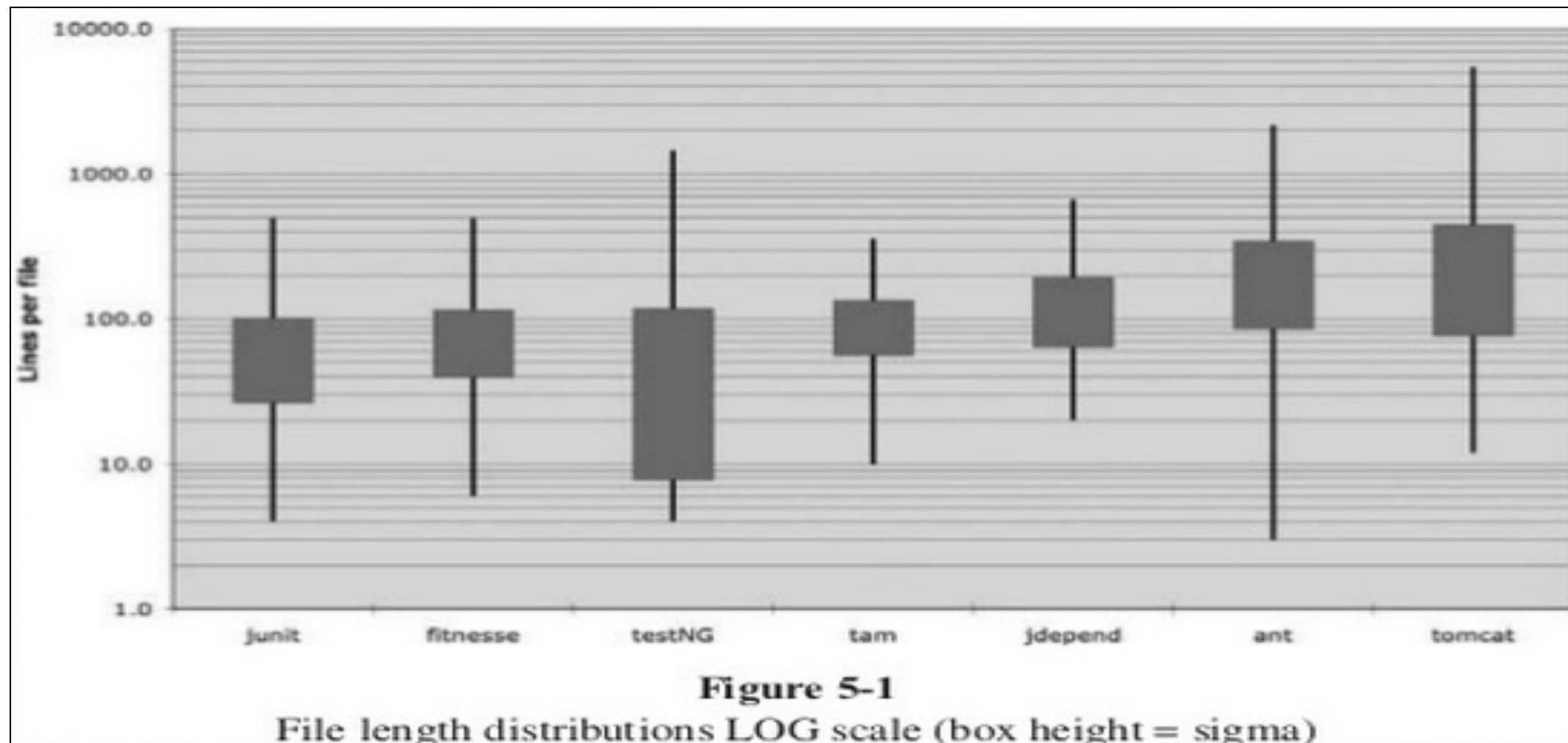
5. 주석

- 좋은 주석
 - 법적인 주석
 - 정보를 제공하는 주석
 - 의도를 설명하는 주석
 - 의미를 명료하게 밝혀주는 주석
 - 결과를 경고하는 주석
 - TODO 주석
 - 중요성을 강조하는 주석
- 나쁜 주석
 - 주절거리는 주석
 - 같은 이야기를 중복하는 주석
 - 오해할 여지가 있는 주석
 - 의무적으로 다는 주석
 - 이력을 기록하는 주석
 - 있으나 마나 한 주석
 - 공로를 돌리거나 저자를 표시하는 주석
 - 주석으로 처리한 코드

```
assertTrue(a.compareTo(b) != 0); // a != b
```

6. 형식 맞추기

1) 파일 길이는 짧아야 한다(ex. 유명 오픈소스 코드 통계)



6. 형식 맞추기

2) 개념은 빈행으로 분리하라

```
package fitnessse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+";
    private static final Pattern pattern = Pattern.compile(
        Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

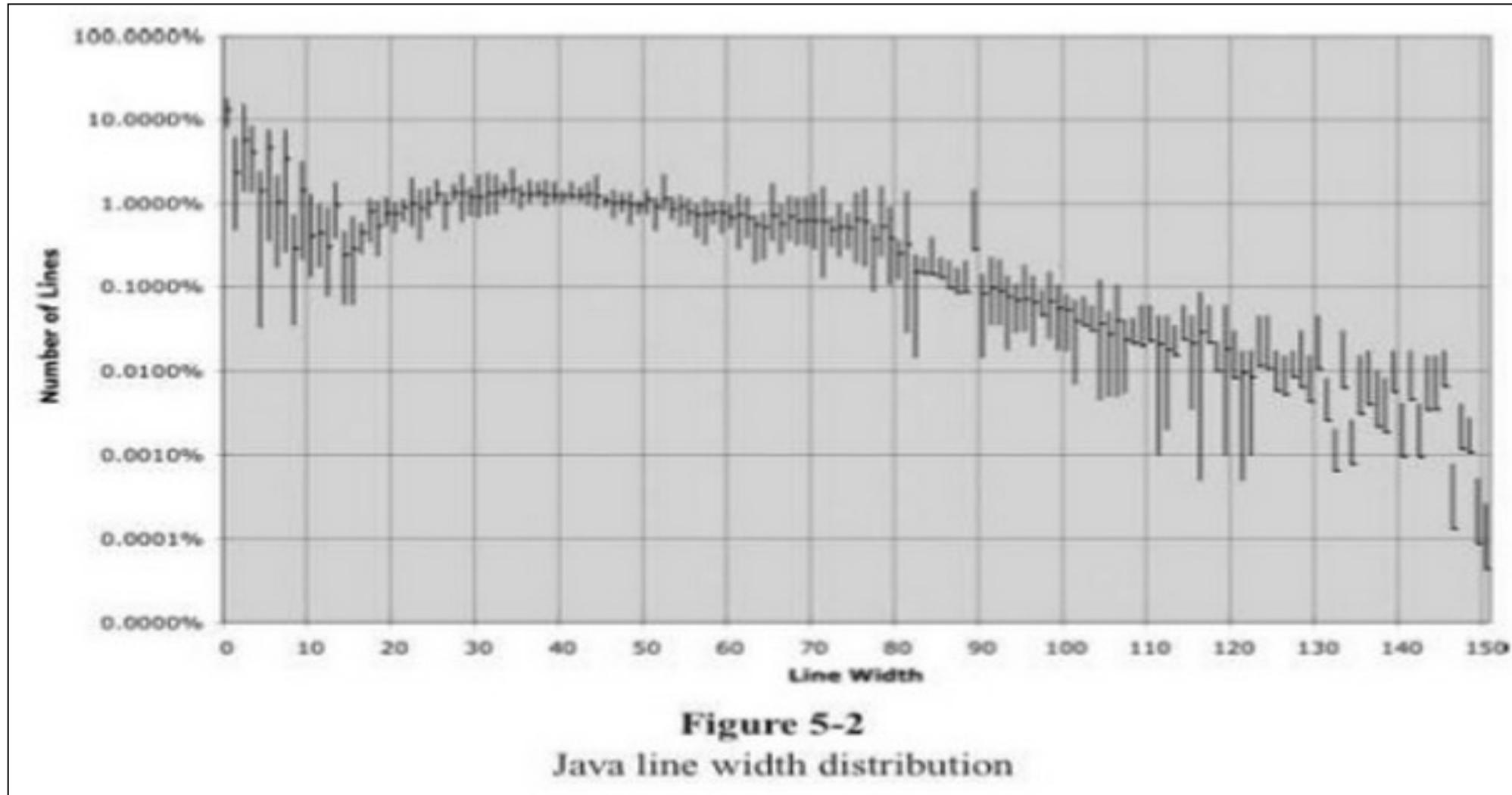
```
package fitnessse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?";
    private static final Pattern pattern = Pattern.compile(
        Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

3) 서로 밀접도

- 서로 밀접한 코드 행은 세로로 가까이 놓도록 한다.

6. 형식 맞추기

4) 행 길이도 짧아야 한다(ex. 유명 오픈소스 코드 통계)



6. 형식 맞추기

5) 미학적인 요소 고려

- 여러 블록에 담긴 코드가 모두 비슷한 일을 하면 실루엣이 동일해 보이게 구성
- 코드 곳곳을 ‘열’로 만들어 줄을 맞추면 코드를 한 눈에 훑어보기 편하게 구성
- 코드 한 곳에서 A, B, C 순으로 언급되면, 다른 곳에서 B, A, C 순으로 언급하지 마라. **의미있는 순서**를 정해 지켜라.
- 빈줄을 이용해 커다란 블록을 논리적인 ‘문단’으로 나눠라.

7. 코드 분량 줄이기

가급적이면 적은 코드를 유지

- 가장 읽기 쉬운 코드는 아무것도 없는 코드!
- 테스트/유지보수/문서화에 유리
- 실천 방안
 - 제품에 필요하지 않는 기능은 제거 (**주석 처리 금물!**)
 - 요구 사항을 재고해서 가장 단순한 형태의 문제를 탐색
 - 주기적으로 프레임워크/라이브러리의 전체 클래스/API를훑어보고 표준으로 제공하는 기능에 친숙해짐

8. 절차적인 코드와 객체 지향 코드

객체 지향이 만능인가?

- 객체는 동작을 공개하고 자료를 숨긴다. 그래서 기존 동작을 변경하지 않으면서 새 객체 타입을 추가하기는 쉬운 반면, 기존 객체에 새동작을 추가하기는 어렵다. 자료 구조는 별다른 동작 없이 자료를 노출한다. 그래서 기존 자료 구조에 새 동작을 추가하기는 쉬우나, 기존 함수에 새 자료 구조를 추가하기는 어렵다.
- 절차적인 코드는 새로운 자료 구조를 추가하기 어렵다. 그러면 모든 함수를 고쳐야 한다. 객체 지향 코드는 새로운 함수를 추가하기 어렵다. 그러면 모든 클래스를 고쳐야 한다.

8. 절차적인 코드와 객체 지향 코드

객체 지향이 만능인가?

- (어떤) 시스템을 구현할 때, 새로운 자료 타입을 추가하는 유연성이 필요하면
객체가 더 적합하다. 다른 경우로 새로운 동작을 추가하는 유연성이 필요하면
자료 구조와 절차적인 코드가 더 적합하다. 우수한 소프트웨어 개발자는 편견
없이 이 사실을 이해해 직면한 문제에 최적인 해결책을 선택한다.

9. 오류 처리

- **Null의 의미는 두 가지**
 - 정말 없는 것
 - 없다고 표현한 것
- **모호성으로 인한 문제점은 셀 수 없다**
 - Null을 반환하거나 Null을 전달하지 마라!
 - 정말 Null이 필요하면 optional을 고려

```
public class BasicTest {  
    @Test  
    public void optionalAbsentTest() throws Exception {  
        Optional absent = Optional.absent();  
        System.out.println("Is absent present? " + absent.isPresent());  
        System.out.println("absent is " + absent.get());  
    }  
}
```

10. 테스트

가독성이 가장 중요한 위치를 차지

- 테스트가 읽기 편해야 유지보수/확장도 쉬워진다.
- 실천 방안
 - 각 테스트의 최상위 수준은 최대한 간결하게 유지
 - 테스트에 실패하면, 버그를 추적해 수정하게 돋는 오류 메시지를 출력해야 함
 - 코드의 구석구석을 철저하게 실행하는 가장 간단한 입력을 사용
 - 무엇이 테스트되는지 분명하게 드러나게 테스트 함수에 충분한 설명이 포함된 이름을 부여
 - 무엇보다 테스트의 수정이나 추가가 쉬어야 한다!

10. 테스트

F.I.R.S.T

- Fast: 테스트는 빨라야 한다.
- Independent: 각 테스트는 서로 의존하면 안 된다. 한 테스트가 다음 테스트가 실행될 환경을 준비해서는 안 된다.
- Repeatable: 테스트는 어떤 환경에서도 반복 가능해야 한다.
- Self-Validation: 테스트는 부울 값으로 결과를 내야 한다. 성공 아니면 실패
- Timely: 테스트는 적시에 작성해야 한다. 단위 테스트는 테스트하려는 실제 코드를 구현하기 직전에 구현한다.

11. 클래스

- SRP, Single Responsibility Principle
- 클래스 체계
 - 정적 공개 상수
 - 정적 비공개 상수
 - 비공개 인스턴스 변수
 - 공개 함수
 - 비공개 함수는 자신을 사용하는 공개 함수 직후에
- 클래스는 작아야 한다!
 - 단일 책임 원칙

12. 시스템

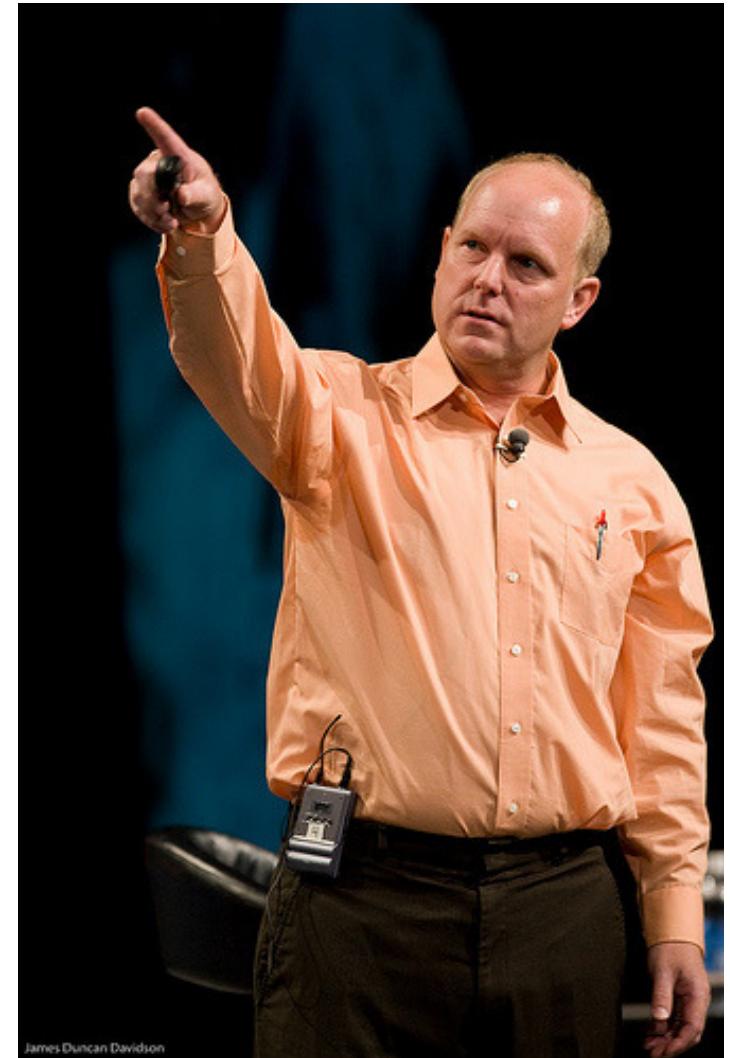
생성과 사용의 분리

- 팩토리
- 의존성 주입 (DI)과 제어 역전(IoC) ex) 스프링 프레임워크
- 생각해볼 문제
 - “처음부터 올바르게” 시스템을 만들 수 있다는 믿음은 미신이다. 대신에 우리는 오늘 **주어진 사용자 스토리에 맞춰 시스템을 구현**해야 한다.
 - 소프트웨어 부문에서 가장 큰 악성 재고는 **개발자의 낭비되는 시간**이다.
 - 소프트웨어 시스템은 물리적인 시스템과 다르다. **관심사를 적절히 분리**해 관리한다면 소프트웨어 아키텍처는 점진적으로 발전할 수 있다.
 - 소프트웨어 시스템은 “**수명이 짧다**”는 본질로 인해 아키텍처의 점진적인 발전이 가능하다.

13. 설계

Kent Beck 의 간단한 설계 규칙 4가지

- 1. 모든 테스트를 실행한다.**
- 2. 중복을 없앤다.**
- 3. 프로그래머 의도를 표현한다.**
- 4. 클래스와 메소드 수를 최소로 줄인다.**

James Duncan Davidson

14. 동시성

동시성 개괄

- 동시성의 의의
 - 동시성은 결합(Coupling)을 없애는 전략이다.
 - 즉, 무엇(What)과 언제(When)를 분리하는 전략이다.
- 동시성의 장점
 - 무엇과 언제를 분리하면 응용 프로그램 구조와 효율이 극적으로 나아진다.
 - 구조적인 관점에서 프로그램은 거대한 루프 하나가 아니라 작은 협력 프로그램 여럿으로 보인다.
 - 따라서 시스템을 이해하기가 쉬워지며 문제를 분리하기도 쉬워진다.

14. 동시성

동시성의 오해와 문제점

- 동시성의 오해
 - 동시성은 항상 성능을 높여준다.
 - 동시성을 구현해도 설계는 변하지 않는다.
 - 적절한 도구를 사용하면 동시성을 이해할 필요가 없다.
- 동시성의 문제점
 - 동시성은 다소 부하를 유발한다. 성능 측면에서 부하가 걸리며, 코드도 더 짜야 한다.
 - 동시성은 복잡하다. 간단한 문제라도 동시성은 복잡하다.
 - 일반적으로 동시성 버그는 재현하기 어렵다. 그래서 일회성 문제로 여겨 무시하기 쉽다.
진짜 결함으로 간주되지 않는다.
 - 동시성을 구현하려면 흔히 근본적인 설계 전략을 재고해야 한다.

14. 동시성

구체적인 지침

- 말이 안 되는 실패는 잠정적인 스레드 문제로 취급하다.
- 다중 스레드를 고려하지 않은 순차 코드부터 제대로 돌게 만들자.
- 다중 스레드를 쓰는 코드 부분을 쉽게 다양한 환경에 끼워 넣을 수 있게 스레드 코드를 구현하라.
- 다중 스레드를 쓰는 코드 부분을 상황에 맞춰 조정할 수 있게 작성하라.
- 프로세스 수보다 많은 스레드를 돌려보라.
- 다른 플랫폼에서 돌려보라.
- 코드에 보조 코드(*Instrument*)를 넣어 돌려라. 강제로 실패를 일으키게 해보라.

15. 점진적인 개선

1) 프로그래밍은 과학보다 공예에 가깝다.

- 퇴고가 없는 작문은 존재하지 않는다. 즉, 깨끗한 코드를 짜려면 먼저 지저분한 코드를 짠 뒤에 정리해야 한다는 의미이다.

2) 자살행위: 초보자는 일단 프로그램이 “돌아가면” 다음 업무로 넘어간다. “돌아가는” 프로그램은 그 상태가 어떻든 그대로 버려둔다. 경험이 풍부한 전무 프로그래머라면 이런 행동이 전문가로서 자살 행위라는 사실을 잘 안다.

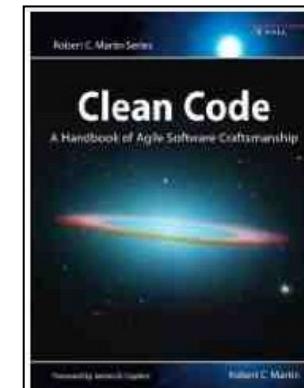
3) 1년 전 코드보다 5분 전 코드가 정리하기가 편하다!

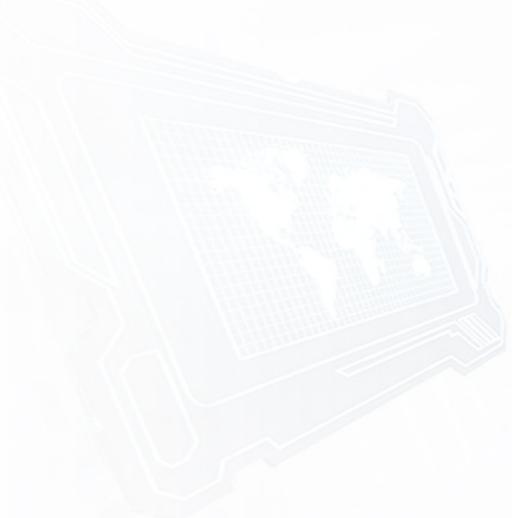
보이스카우트 규칙

“Leave the campground cleaner than you found it.”

(캠프장에 처음 왔을 때 보다 더 깨끗하게 해놓고 떠나라.)

전문가 정신 = 지속적인 개선





Code Smell

언제 리팩토링을 해야 하는가?

- **틈틈이 계속**
 - 리팩토링 자체가 목적이 아니라, 다른 것을 하기 위해 리팩토링을 하는 것이고, 리팩토링은 그 다른 것을 하는데 도움을 준다.
- **삼진 규칙**
 - 세 번째로 비슷한 것을 하게 되면 리팩토링을 한다.
- **기능을 추가할 때, 버그를 수정할 때**
 - 코드에 대한 이해를 돋기 위해서
- **코드 검토 시**
 - 고수준의 의견을 얻을 수 있음
 - 준비가 많이 필요함

리팩토링 할 때의 문제

- 데이터베이스와 연관된 코드의 리팩토링
- 인터페이스의 변경
 - Publish 된 인터페이스의 이름 및 파라미터의 대한 리팩토링
 - 새로운 I/F를 만들고 Delegation 사용
 - 예방책: 애매한 메소드는 API에 넣지 말자
 - Publish 된 인터페이스의 Throws 절의 변경
 - Delegation 사용 불가
 - 예방책: Super Exception 사용

리팩토링 할 때의 문제

- **리팩토링이 어려운 디자인 변경**
 - 디자인에 실수가 있어 마음대로 리팩토링 할 수 없을 때
 - 어떤 디자인 결정 사항이 너무 중요해서 리팩토링을 기대할 수 없는 경우
 - 보안 문제, 퍼포먼스 문제
- **언제 리팩토링을 하지 말아야 하는가?**
 - 코드를 처음부터 작성하는 게 나을 정도로 엉망인 경우
 - 현재의 코드가 작동하지 않을 경우
 - 마감일에 가까울 경우

Code smell

- 코드에 잠재된 문제에 대한 경고
- 리팩토링을 해야 하는 문제점 표시

- 대표적인 *Code smell*

- 중복코드
- 부적절한 이름
- 기능에 대한 욕심
- 부적절한 친밀
- 주석
- 긴 메소드
- 긴 파라미터 리스트
- Switch 문

Code smell - 중복 코드

- 가장 많이 발생하는 *Code smell*
- 2가지 종류의 중복
 - Obvious 중복
 - 보통 Copy & Paste에 의해 발생
 - Unobvious 중복
 - 계층구조의 병렬적 구현
 - 비슷한 알고리즘 (ex, 문자열, 도메인 특화 로직)
- 해결책
 - 하나의 클래스에서 중복
 - ExtractMethod
 - 두 형제 클래스에서 중복
 - ExtractMethod → Pull Up Field or Pull Up Method → Form Template Method

Code smell - 부적절한 이름

- 잘못 된 이름은 코드의 이해를 방해한다.

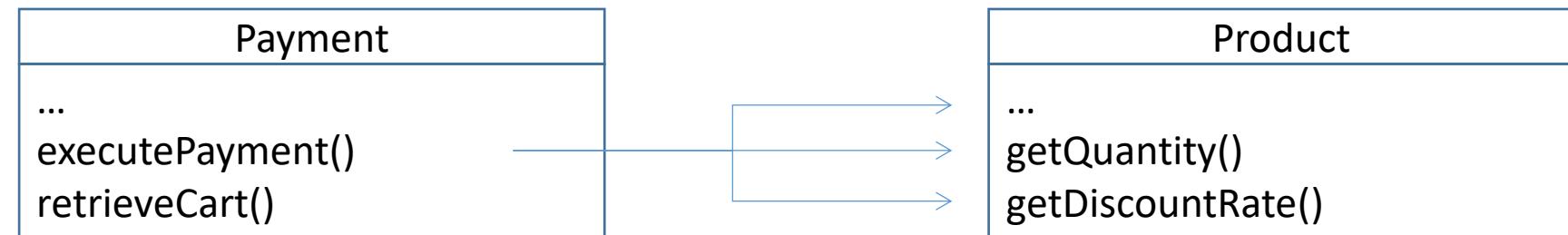
“People often make assumptions based on the object names alone”

- Ward Cunningham

- 해결책
 - Rename Method
 - Rename Field
 - Rename Constants
- Tip !
 - 일관성 없는 상세이름 보다는 일관성 있는 광의의 이름이 적합
 - add, register, put, create → add
 - 일관성 있는 이름을 위해서 용어 사전은 필수

Code smell - 기능에 대한 욕심

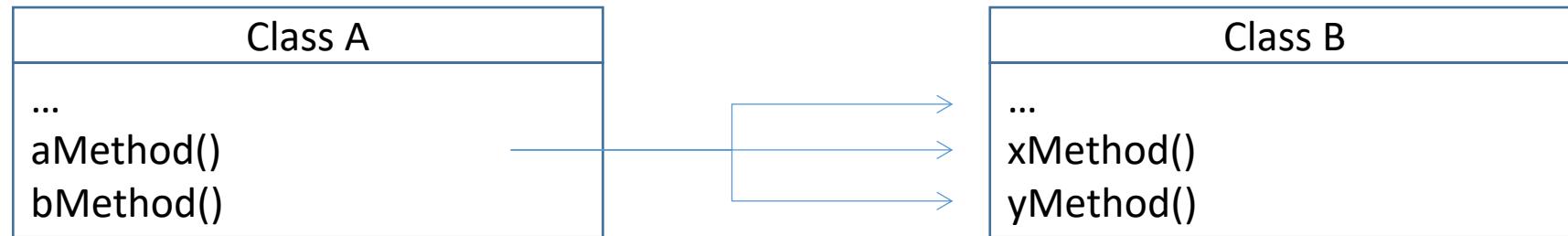
- 특정 클래스 내의 메소드가 동작을 위해 다른 클래스에 있는 정보를 많이 필요로 하는 경우



- 해결책
 - Move Method

Code smell - 부적절한 친밀

- 특정 클래스 내의 메소드가 다른 클래스에 있는 메소드 (**Private 이어야 할**)를 더 많이 사용하는 경우



- **해결책**
 - 두 개의 독립적인 클래스가 엉켜 있다면
 - Move Method
 - 서로를 가리키고 있다면
 - Change Bidirectional Reference to Unidirectional
 - 중재하는 클래스가 없기 때문에 엉켜 있다면
 - Extract Hierarchy → Hide Delegate

Code smell - 주석

- 주석은 대부분 코드가 명확하지 않다고 판단 될 때 쓰임
- 주석은 나쁜 코드를 유도함
- 해결책
 - 코드의 일정 블록 설명
 - Extract Method
 - 메소드가 하는 일 설명
 - Rename Method
 - 선 조건 설명
 - Introduce Assertion

Code smell - 긴 메소드

- 긴 메소드는 코드를 이해하기 힘들게 만든다.
- 주석을 달아야 할 필요를 느낄 때마다 메소드로 분리
- 해결책
 - Extract Method
- 주의점
 - 코드의 이해를 돋기 위해서는 코드의 의도를 잘 나타내는 이름으로 명명
 - 부적절한 이름을 가진 짧은 메소드 집합은 긴 메소드보다 이해하기 힘들다.

Code smell - 긴 파라미터 리스트

- 객체지향 코드에서는 파라미터가 많을 필요가 없다.
- 문제점
 - 이해하기 어려움
 - 일관성 없음
 - 변경이 많이 됨
- 해결책
 - 이미 알고 있는 다른 객체로부터 값 얻어올 수 있음
 - Replace Parameter with Method
 - 매개변수가 하나의 Object로부터 나오면
 - Preserve Whole Object
 - 매개변수 데이터가 하나의 논리 객체로 부터 얻어지는게 아닐 경우
 - Introduce Parameter Object
- 주의점
 - Caller와 Callee간의 데이터 종속성을 만들고 싶지 않을 경우 리펙토링을 피해야 함

Code smell - Switch 문

- **Switch 문은 본질적으로 중복의 문제를 발생**
 - 타입 추가 시, 관련된 모든 Switch문을 수정해야 함
- **좋은 OOP 코드의 특징은 Switch 문이 비교적 적음**
- **해결책**
 - 동일한 조건에 대한 Switch문이 여러 곳에서 사용됨
 - 1) Step1: Extract Method를 통해 각 조건 절에서 코드 추출
 - 2) Step2: Move Method를 통해 연관된 코드를 옮바른 클래스로 옮김
 - 3) Step3: Replace Type Code with Subclass or Replace Type Code with State/Strategy 를 통해 상속구조 만듬
- **주의점**
 - Switch문의 중복이 많이 않다면 바꿀 필요 없음

Example 1 - 함수

```
public bool isEdible() {  
    if (this.ExpirationDate > Date.Now &&  
        this.ApprovedForConsumption == true &&  
        this.InspectorId != null) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

How many things is the function doing?

Example 1 - 함수

```
public bool isEdible() {  
    return isFresh() &&  
        isApproved() &&  
        isInspected();  
}
```

Now the function is doing one things!

Example 2 - 예외

```
public int foo(){  
    ...  
}  
  
public void bar(){  
    if(foo() == OK)  
        ...  
    else  
        // error handling  
}
```

Errors have to be encoded
Checks (when performed) require a lot of code
It's harder to extend such programs

Example 2 - 예외

```
public void foo()  
    throws FooException{  
    ...  
}  
  
public void bar(){  
    try{  
        foo();  
        ...  
    } catch(FooException){  
        // error handling  
    }  
}
```

*No need to mix return values
and control values*

Cleaner syntax

Easier to extend

Example 3 - 반복1

```
public void bar(){  
    foo("A");  
    foo("B");  
    foo("C");  
}
```

DO NOT EVER COPY AND PASTE CODE

Example 3 - 반복1

```
public void bar(){  
    String [] elements = {"A", "B", "C"};  
  
    for(String element : elements){  
        foo(element);  
    }  
}
```

*Logic to handle the elements
it's written once for all*

Example 4 - 절차적인 코드

```

public class Square {
    public Point toLeft;
    public double side;
}

public class Rectangle {
    public Point toLeft;
    public double height;
    public double width;
}

public class Circle {
    public Point center;
    public double radius;
}

```

```

public class Geometry {
    public final double PI = 3.141592653589793;

    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;
        }

        throw new NoSuchShapeException();
    }
}

```

- *Geometry* 클래스에 둘레 길이는 구하는 *perimeter()* 메소드를 추가한다면?
 → 도형 클래스는 아무 영향도 받지 않는다!
- 새 도형을 추가하고 쉽다면?
 → *Geometry* 클래스에 속한 함수를 모두 고쳐야 한다!

Example 4 - 객체지향 코드

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

    public double area() {
        return side * side;
    }
}
```

```
public class Rectangle implements Shape {
    private Point toLeft;
    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 = 3.141592653589793;

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

- 새 도형을 추가해도 기존 메소드에 아무런 영향을 미치지 않는다.
- 새 메소드를 추가하고 싶다면 도형 클래스 전부를 고쳐야 한다.

Example 5 - 단위 테스트

```
@Test  
public void turnOnLoTempAlarmAtThreshold() throws Exception {  
    hw.setTemp(WAY_TOO_COLD);  
    controller.tic();  
    assertTrue(hw.heaterState());  
    assertTrue(hw.blowerState());  
    assertFalse(hw.coolerState());  
    assertFalse(hw.hiTempAlarm());  
    assertTrue(hw.loempAlarm());  
}
```

heaterState 상태 확인 후 assertTrue를 읽는다.

Example 5 - 단위 테스트

```

@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchl", hw.getState());
}

@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
}

@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBchl", hw.getState());
}

@Test
public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHl", hw.getState());
}

```

대문자는 '켜짐', 소문자는 '꺼짐'
문자는 {heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}

```

public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
}

```

Example 6 - 반복2

```
public void scaleToOneDimension (float desiredDimention, float imageDimension) {  
    if (Math.abs(desireDimension - imageDimension) < errorThreshold)  
        return;  
  
    float scalingFactor = desireDimension / imageDimension;  
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);  
  
    RenderedOp newImage = ImageUtilities.getScaledImage(  
        image, scalingFactor, scalingFactor);  
    image.dispose();  
    System.gc();  
    image = newImage;  
}  
  
public synchronized void rotate(int degrees) {  
    RenderedOp newImage = ImageUtilities.getScaledImage(  
        image, degrees);  
    image.dispose();  
    System.gc();  
    image = newImage;  
}
```

Example 6 - 반복2

```
public void scaleToOneDimension (float desiredDimention, float imageDimension) {  
    if (Math.abs(desireDimension - imageDimension) < errorThreshold)  
        return;  
  
    float scalingFactor = desireDimension / imageDimension;  
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);  
  
    replaceImage(ImageUtilities.getScaledImage(image, scalingFactor, scalingFactor));  
}  
  
public synchronized void rotate(int degrees) {  
    replaceImage(ImageUtilities.getScaledImage(image, degrees));  
}  
  
private void replaceImage(RenderedOp newImage) {  
    image.dispose();  
    System.gc();  
    image = newImage;  
}
```

Example 7 - 반복3

```
public class VacaionPolicy {  
    public void accrueUSDivisionVacation() {  
        // 지금까지 근무한 시간을 바탕으로 휴가 일수를 계산하는 코드  
        // ...  
        // 휴가 일수가 미국 최소 법정 일수를 만족하는지 확인하는 코드  
        // ...  
        // 휴가 일수를 금여 대장에 적용하는 코드  
        // ...  
    }  
  
    public void accrueEUDivisionVacation() {  
        // 지금까지 근무한 시간을 바탕으로 휴가 일수를 계산하는 코드  
        // ...  
        // 휴가 일수가 유럽연합 최소 법정 일수를 만족하는지 확인하는 코드  
        // ...  
        // 휴가 일수를 금여 대장에 적용하는 코드  
        // ...  
    }  
}
```

Example 7 - 반복3

```
abstract public class VacaionPolicy {  
    public void accrueUSDivisionVacation() {  
        calculateBaseVacationHours();  
        alterForLegalMinimums();  
        applyToPayroll();  
    }  
  
    private void calculateBaseVacationHours() { /* ... */ }  
    abstract protected void alterForLegalMinimums();  
    private void applyToPayroll() { /* ... */ }  
}  
  
public class USVacationPolicy extends VacaionPolicy {  
    @Override protected void alterForLegalMinimums() {  
        // 미국 최소 법정 일수를 사용한다.  
    }  
}  
  
public class EUVacationPolicy extends VacaionPolicy {  
    @Override protected void alterForLegalMinimums() {  
        // 유럽연합 최소 법정 일수를 사용한다.  
    }  
}
```



Code Review

코드 리뷰란?

- 소프트웨어의 품질을 보장
 - 테스팅, 일일 빌드, 프레임워크의 사용, 개발 패턴, 코드 리뷰



가장 효과적

“코드를 실행하지 않고 사람이 검토하는 과정을 통하여
코드상에 숨어있는 잠재적인 결함(Defect)를 찾아내고
이를 개선하는 일련의 과정”

- 테스팅의 범주에서는 정적인 분석

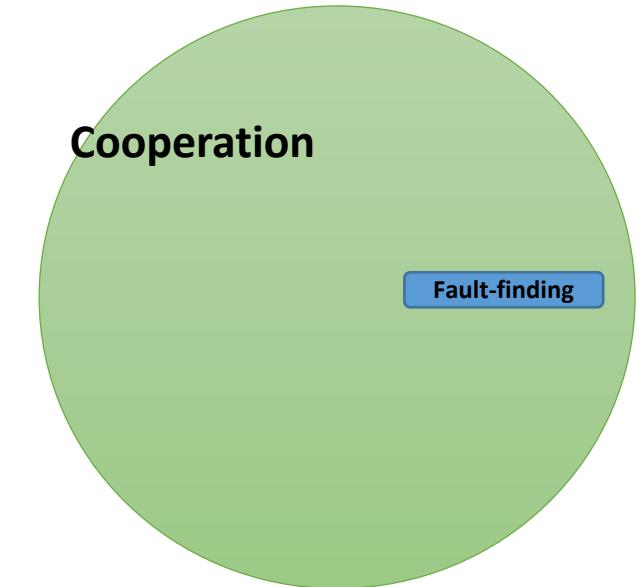
코드 리뷰의 목적

“It is intended to find and fix MISTAKES overlooked”

- Wiki

“Goal is COOPERATION, not fault-finding”

- Guido Van Rossum



- 서로 배우고 가르쳐 주는 시간
→ 개발자들이 서로 성장하도록 정보 공유, 교육
- 팀의 협업능력을 높이기 위한 노력
- Bad Code Smell 찾기

코드 리뷰 기법

1. 얼마나 정석적이고 프로세스적(정형성)이냐?

- Formal/Lightweight 방법

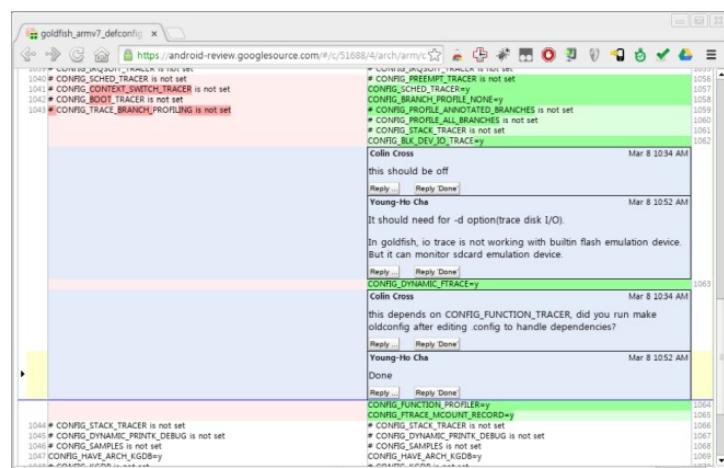
2. Offline vs. Online

- 직접 커뮤니케이션

- 메신저, Email, 기타 자동화된 코드리뷰 도구

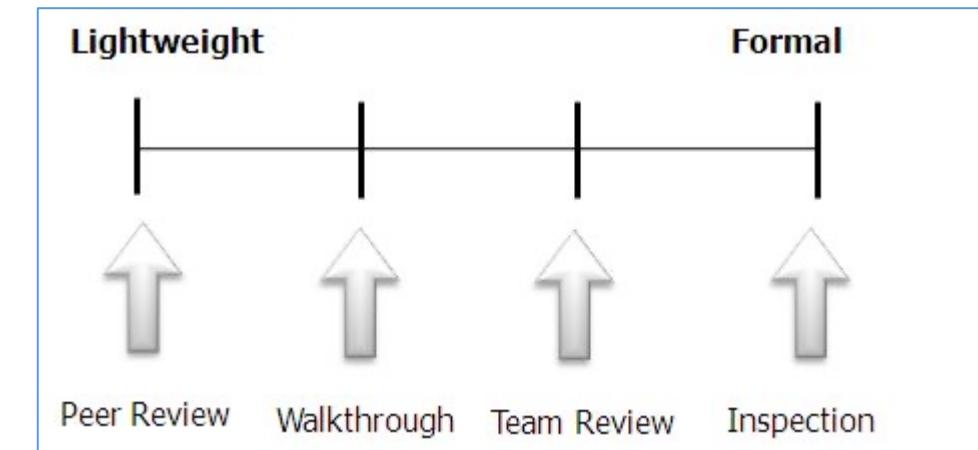
```

base: adds assumed tag pscsi_max_size() used by pscsi.h assignment via
> > req->cmd = &pt->pscsi_cdb[0] and drop the original CDB memcpys() of pt->pscsi_cdb[] into struct request->cmd[] -> struct request->_cmd[MAX_BLK_CDS].
> > Tested with nscsi_debug w/ IDWRITE_BLOCK and TCM_Loop w/ TCM/pSCSI backstores
> > using 'sgv4_xdwritereread -e'.
> > Signed-off-by:
> > ---
> > drivers/target/target_core_pscsi.c | 11 ++++++++-----+
> > drivers/target/target_core_pscsi.h | 1 +
> > 2 files changed, 86 insertions(+), 26 deletions(-)
> >
> > diff --git a/drivers/target/target_core_pscsi.c b/drivers/target/target_core_pscsi.c
> > index beea07..60f825d 10944
> > --- a/drivers/target/target_core_pscsi.c
> > +++ b/drivers/target/target_core_pscsi.c
> > @@ -72,29 +72,37 @@ static void *pscsi_allocate_request(
> >     static inline void pscsi_blk_init_request(
> >         struct se_task *task,
> >         struct pscsi_plugin_task *pt)
> >     {
> >         struct pscsi_plugin_task *pt,
> >         struct request *req,
> >         int bidi_read)
> >     {
> >         /*
> >          * Defined as "scsi command" in include/linux/blkdev.h.
> >          */
> >         pt->pscsi_req->cmd_type = REQ_TYPE_BLOCK_PC;
> >         req->cmd_type = REQ_TYPE_BLOCK_PC;
> >         /*
> >          * For the extra DID-COMMAND READ struct request we do not
> >          * need to setup the remaining structure members
> >          */
> >         if (bidi_read)
> >             return;
> >
> > I think that if you embed this pscsi_blk_init_request() inside it's
> > caller it would be more clear.
> >
> > But if the caller becomes too big, and this is a logical separation
> > then do the bidi_req->cmd_type = REQ_TYPE_BLOCK_PC in the caller
> > and call this one only for the main request, as before.
>
```



정형성 리뷰 방법

- **Formal한 코드리뷰**
 - Defect의 발견에 집중
 - 소프트웨어 개발 주기의 후반에 위치
- **Lightweight한 코드 리뷰**
 - Defect의 발견
 - 같은 로직을 여러 관점에서 생각하는 아이디어 회의
 - 주니어 개발자에게로의 지식 전달



Offline vs. Online 방법

- 오프라인 코드 리뷰
 - (+) 코드와 업무에 대한 팀원들의 이해도가 높아짐
 - (+) 코드리뷰를 처음 진행할 때 서로 도움을 줄 수 있음
 - (-) 시간이 많이 소요되거나 깊이 있게 살펴보기 어려울 수 있음
- 온라인 코드 리뷰
 - (+) 시간이 절약됨
 - (+) 업무를 잘 알거나 실력 있는 엔지니어가 코드를 집중적으로 보게 됨
 - (-) 몇몇 사람들에게 코드 리뷰 업무가 집중 될 수 있음
 - (-) 리뷰어 개개인의 능력에 많이 좌지우지 됨

코드 인스펙션

- 코드리뷰 기법 중에서 가장 정형화된 패턴의 기법
- 전문화된 코드 리뷰팀이 시스템이 어느 정도 구현된 단계에서 일정한 패턴을 가지고 코드를 분석
- 인스펙션팀은 크게 4가지 역할을 가지고 구성



코드 인스펙션

1. Moderator

- 인스펙션 팀의 실제적인 매니저
- 인스펙션 팀과 개발 팀 간의 인터페이스를 담당
- 필요한 리소스와 인프라를 확보
- 인스펙션에 대한 프로세스 정의와 산출물의 정리
- 인스펙션이 언제 끝날 것인지 (Exit Criteria)를 정의하는 역할

2. Reader

- 각종 산출물들을 읽고, 인터뷰 등을 통해서 전체 시스템을 이해하여 인스펙션 팀이 어떤 흐름으로 인스펙션을 진행할지에 대한 방향 지시
- Reader는 시스템의 큰 흐름과 구조를 잘 이해 + 팀 내에서 가장 많은 도메인 지식을 갖는 사람
- 상황에 따라서 문제가 발생할 수 있는 지점을 미리 예측
- 실제 인스펙션은 이 Reader의 방향성에 따라서 인스펙션을 진행

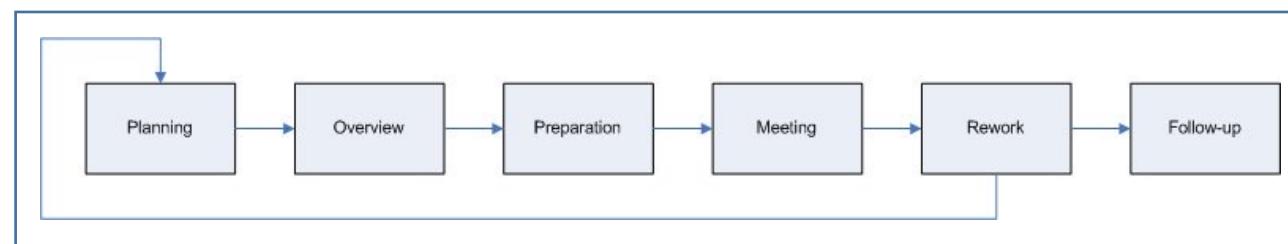
코드 인스펙션

3. Designer/Coder

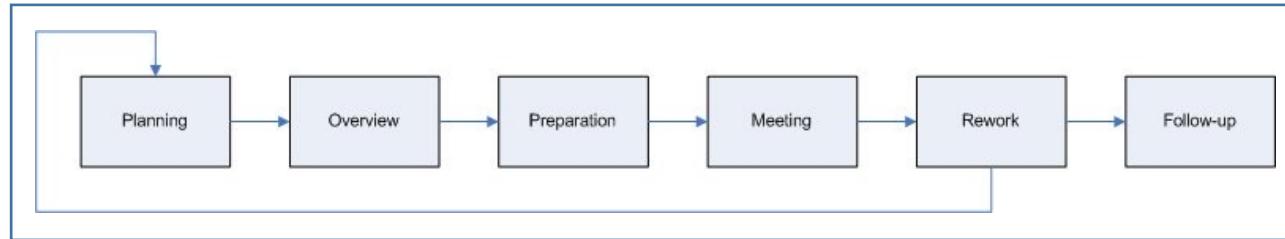
- Reader가 지시한 방향에 따라서 코드를 검증
- 잠재적인 Defect의 발견 및 권장 수정 방안을 만들어 냄

4. Tester

- 인스펙션이 진행중인 모듈에 대해서 테스트를 수행하고 Defect를 찾아내는 역할
- Designer나 Coder가 권장한 수정 코드 안에 대한 검증
- 실제 업무 개발자가 수정해온 코드에 대한 검증 작업 수행



코드 인스펙션 절차



1. Planning

- 전체 코드 인스펙션에 대한 계획을 수립
- 기간, 대상, 종료 조건 (금액 기준, 목표 TPS를 성취하는
지, 시간에 따른 종료 조건, 등등)
- 인스펙션 팀 SET UP

2. Overview

- 인스펙션 팀에 시스템과 구성 제품 등에 대한 교육
- 팀원간의 역할 정의

3. Preparation

- Inspection을 하기 위한 사전 준비 단계
- 각자의 역할별로 필요한 문서를 습득해서 이해하고
필요하다면 인터뷰도 진행
- 필요에 따라서 Tool과 환경 구축

4. Meeting (Inspection)

- 각자의 역할에 따라서 Inspection을 수행
- Inspection을 통해서 결함을 발견하고, 모든 결함은 기록
- 기록된 결함은 담당 업무 개발자에게 전달 → FIX
- 필요에 따라서는 권장 수정안을 전달하기도 한다.

5. Rework

- 보고된 Defect를 수정

6. Follow-up

- 보고된 모든 Defect가 수정되었는지 확인

팀 리뷰

- 코드 인스펙션보다 좀 덜 정형화 되었지만 그래도 일정한 계획과 프로세스를 따른다.
- 코드 인스펙션 프로세스의 단계 역할은 중복되거나 생략될 수 있음
→ Planning ,Overview ,Preparation등의 사전 준비 단계는 생략
- 발표자(Author)와 Moderator는 필수적으로 구성
- 리뷰시간에는 발표자(코드를 만든 사람)가 코드에 대한 설명
- Moderator는 리뷰의 주제를 선정하여 리뷰를 진행
 - 리뷰에서 나온 의견을 정리해서 Action Item으로 기록 (Wiki 등)
 - Action Item들은 프로젝트 관리자가 실제 프로젝트 Task로 관리
→ 일정에 반영되어야 한다.
- 일주일에 한번 정도 팀 리뷰를 수행
 - 특정 모듈이나 기능이 완료되는 시점 시점에 수행
 - 테스트 결과를 가지고 리뷰

: -01-19 Logging framework 2'nd review

This page last changed on 1# 19. 2009 by admin.

Table of contents

- Speaker : KsAhsn
- Objective
- Info
- Action Item
- Suggestion

Speaker : KsAhsn ← 코드 리뷰 발표자

Objective ← 코드 리뷰 목적 및 대상

- Review Logging Framework Code Review

Info ← 코드 리뷰 결과 발견된 개선 사항

- Review implementation of [OAF Logging Design](#)

Action Item

- AI 1. Performance testing between file persistence based logging vs direct file logging (T99_1)
- AI 2. Change reload interval time set up implementation to configuration file (T99_2)
- AI 3. Review synchronization handling in hashmap (T99_3)
- AI 4. Add "GET" method in REST interface and reload (T99_4)
- AI 5. Add comment and refactor naming convention (T99_5)
- AI 6. change JNDI look up ip for Datasource, and separate Datasource name to configuration file (T99_6)
- AI 7.Implement MDB to support separate logging file, and add rotation capability. File size is assigned in (T99_7)
- AI 8.configuration file (T99_8)
- AI 9.Enhance DB programing to getting DB connection Every time (T99_9) ← TASK NUMBER

Suggestion

Walkthrough

- 단체로 하는 코드 리뷰 기법 중에서 가장 비정형적인 방법 중에 하나
- 발표자가 리뷰의 주제와 시간을 정해서 발표
 - 동료들로부터 의견이나 아이디어를 듣는 시간을 갖는다
- 주로 사례에 대한 정보 공유나, 아이디어 수집을 위해서 사용
- 개발을 위한 프로세스에서 보다는 “Bug 사례에 대한 회의”와 같은 정보 공유성격에 유리
- 유일하게 발표자만이 리뷰를 주관하고 발표하는 역할
 - 다른 참여인원들은 아무런 책임이나 역할을 가지지 않고 자유롭게 의견을 개진
- Walkthrough는 정기적으로 (주일에 한번) 진행할 수도 있으며, 또는 정보공유나 아이디어 수집이 필요할 경우 비정기적으로도 진행할 수 있음
 - 정기적으로 진행하는 것이 참여율이나 집중도가 더 높음

Peer Review or Over the shoulder review

- 2~3명 (주로 2명)이 진행하는 코드 리뷰의 형태
- 코드의 작성자가 모니터를 보면서 코드를 설명하고 다른 한 사람이 설명을 들으면서 아이디어를 제안하거나 Defect를 발견하는 방법
- 사전 준비 등이 거의 필요 없고, 필요 할 때마다 자주 사용할 수 있는 리뷰 방법
- 주로 Senior 개발자(사수)가 Junior 개발자를 멘토링 할 때 사용
 - Junior 개발자에 대한 교육과 함께, Junior 개발자가 양산한 코드에 대한 품질을 관리
- Senior 개발자의 리뷰 역량에 따라서 결과물의 품질이 달라질 수 있음
- Senior 개발자의 시간 투여량이 많은 만큼 Senior 개발자의 참여도가 떨어질 수 있다.
 - 형식적일 수 있음
 - 프로젝트 관리 관점에서 Peer Review 에 대해서도 프로젝트 스케줄 상의 Task로 잡아서 하나의 독립된 업무로서 시간과 노력을 투자할 수 있도록 지원

Pass-around

- 온라인 보다는 오프라인 위주로 진행되는 리뷰
- 작성자가 리뷰를 할 부분을 메일이나 시스템을 통해서 등록
 - 참석자들이 메일을 통해서 각자의 의견을 개진하는 방식
- 시간과 장소에 구애를 받지 않는 방식
 - 원격에서 작업하는 팀의 경우 유리한 리뷰 방식
 - 반대로 Ownership이 애매하여 원하는 결과가 나오지 않는 경우가 많다.
- 실시간이 아닌 비동기적인 커뮤니케이션으로 인해서 커뮤니케이션 속도가 매우 느리다는 단점

어떤 코드 리뷰 기법?

1. 코드 인스펙션

- 코드 인스펙션의 전제는 전문성을 가지고 있는 인스펙션 팀이 일정한 프로세스와 패턴에 따라서 개선안을 찾는 작업
→ 즉, 고도로 훈련됨 팀과 기간이 필요하고, 어느 정도 개발이 완료되어 있는 인스펙션 대상(시스템)이 있는 것을 전제로 한다.
- 인스펙션의 시기는 시스템이 개발되어 있는 시점인 Release이 유용하다.
- 개발 초기에 비기능적인 구현을 끝낸 경우 1차 Inspection
→ 개발이 끝난 후 시스템 테스트 (성능, 확장성, 안정성 등) 시점에 2차 Inspection 실행 권장

어떤 코드 리뷰 기법?

1. 코드 인스펙션

- 인스펙션을 수행하는 주체
 - Task Force를 운영하여 프로젝트를 돌아다니면서 인스펙션을 수행
 - 여러 개의 솔루션을 인하우스 개발하는 업체 (네이x)와 같은 업체는 QA 조직 내에 자체적으로 인스펙션팀을 운영하는 것을 권장
 - 그 외의 일반 업체 (갑)나 기업의 경우 인스펙션 프로세스를 전체 개발 프로세스와 프로젝트 비용 산정에 포함하고 SI나 벤더 컨설팅을 활용하는 것을 권장.
- 인스펙션의 결정 주체는 주로 PMO(Project Management Office)와 AA (Application Architect) 되는 것이 바람직
 - 대체로 개발 주체 조직 외부에서 인력을 데리고 오고, 여러 팀이 함께 인스펙션에 참여해야 하며 일정에 대한 조정과 인스펙션 결과에 대한 반영이 필요.

어떤 코드 리뷰 기법?

2. 팀리뷰

- 각 개발 유닛에서 활용하기 좋은 기법
- PL (Project Leader)의 역량아래 수행할 수 있으며, 팀원이 리뷰어가 되기 때문에 팀 단위에서 활용
- 일주일에 한번 정도, 한 시간 정도 리뷰를 정기적으로 수행
- 시니어 개발자나 PL이 리뷰 대상 모듈을 선정하고 개발자에게 리뷰를 준비
- 리뷰는 발표자가 리드
- 리뷰에서 나온 의견을 Action Item으로 잡아서 PL이 발표자의 스케줄로 조정을 해주는 작업이 필수
- 위키와 같은 문서 공유 시스템을 이용하여 반드시 리뷰의 결과를 남기도록
 → 리뷰의 결과는 Task Management의 Task로 연결
- 리뷰되고 반영된 내용은 그냥 넘어가지 않도록 하고, 재 테스트를 통해서 반영 내용을 반드시 검증

어떤 코드 리뷰 기법?

3. Walkthrough

- 비정기적으로 언제나 개최할 수 있는 일종의 아이디어 회의
- 팀리뷰처럼 PL이나 시니어 개발자가 중재를 하지 않기 때문에 구성원들의 의욕이 낮을 경우 효과가 매우 낮음
- 개발팀보다는 QA나 운영팀에서 장애 사례나 버그 수정 사례 등의 정보 교환 목적으로 사용하는 것이 좋음

4. Peer Review

- 신입 개발자 교육이나, 해당 제품이나 기술에 전문적인 지식이 없는 경우에 Knowledge Transfer(지식공유)와 품질 유지를 위해서 유용
- 리뷰어의 Task 부담이 가중되기 때문에 리뷰어에 대한 스케줄 배려가 필요

어떤 코드 리뷰 기법?

	시기	효과	변경에 대한 비용	수행 비용	주체
인스펙션	1차 Release, 시스템 테스트	매우높음	매우 높음	높음	PMO,QA,AA
팀리뷰 ★	매주	높음	보통	보통	PL
Walkthrough	비정기적	낮음	보통	낮음	원하는 개발자
Peer Review	필요한 경우	경우에 따라 높음	낮음	보통	시니어 개발자

코드 리뷰 시점

- 소스 코드를 저장소에 넣기 전에 수행하는 방식인 “Pre-commit” 방식
- 소스 코드를 저장소에 넣은 다음 수행하는 “Post-commit” 방식

- Pre-commit

- (+) 좀 더 안전한 코드가 repository에 들어가게 됨
- (-) 일정이 부족할 경우 간소화 할 우려있음
- (-) 별도의 도구나 Local Repository가 지원되는 환경이 제공될 때 효율이 더 높다.

- Post-commit

- (+) 리뷰어가 대상 코드에 손쉽게 접근 가능
- (-) 불안전한 코드가 repository에 들어가게 됨

What? When?

- 무엇을 리뷰하는가?
 - 코드리뷰보다 더 중요한 것은 스펙과 설계 리뷰
 - 스펙과 설계리뷰가 더 어려운 이유는 스펙과 설계를 제대로 작성하지 않기 때문
 - 코드리뷰 때보다 스펙과 설계 리뷰 시에 더 다양하고 중요한 것을 배우고 공유할 수 있다.
- 언제 리뷰하는가?
 - 코드리뷰를 포함해 리뷰의 실패로 이어지는 대표적인 방법은 나중에 몰아서 리뷰
 - 별다른 전략 없이 1주나 2주에 한번씩 개발자들이 모여서 지금까지 작성한 코드를 놓고 끝장 리뷰
 - 사전에 검토하지 않고 참석해서 내용을 파악하지도 못하고 장시간 모여 리뷰를 하게 되면 집중력이 떨어지고 형식적인 일
 - 성공적인 리뷰를 하려면 그때 그때 바로 해야 한다. → Peer desk check
 - 소스코드 리뷰시스템 → 원격지에 있는 개발자와도 리뷰가 가능

Who?

- 누가 리뷰하는가?
 - 리뷰자를 프로세스에 강제로 지정하는 비효율적인 리뷰가 아닌 리뷰 내용과 목적에 따라서 적절한 사람이 리뷰를 할 수 있도록 해야 함
 - 리뷰는 목적에 따라 다양한 사람들이 할 수 있다.
 - 코드리뷰는 주로 고참개발자들이 진행하지만 개발자들끼리 서로 리뷰를 할 수도 있다.
 - 내용에 따라서 어려운 것은 특별히 고참개발자를 지정해서 리뷰를 요청
 - 일반적인 것들은 동료와 같이 리뷰
 - 스펙과 설계를 작성할 때는 각 분야의 전문가와 리뷰
 - 마케팅팀, 영업팀, QA팀 등과 해당 팀과 관련된 내용을 리뷰
 - 설계를 할 때는 아키텍트들의 도움을 받도록
 - 특정 기술에 대해서는 해당 기술의 전문가의 리뷰를 받도록

How?

- 어떻게 리뷰하는가?
 - 리뷰는 감사(Audit)가 아니다.
 - 코딩 한줄 더 하는 것보다 리뷰를 해주는 것이 전체적으로 이익
 - 리뷰를 하기 위해 소스코드, 설계문서, 스펙문서 필요
 - 자료를 미리 배포가 되고 검토를 한 후에 리뷰를 진행하는 것이 더 효율적
 - 온라인으로도 충분히 리뷰를 진행 가능

코드리뷰 진행 후 결과

1. 모르면 문제가 될만한 중요한 문제점
2. *For newbie.* 처음 온 사람을 위해. 초심자가 알아야 할 것.

딱 두 가지만 온라인(팀원들이 언제나 볼 수 있는 곳)에 간결하게 남기는 것이 핵심!

코드리뷰 진행 시 유의사항 1)

- 자유롭게 의사소통을 진행할 수 있는 분위기를 조성한다.
- 쿨하게 얘기하고 상처받지 않기
- 팀원들이 친해져야 한다. (적어도 서로의 말에 상처받지 않을 만큼)
- 코드를 사람과 일치시켜서 얘기하지 않는다.
- 표현에 유의하기
- 공격적인 어조는 여러모로 좋지 않다.
- 상대의 지적을 비난으로 생각하지 않는다.
- 리뷰시간이 논쟁으로 시간을 소모하지 않도록 주의한다.
- 문제의 해결책에 대한 논의가 길어질 경우 따로 시간을 정해 진행한다.

코드리뷰 진행 시 유의사항 2)

- 결정은 본인이 하게 하고, 다른 사람은 존중하기
- 추후에 문제가 생겼을 경우에도 그 역시 존중하고(비난하지 않고) 빨리 해결책 찾기
- 협의가 되지 않는 경우 시니어에게 결정권 내리기
- 진행자를 선정 후에 진행한다.
- 진행자는 시간관리, 논쟁 중재 등을 수행한다.
- 시간 엄수
- 시작 전에 반드시 진행 시간을 정한다.
- 매 리뷰는 1시간 이내를 권장

문제점1) 코드리뷰 → 강제화

- 결과는...

효율적인 리뷰가 되기보다 < 의무적인 리뷰

- 리뷰에 대한 나쁜 기억
- 리뷰문화가 제대로 정착하지 못함
- 포기하거나 강제화 덕에 비효율적이지만 명맥만 유지

그러나,

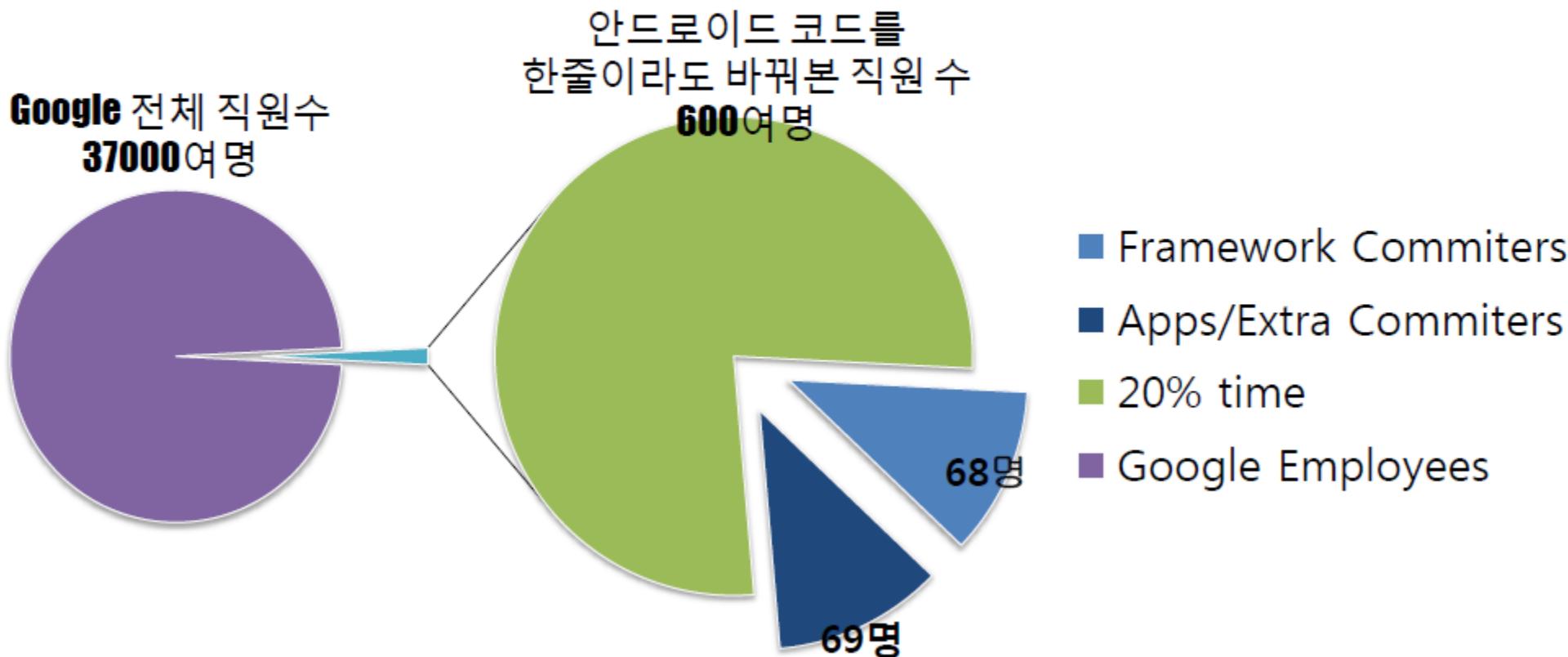
“리뷰문화는 어렵다고 포기해도 될 만큼 사소하지 않다.”

문제점2)

- “내가 만든 코드를 남이 잘못되었다고 이야기 한다.”
 - **취조? 방어적!**
- 리뷰의 주요 목적은 결함의 발견과 개선 방안
 - **감정싸움이 아님!**
- 어떤 프로세스나 시스템으로 될 수 있는 일이 아니라 “사람 사이의 관계”에서 발생되는 일
 - **“문화”의 변화가 필수!**
- 대부분의 리뷰 기법들은 리뷰와 그에 대한 후속 처리가 시간과 사람이 필요한 일
 - **프로젝트 운영 관점에서 시간과 리소스에 대한 배려 필요**

구글 직원의 AOSP 참여 현황

Google Employees



Guido Van Rossum

- 네덜란드 출신의 프로그래머
 - 파이썬 프로그래밍 창시자
 - 2005 ~ 2012 → Google
 - 2013 ~ → Dropbox
-
- Google 재직 당시 Mondrian 이라는 코드 리뷰 시스템 개발



Mondrian vs. Rietveld

Mondrian

- Written by **Guido Van Rossum**
- Written with **Python**
- Announced in 2006
- Integrate with **Perforce**
- Hosted and Used at **Google Internally**

Rietveld

- Written by **Guido Van Rossum**
- Written with **Python**
- Announced in 2008
- Integrated with **Subversion**
- Host on **Google App Engine**
- Used by **Chrome Project**

Gerrit

- Gerrit (2008-2012)
- Written by Sean O. Pearce
- fork from Rietveld
- Integrated with Git
- add Access Control List feature
- Gerrit2 (2008 -)
- Rewritten Gerrit with JavaEE
- Used by Android Project

특징

- 여러 가지 환경에서 운영 가능
 - jvm만 설치되어 있으면 PC에서도 운영 가능
- 표준 servlet container를 지원
- 여러 가지 인증 방식 지원
 - http/ldap/openid
- 여러 가지 database 지원
 - mysql/postgresql/h2(내장 DB)



Gerrit

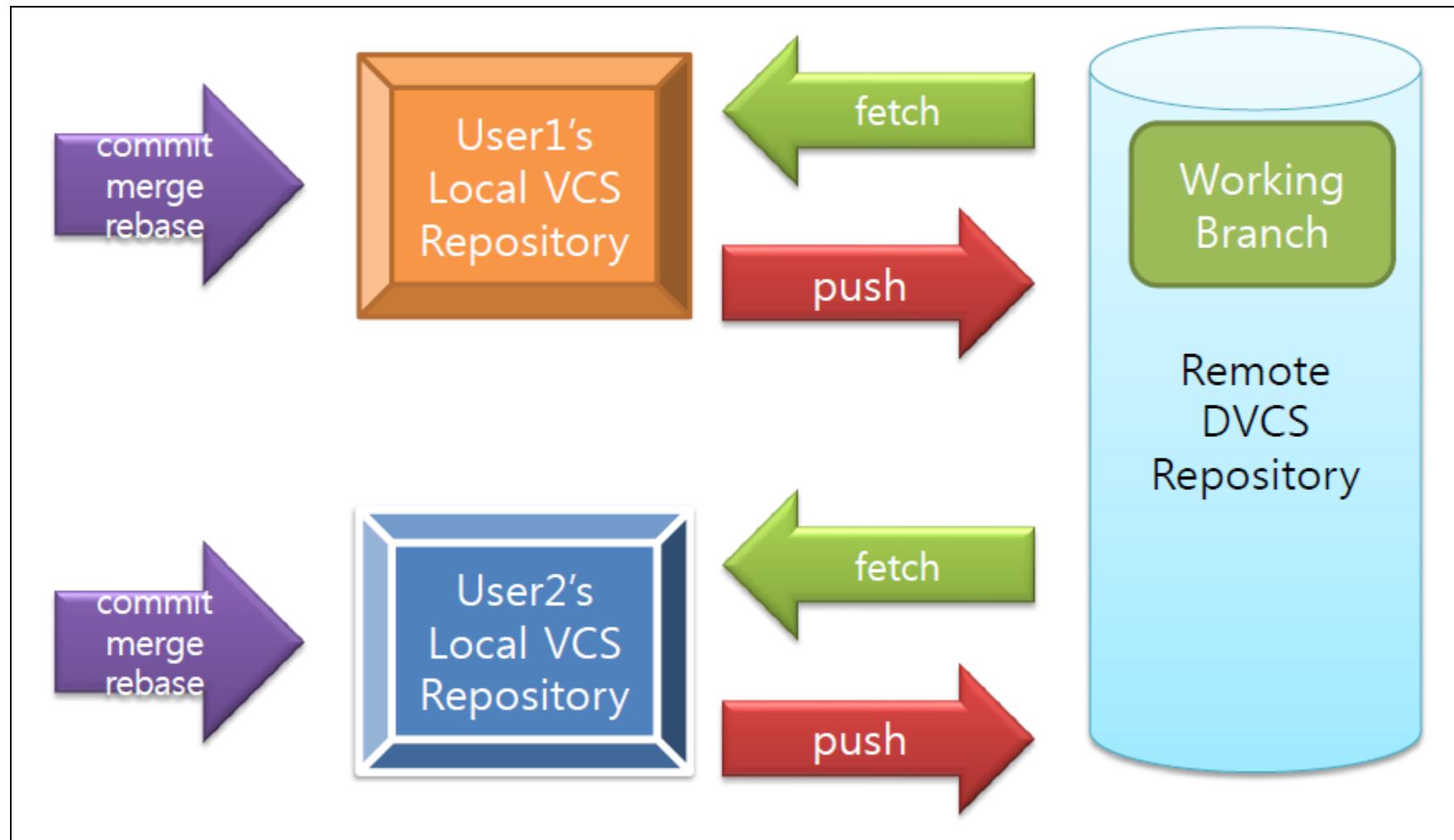
기능

- Git 저장소
 - jgit을 이용해서 git 저장소 구현
- Access Control List
 - 그룹단위로 사용자의 권한 설정 가능
 - git의 기능 및 저장 위치 별로 그룹 설정 가능
→ fetch, push, tag, branch...
 - 특정 그룹에게 검증 범위를 지정 가능
→ review, verify, submit...
- Source Review Board
 - 임시 git branch를 자동으로 생성해서 코드를 올릴 수 있고, 그 코드에 대해서 평가할 수 있는 웹 보드 시스템 제공

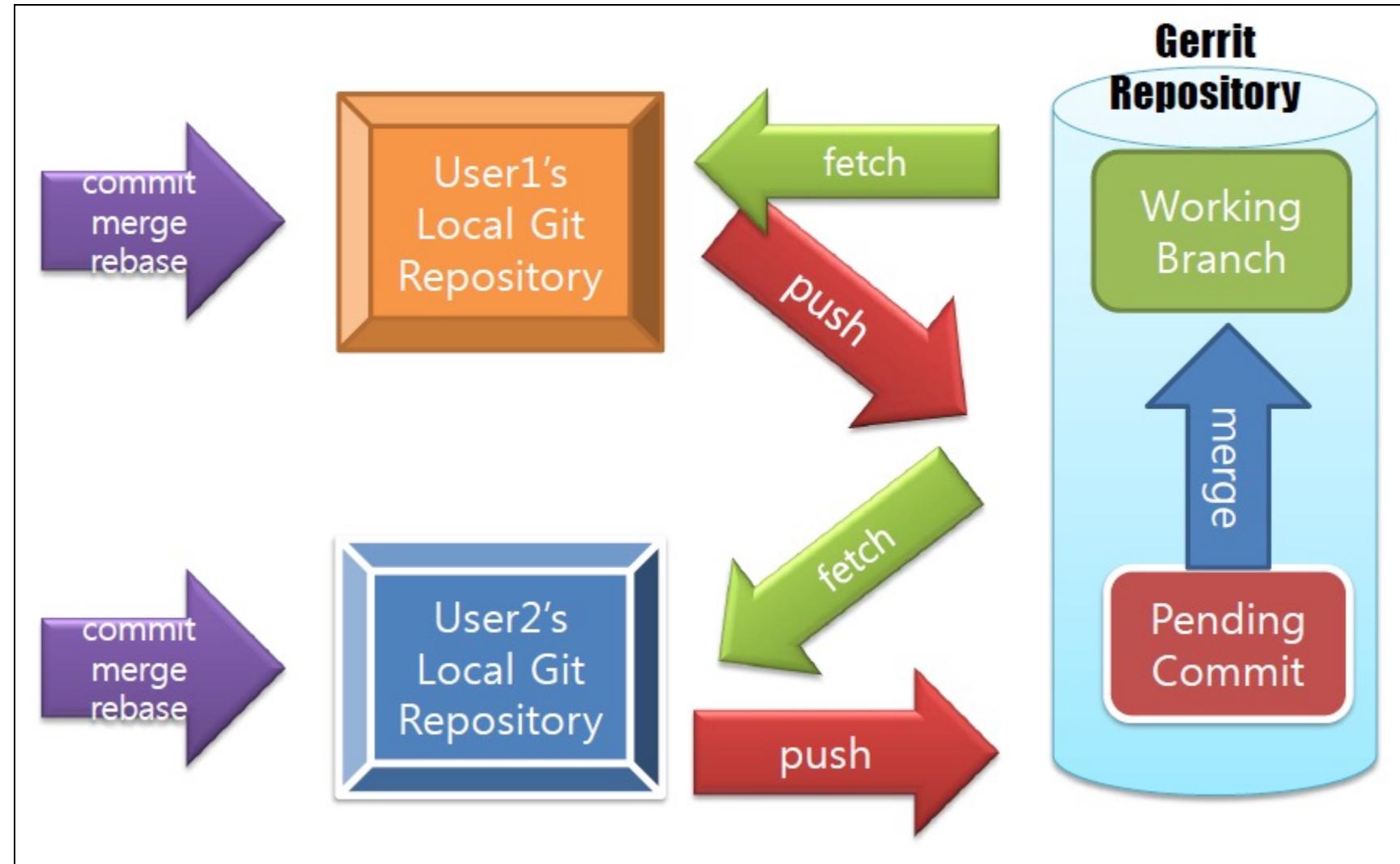
인터페이스

- for Human
 - 웹 인터페이스
- for External Integration
 - ssh
 - 커맨드라워 워터페이스
 - 2개의 커맨드 제공
 - git
 >> git protocol 구현
 - gerrit
 - REST api
 - json형식으로 gerrit의 데이터 제공

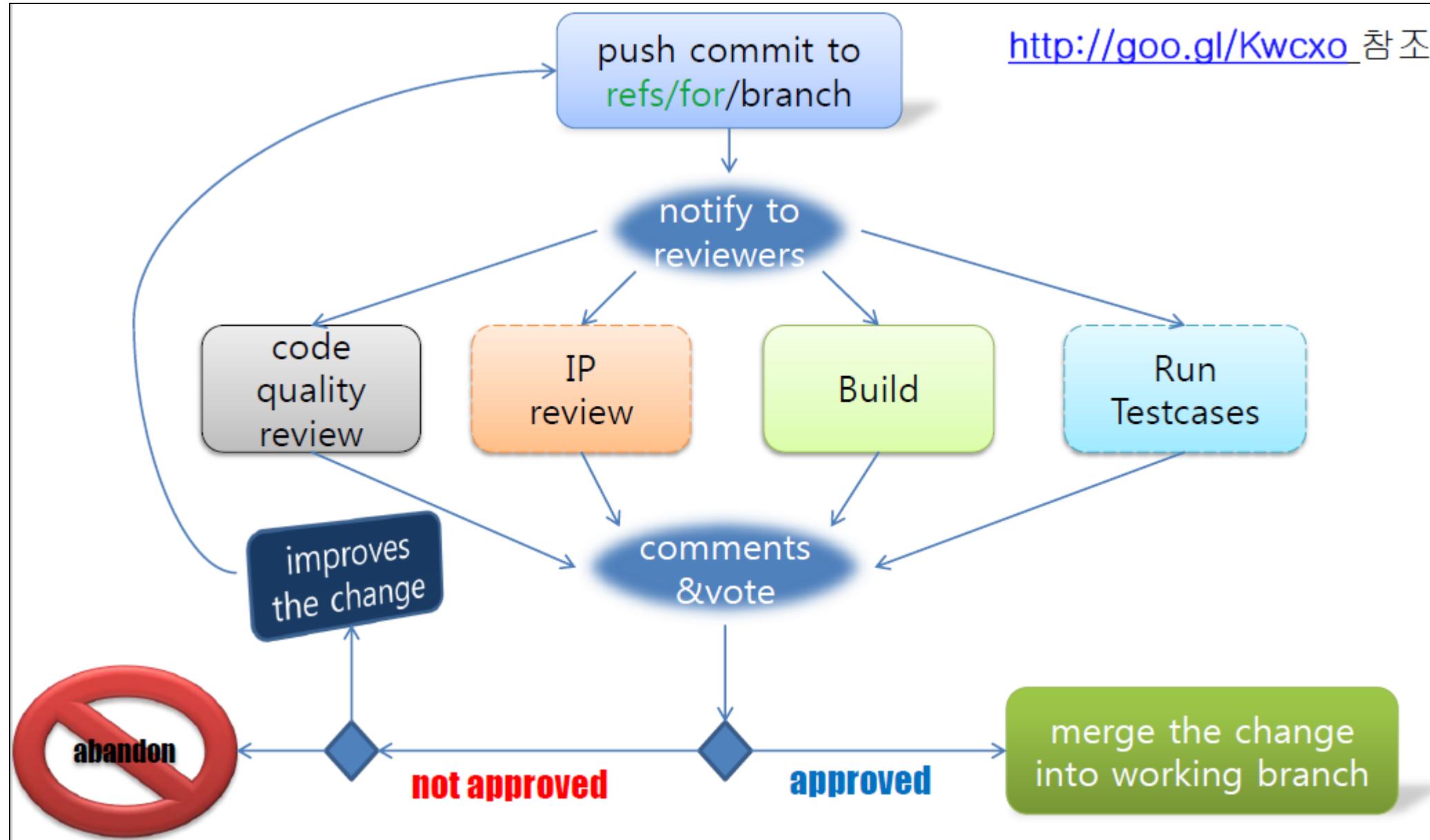
DVCS(Git) Workflow



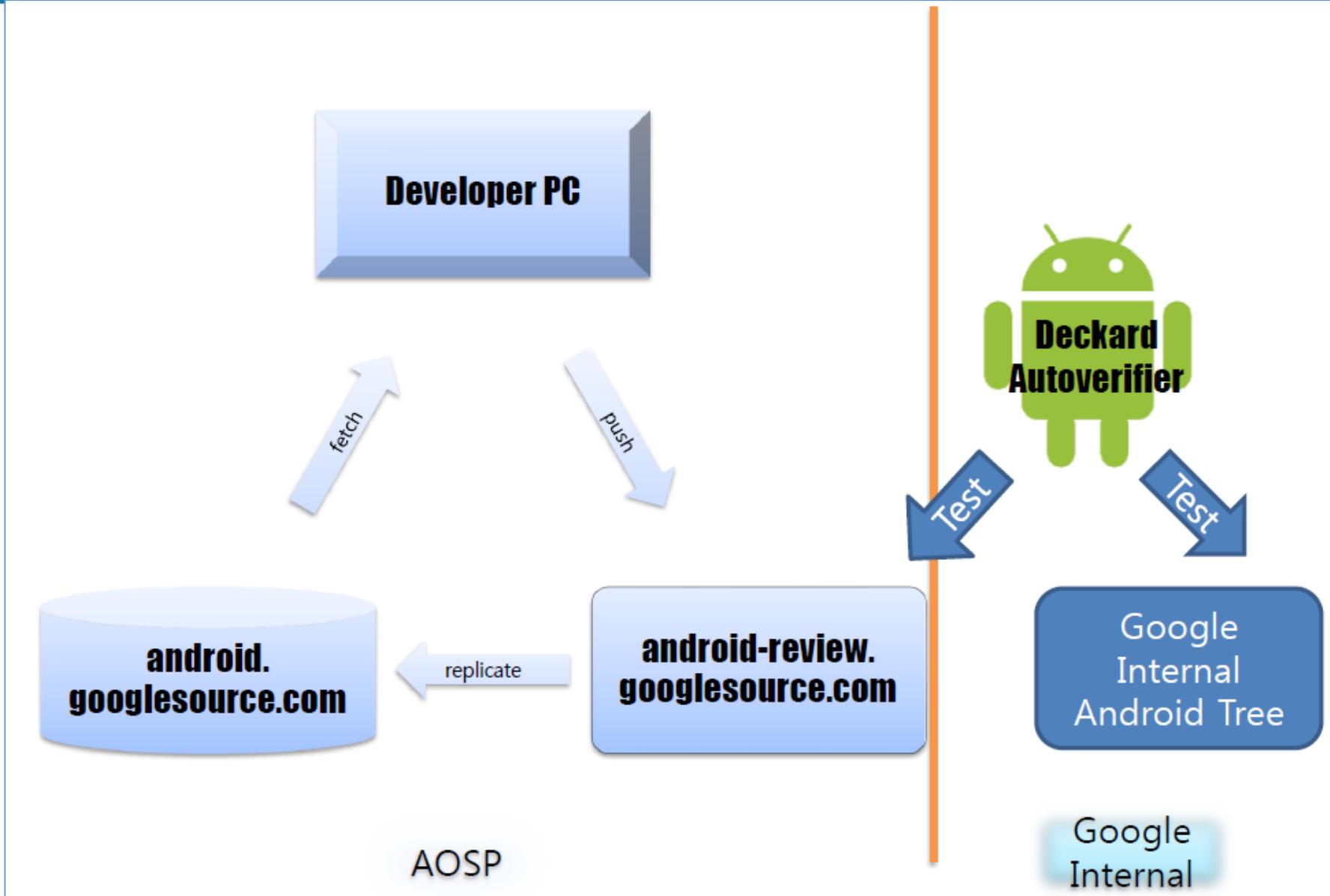
Gerrit Workflow



Gerrit Detailed Workflow

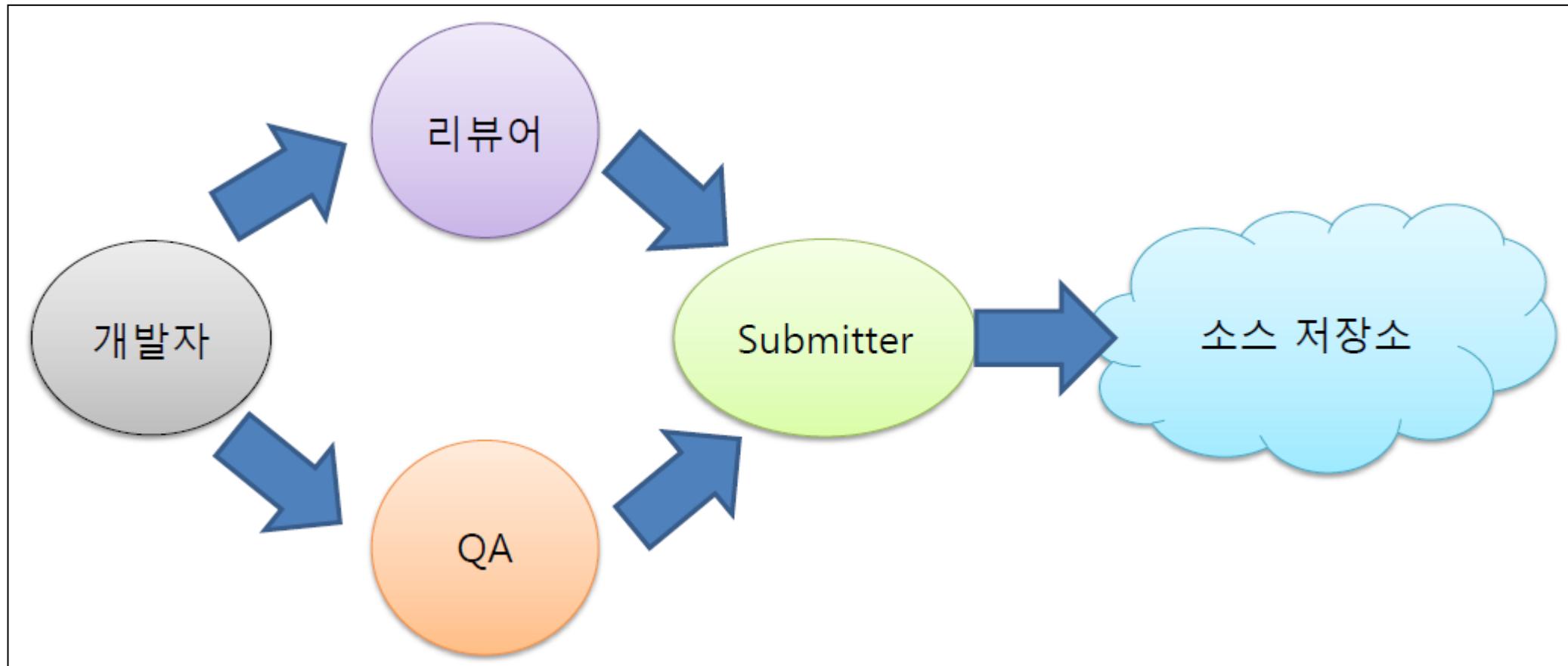


Google Android Source Hosting

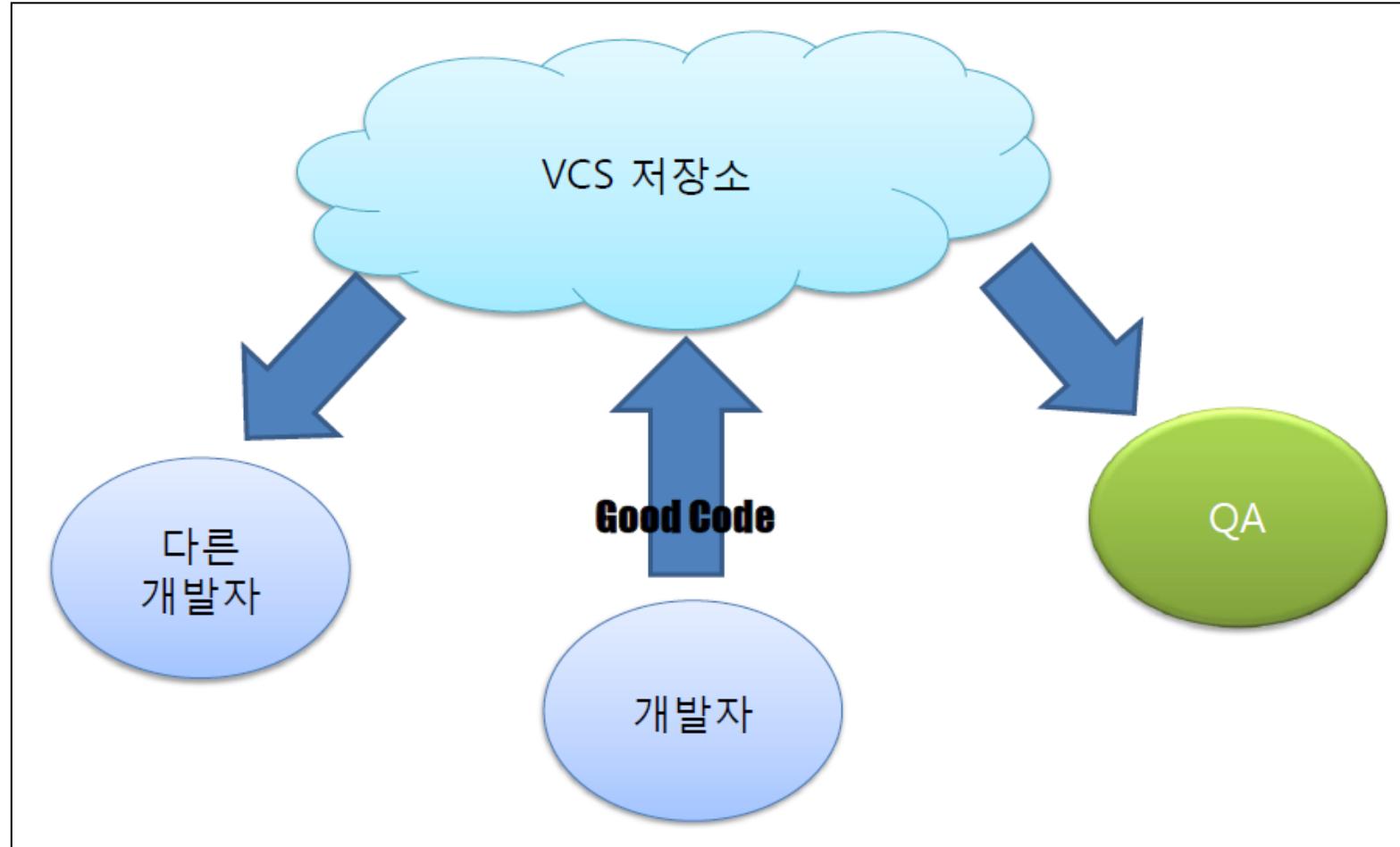


코드 리뷰 시스템

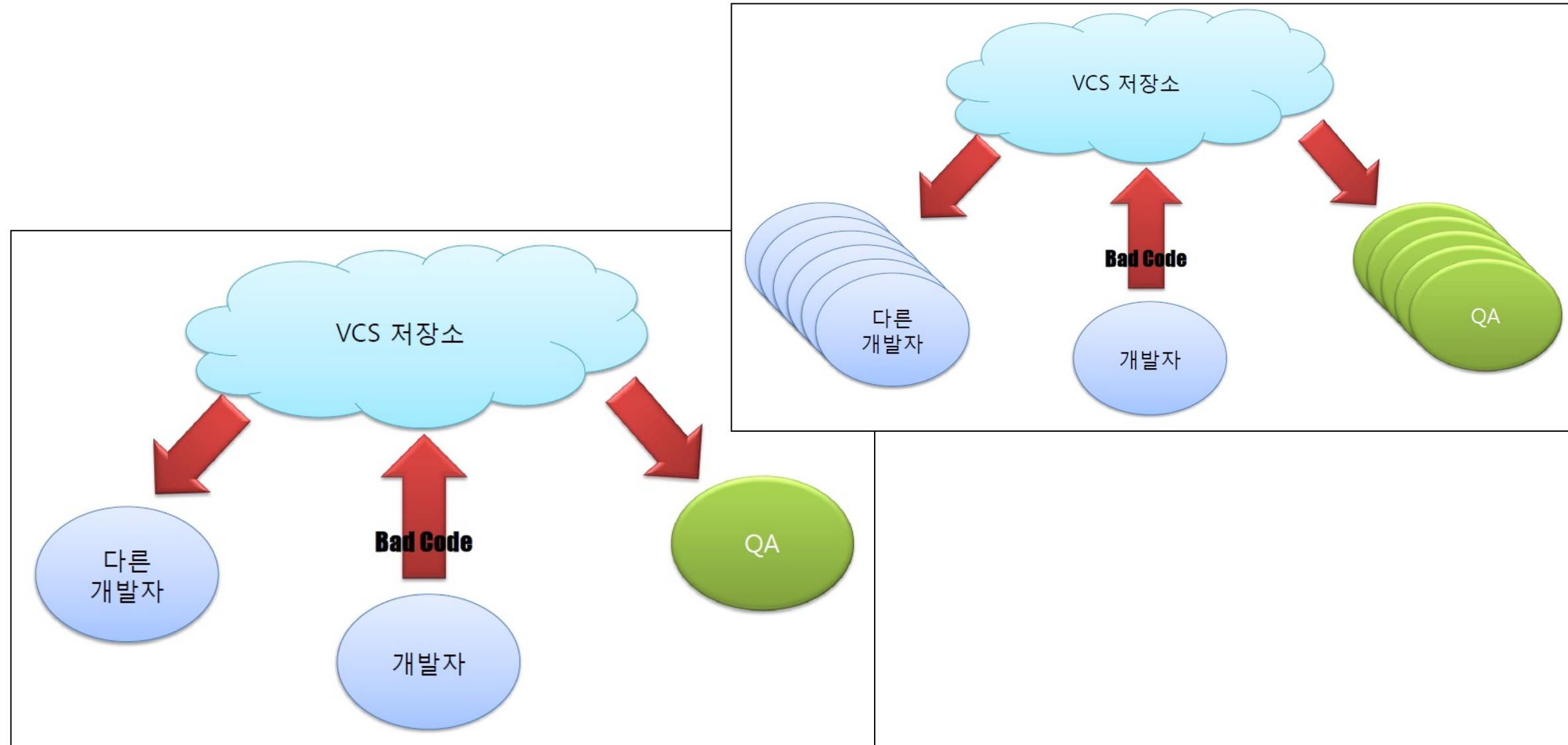
- 코드 리뷰를 개발 프로세스에 포함



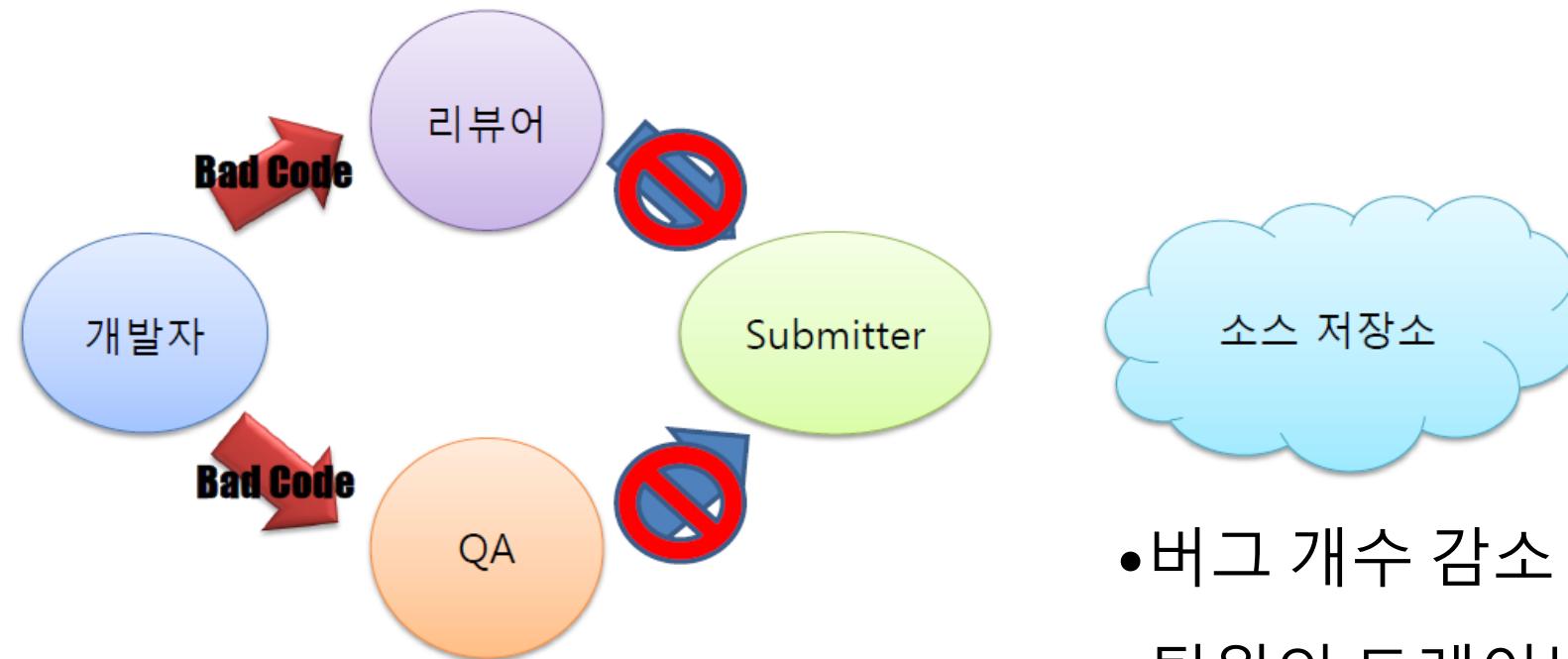
모두가 원하는 상황



하지만 현실은...



코드 리뷰 시스템



- 버그 개수 감소
- 팀원의 트레이닝 도구로 활용
- 코드 가독성 증가 및 품질 상승
- 코드 세부 구현사항에 대한 기록 보존
- 팀 역량의 상향 평준화
- 디버깅 시간 및 프로젝트 수행 기간 단축

코드 리뷰 SW

- 코드 리뷰 절차를 소프트웨어(일반적으로 웹 게시판형식)로 구현해둔 것.
 - Mondrian (Google)
 - Rietveld (Google)
 - Gerrit (Google)
 - Phabricator (Facebook)
 - ReviewBoard
 - Barkeep
 - RhodeCode
 - GerritForge (Based on Gerrit)

“사실 리뷰를 통하지 않고서는 개발자들의
핵심 역량이 성장하기는 어렵다”

“직원들의 적응 상태를 봐가면서 아주 천천히 한발씩”

“리뷰를 강제화하되 벌칙보다는 포상으로 정착을 유도”

Gerrit 설치

1. 설치 환경

- 운영체제 : 우분투 (Ubuntu)
- Gerrit : Gerrit 2.7

2. 고려 사항

- DB : H2, MySQL, PostgreSQL
- 인증 방식 : Open ID, HTTP, LDAP
- 서버 : WAS (Jetty 내장, 상용 WAS 지원)

Gerrit 설치

3. 준비 사항

- 자바 설치
 - sudo add-apt-repository ppa:webupd8team/java
 - sudo apt-get update
 - sudo apt-get install oracle-java7-installer
 - java -version
- SSH 설치
 - sudo apt-get install openssh-server
 - sudo ufw allow ssh
- Apache 설치
 - sudo apt-get install apache2
 - sudo apt-get install libapache2-mod-proxy-html
 - sudo a2enmod proxy
 - sudo a2enmod proxy_http
 - sudo service apache2 restart
- Git 설치
 - sudo add-apt-repository ppa:git-core/ppa
 - sudo apt-get update
 - sudo apt-get install git-core git-review
 - git version

Gerrit 설치

4. Http 인증을 위한 VirtualHost Proxy 설정

- VirtualHost 작성
 - sudo nano /etc/apache2/sites-available/gerrit2.conf
- Apache에 site-enabled 설정
 - cd /etc/apache2/site-enabled
 - sudo ln -s ../sites-available/gerrit2.conf ./gerrit2.conf

5. as Gerrit 사용자 등록

- htpasswd -c /home/gerrit2/gerrit/etc/passwords "admin"

6. Apache 재시작

- sudo service apache2 restart

```
<VirtualHost *:8080>
    ServerName localhost
    ProxyRequests Off
    ProxyVia Off
    ProxyPreserveHost on
<Proxy *>
    Order deny,allow
    Allow from all
</Proxy>
<Location /login/>
    AuthType Basic
    AuthName "Gerrit Core Review"
    Require valid-user
    AuthUserFile /home/gerrit2/gerrit/etc/passwords
</Location>
    AllowEncodedSlashes on
    ProxyPass / http://localhost:8081/
</VirtualHost>
```

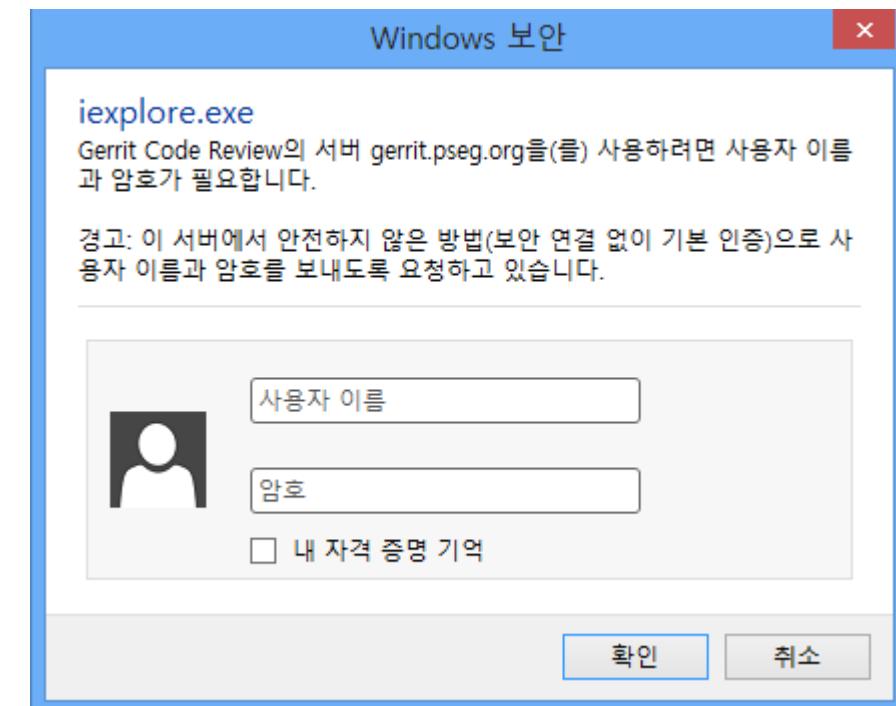
Gerrit 설치

7. Gerrit 다운로드

- wget <https://gerrit-releases.storage.googleapis.com/gerrit-2.7.war>

8. Gerrit 설치

- java -jar <downloaded path>/Gerrit-2.7.war init -d <installation directory>

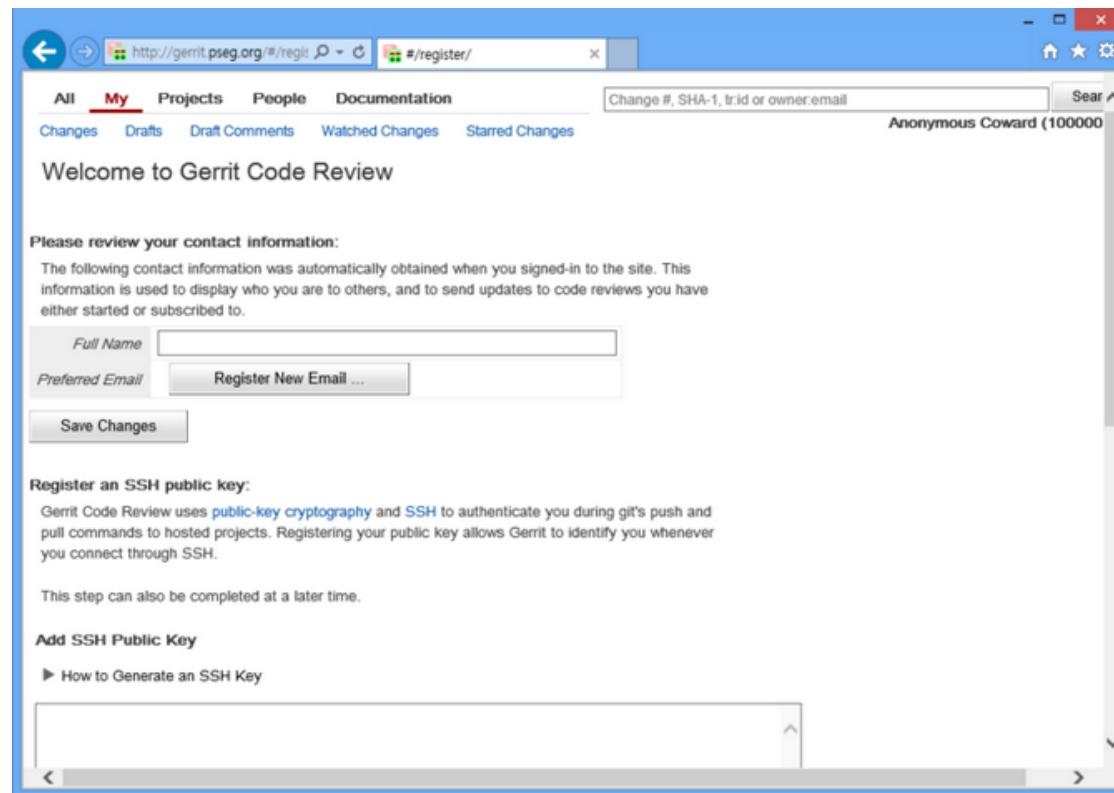


Gerrit 설치

9. Gerrit 설정 변경

- sudo nano etc/gerrit.config
- ./bin/gerrit.sh stop
- ./bin/gerrit.sh start

10. Welcome to Gerrit Code Review



[gerrit]

```
basePath = git
canonicalWebUrl = http://localhost:8080/
[database]
type = mysql
hostname = localhost
database = reviewdb
username = gerrit2
```

[auth]

```
type = HTTP
logoutUrl = http://aa:aa@localhost:8080/
```

[sendemail]

```
smtpServer = smtp.gmail.com
smtpServerPort = 465
smtpEncryption = SSL
smtpUser = *****
smtpPass = *****
```

[container]

```
user = gerrit2
javaHome = /usr/lib/jvm/java-7-oracle/jre
```

[sshd]

```
listenAddress = *:29418
```

[httpd]

```
listenUrl = http://*:8081/
```

[cache]

```
directory = cache
```

Gerrit 사용

1. 사용자 추가

- htpasswd ~/gerrit/etc/passwords "user"

2. 로그인 후 기본 정보 수정

- Preferred Email 인증
- SSH Key 발행
- Http password 생성

3. 프로젝트 생성

4. 프로젝트 그룹 생성

All My Projects People Plugins Documentation Change #, SHA-1, tr:id or owner:email

List Create New Project

Create Project

Project Name: TestProject

Rights Inherit From: All-Projects

Create initial empty commit
 Only serve as parent for other projects

Parent Suggestion Project Description

All-Projects Access inherited by all other projects.

All My Projects People Plugins Documentation Change #, SHA-1, tr:id or ow

List Groups Create New Group

Group TestGroup

General Members

Members user

Member	Email Address
<input type="checkbox"/> admin	

Included Groups

Group Name	<input type="button" value="Add"/>
Group Name	Description

Gerrit 사용

5. 프로젝트 클론 및 작업
6. Eclipse에서 Git repository 생성
7. Source 수정
8. \$ git commit -a -m "Test"
9. \$ git review
 - .gitreview 파일 필요 시 다음 내용 생성

```
[gerrit]
host=localhost
port=29418
project=TestProject.git
defaultbranch=master
```

The screenshot shows the Gerrit web interface with the 'Projects' tab selected. The project name is 'TestProject'. The 'General' tab is selected under the clone section, showing the clone URL: `git clone ssh://admin@gerrit.mvcircle.com:29418/TestProject`. The 'Description' field is empty. In the 'Project Options' section, the 'Submit Type' is set to 'Merge if Necessary', 'State' is 'Active', 'Automatically resolve conflicts' is 'INHERIT (true)', and 'Require change-ID in commit message' is 'INHERIT (true)'. In the 'Contributor Agreements' section, 'Require signed-off-by in commit message' is set to 'INHERIT (false)'. A 'Save Changes' button is at the bottom.

Gerrit 사용

10. Reviewer 설정

- All > Open에서 review 확인
- Add Reviewer

11. Review

- review 점수가 2점이 넘으면 코드는 Git으로 Commit 된다.

The screenshot shows the Gerrit web interface with a modal dialog for a code review. The main page in the background displays a change detail for a commit with Change-ID I76f81f28168da05e3438195f67433a4135d978c4. The modal dialog is titled "Code-Review:" and contains the following sections:

- Code-Review:**
 - +2 Looks good to me, approved
 - +1 Looks good to me, but someone else must approve
 - 0 No score
 - 1 I would prefer that you didn't submit this
 - 2 Do not submit
- Cover Message:** A large text area with a red border where the user can enter a message.

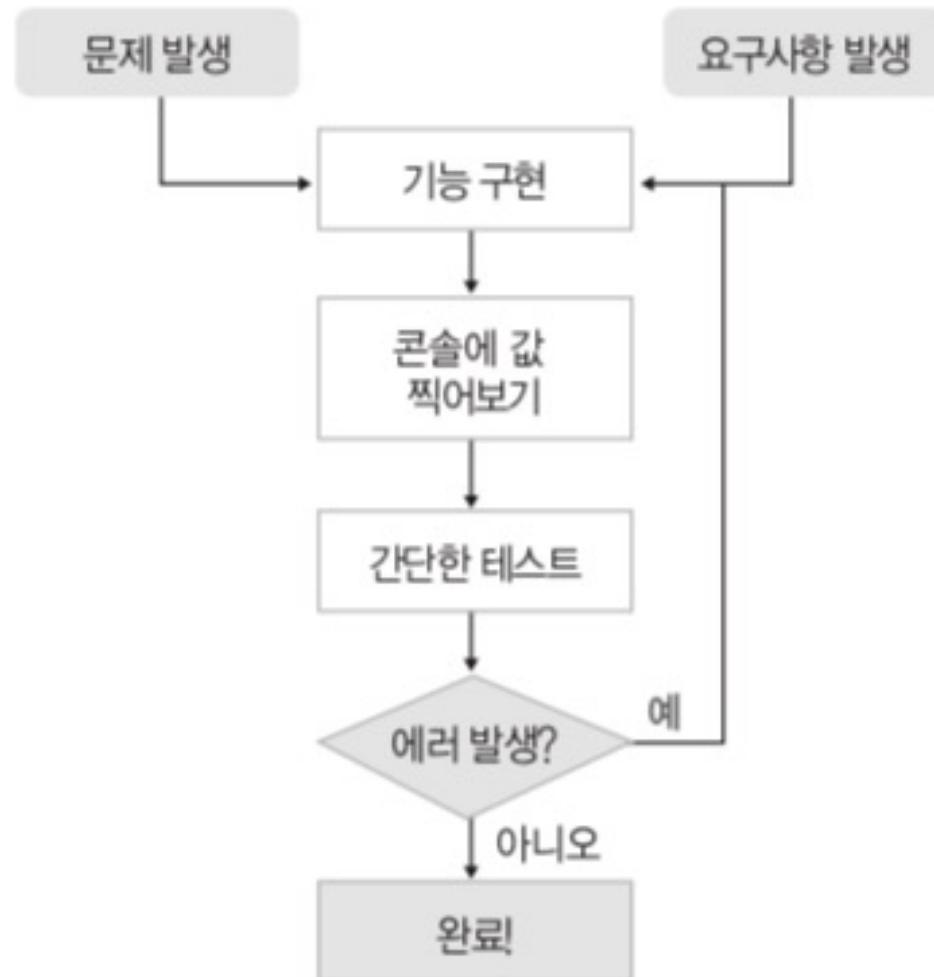
At the bottom of the dialog are two buttons: "Publish Comments" and "Cancel".



TDD

Test Driven Development

■ 기존 소프트웨어 개발 방식



- 특정 모듈의 개발 기간이 길어질수록 개발자의 목표의식이 흐려진다.
- 작업 분량이 늘어날수록 확인이 어려워진다.
- 개발자의 집중력이 필요해진다.
- 논리적인 오류를 찾기가 어렵다.
- 코드의 사용 방법과 변경 이력을 개발자의 기억력에 의존하게 되는 경우가 많다.
- 테스트 케이스가 적혀 있는 엑셀 파일을 보면 매번 테스트를 실행하는 게 점점 귀찮아져서는 점차 간소화하는 항목들이 늘어난다.
- 코드 수정 시에 기존 코드의 정상 동작에 대한 보장이 어렵다.
- 테스트를 해보려면 소스코드에 변경을 가하는 등, 번거로운 선행 작업이 필요할 수 있다.
- 그래서 소스 변경 시 해야 하는 회귀 테스트는 곧잘 희귀 테스트(rare test)가 되기 쉽다.
- 이래저래 테스트는 개발자의 귀중한 노동력(man-month)을 적지 않게 소모한다.

Test Driven Development

“Test the program before you write it.”

- Kent Beck



Kent Beck (1961~)

- 미국 sw 엔지니어
- XP(eXtreme Programming) 창시자
- Eric Gamma와 함께 JUnit 개발
- “객체 지향 프로그래밍을 위한 패턴 언어의 사용” 발표

Test Driven Development

■ *XP(eXtreme Programming)*

- 요구의 변동이 심한 경우에 적합
- 모델링, 문서화 작업을 줄이고 개발 작업에 집중
- 품질 우선
- 유연성 중시
- 반복 점증 개발
- 적은 인원에 적합
- 4가치: 의사소통, 단순성, 피드백, 용기
- 5가치: 의사소통, 단순성, 피드백, 용기, 존중

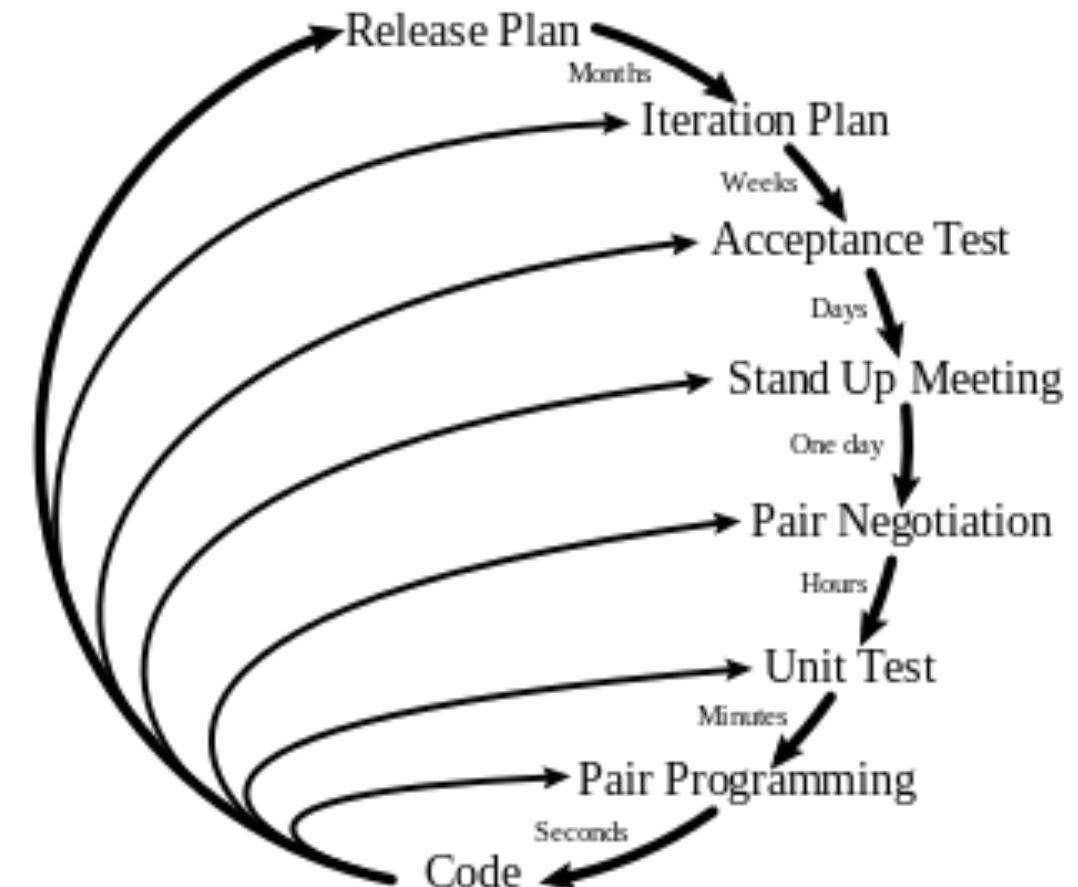
(communication, simplicity, feedback, courage, respect)

Test Driven Development

- **XP(*eXtreme Programming*)**

- 개발 사이클을 반복 수행
- 테스트 코드를 먼저 작성 ★★
- 조금씩 자주 배포
- 지속적인 코드 개선
- 테스트 통과한 것만 배포
- 요건에 없는 것은 배제
- 야근 X. 정규근무시간에 충실
- 일은 단순하게 처리

Planning/Feedback Loops



Test Driven Development

■ *Agile software development*

- 가벼운 프로세스
- 협업+피드백 ★
- 민첩함, 능동적, 자발적, 형식에 구애 받지 않음
- 반복 점진 개발 + 품질 개선 활동
 - 짧은 기간 단위의 **반복 절차를 통해 리스크를 줄임**
 - 개발 주기(계획, 개발, 출시)가 여러 번 반복
- 고객의 피드백에 민첩하게 반응
- 문서작업 줄이고 프로그래밍에 집중
 - 프로그래밍에 집중하는 유연한 개발 방식

Test Driven Development

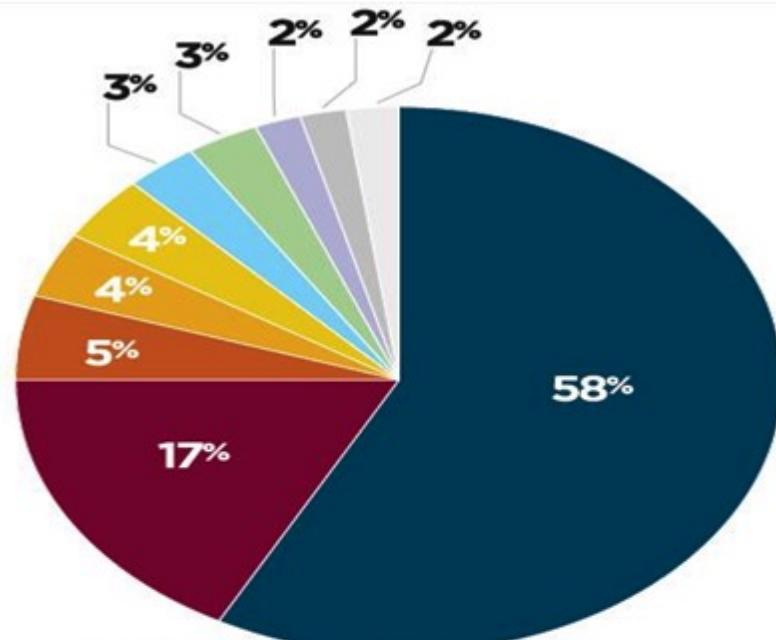
- ***Test First Development + Refactoring***

“Clean code that works”

- Ron Jeffries

Test Driven Development

- Agile 개발 방식 하나인 XP의 실천 방식

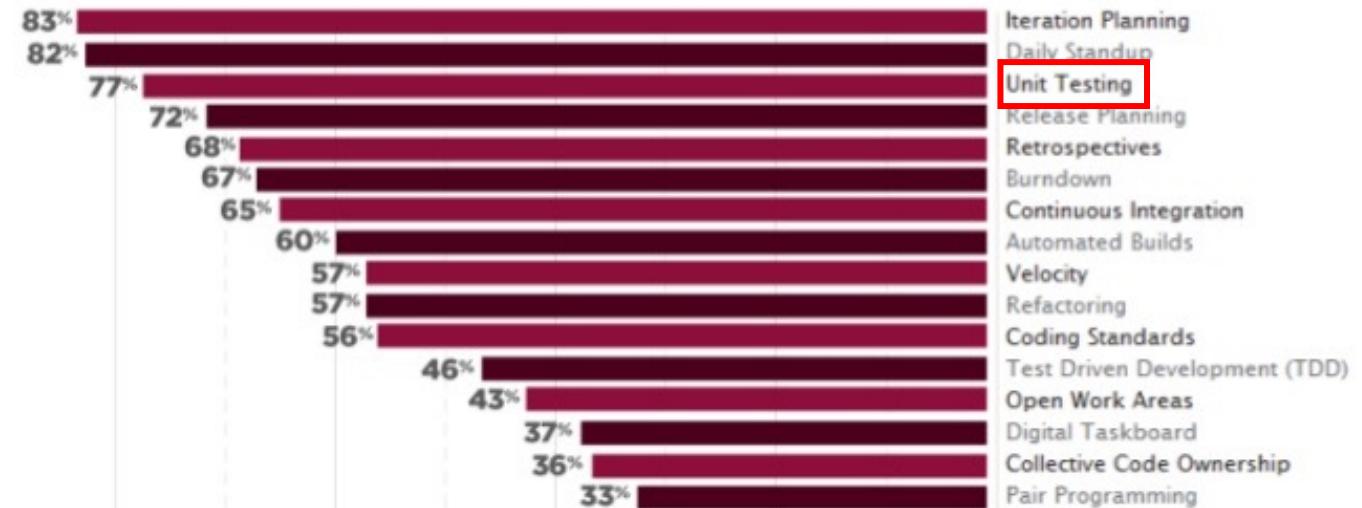


AGILE METHODOLOGY MOST CLOSELY FOLLOWED

Scrum or Scrum variants were by far the most common agile methodologies employed.

- SCRUM
- SCRUM/XP HYBRID
- CUSTOM HYBRID
- OTHER
- EXTREME PROGRAMMING (XP)
- DON'T KNOW
- SCRUMBAN
- LEAN
- FEATURE DRIVEN DEVELOPMENT (FDD)
- AGILEUP

Agile techniques employed



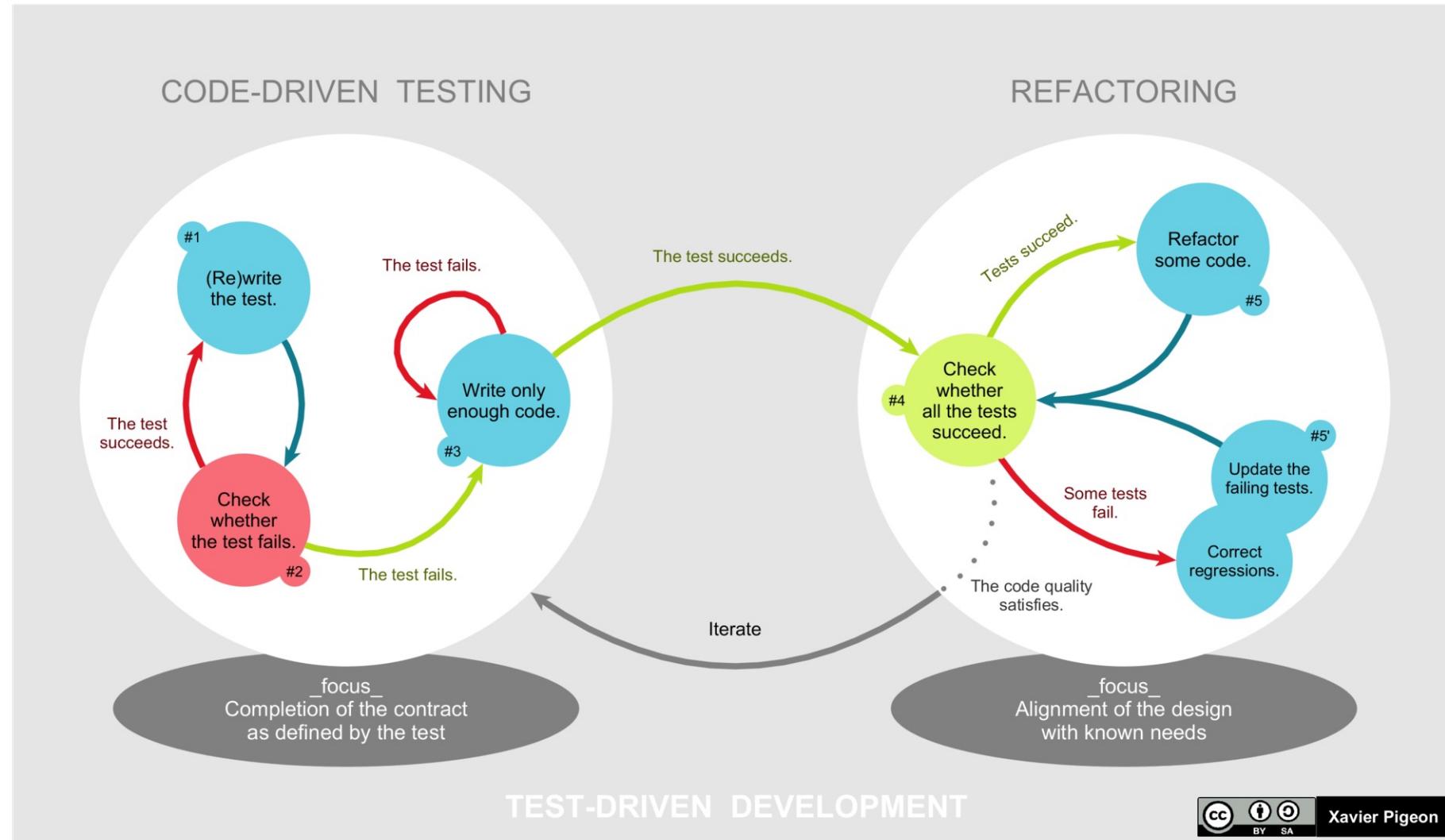
Test Driven Development

- 테스트 주도 개발의 진행 방식

- 질문(Ask)
 - 테스트 작성을 통해 시스템에 질문 → **테스트 수행 결과는 실패**
- 응답(Respond)
 - 테스트를 통과하는 코드를 작성해서 질문에 대답 → **테스트 성공**
- 정제(Refine)
 - 아이디어를 통합하고, 불필요한 것은 제거
 - 모호한 것은 명확히 해서 대답을 정제 → **리팩토링**
- 반복(Repeat)
 - 다음 질문을 통해 대화를 계속 진행

Test Driven Development

▪ 테스트 주도 개발의 진행 방식



Test Driven Development

■ 실습1

- TDD의 기본적인 진행 방식을 학습
- 시간을 절약시켜줄 IDE의 관련 기능 사용
- TDD를 진행할 때 발생 할 수 있는 몇가지 기본적인 고려사항 확인

- 1) 계좌 생성 테스트
- 2) 잔고 조회 테스트
- 3) 입금과 출금 테스트

Test Driven Development

- **실습1**

- 1) 계좌 생성 **Test**

- 클래스 명: Account

```
import org.junit.Test;  
  
public class AccountTest {  
    public void testAccount() {  
        Account account = new Account();  
        if (account == null) {  
            throw new Exception("계좌생성 실패");  
        }  
    }  
}
```



- **Compile Error!**
- **Account 생성**
- **테스트 실패**

Test Driven Development

■ 실습1

- 1) 계좌 생성 테스트
- 2) 잔고 조회 테스트
- 3) 입금과 출금 테스트

2) 잔고 조회 *Test*

- 잔고 조회
 - 10000원으로 계좌 생성
 - 잔고 조회 결과 일치

```
@Test  
public void testGetBalance() throws Exception {  
    Account account = new Account();  
    if (account.getBalance() != 10000) {  
        fail();  
    }  
}
```

Test Driven Development

■ 실습1

1) 계좌 생성 테스트

2) 잔고 조회 테스트

- 10000, 1000, 0원으로 계좌 생성
- 잔고 조회 결과 일치

3) 입금과 출금 테스트

- assertTest() 메소드 사용

```
@Test  
public void testGetBalance() throws Exception {  
    Account account = new Account(10000);  
    assertEquals(10000, account.getBalance());  
  
    account = new Account(1000);  
    assertEquals(1000, account.getBalance());  
  
    account = new Account(0);  
    assertEquals(0, account.getBalance());  
}
```

Test Driven Development

■ 실습1

1) 계좌 생성 테스트

2) 잔고 조회 테스트

- 10000, 1000, 0원으로 계좌 생성
- 잔고 조회 결과 일치

3) 입금과 출금 테스트

3) 입금과 출금 *Test*

- 입금, 출금 기능 선언

```
public void deposit(int i) {
    // TODO Auto-generated method stub
}
```

```
public void withdraw(int i) {
    // TODO Auto-generated method stub
}
```

```
@Test
public void testDeposit() throws Exception {
    Account account = new Account(10000);
    account.deposit(1000);
    assertEquals(11000, account.getBalance());
}
```

```
@Test
public void testWithdraw() throws Exception {
    Account account = new Account(10000);
    account.withdraw(1000);
    assertEquals(9000, account.getBalance());
}
```

Test Driven Development

■ 실습1

1) 계좌 생성 테스트

2) 잔고 조회 테스트

- 10000, 1000, 0원으로 계좌 생성
- 잔고 조회 결과 일치

3) 입금과 출금 테스트

3) 입금과 출금 *Test*

- 입금, 출금 기능 구현

```
public void deposit(int i) {  
    this.balance += i;  
}
```

```
public void withdraw(int i) {  
    this.balance -= i;  
}
```

Test Driven Development

■ 실습1

1) 계좌 생성 테스트

2) 잔고 조회 테스트

- 10000, 1000, 0원으로 계좌 생성
- 잔고 조회 결과 일치

3) 입금과 출금 테스트

- setUp(), tearDown()
- @Before, @After

```
private Account account;
```

```
@Before
```

```
public void setUp() {  
    account = new Account(10000);  
}
```

Test Driven Development

■ 장점

- 개발의 방향을 잃지 않게 유지해준다
- 품질 높은 소프트웨어 모듈 보유
- 자동화된 단위 테스트 케이스를 갖게 된다
- 사용설명서 & 의사소통의 수단
- 설계 개선
- 보다 자주 성공한다

■ 로버트 마틴 (Robert C. Martin) TDD 원칙

- 1) 실패하는 테스트를 작성하기 전에는 절대로 제품 코드를 작성하지 않는다.
- 2) 실패하는 테스트 코드를 한 번에 하나 이상 작성하지 않는다.
- 3) 현재 실패하고 있는 테스트를 통과하기에 충분한 정도를 넘어서는 제품 코드를 작성하지 않는다.

■ xUnit 프레임워크

- 1998년 Kent Beck이 개발
- Unit Testing Framework
 - 모든 함수와 메소드에 대한 Test Case를 작성하는 절차
- 장점
 - 문제점 발견이 쉬움
 - 변경이 쉬움 → Refactoring
 - **Refactoring**: 결과 값은 바뀌지 않으면서 코드의 구조 변경
 - Regression Testing
 - 통합이 간단
- Junit, Cunit, CppUnit, PyUnit, NUnit

- **xUnit 프레임워크**

- ***Test Runner* → 테스트 작업을 수행**
- Test Case
- ***Test Fixture* → 테스트를 위한 선조건 → 구현한 것이 Test Case Class**
- Test Suite
- Test Execution
- Test Result Formmater
- ***Assertion* → 테스트 결과가 예상과 같은지를 판별**

xUnit

- **xUnit 프레임워크**

- ***Test Runner***
- Test Case
- ***Test Fixture***
- Test Suite
- Test Execution
- Test Result Formmater
- ***Assertion***

■ xUnit 프레임워크

- Test Fixture

- System Under Test를 실행하기 위해 필요한 모든 것
- 매번 동일한 결과를 얻을 수 있도록 ‘**기반이 되는 상태나 환경**’

→ **일관된 테스트 실행 환경 (Test Context)**

xUnit

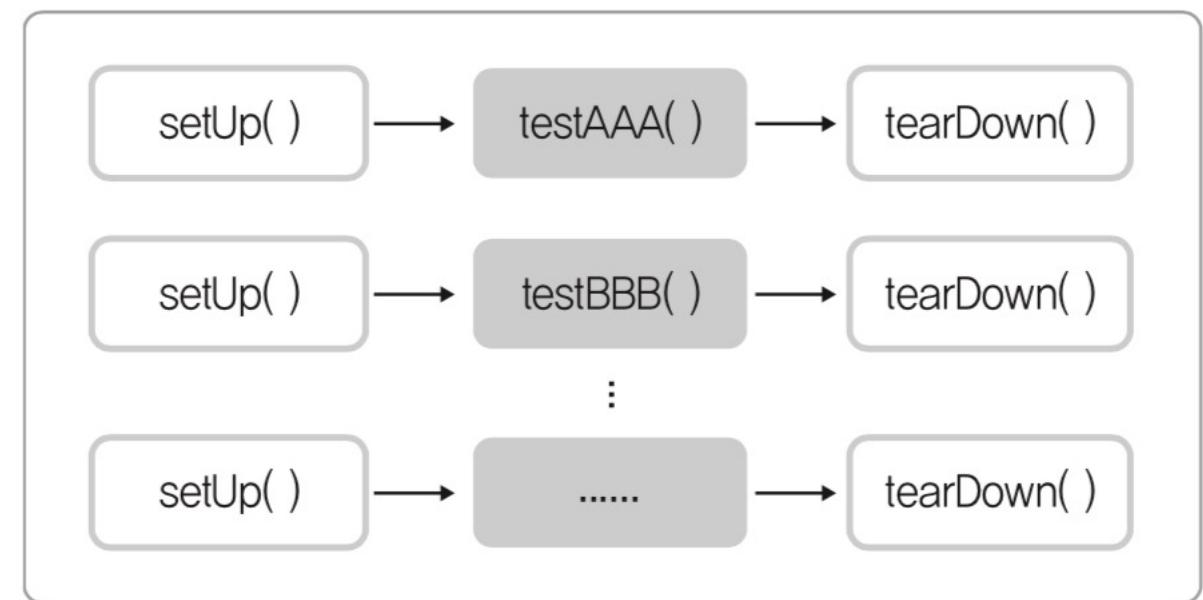
- **JUnit**

- **JUnit 3**
- **JUnit 4**
- **JUnit 5**

- **JUnit 3**

- TestCase 상속
- 테스트 메소드는 test로 시작
- setUp(), tearDown()

TestClassA



■ JUnit 3

- ***assertEquals***([message], expected, actual)
- ***assertTrue***([message], expected)
- ***assertFalse***([message], expected)
- ***assertNull***([message], expected)
- ***assertNotNull***([message], expected)
- ***fail***([message])

- **JUnit 4 특징**

- 1) Java 5 애노테이션 지원
- 2) test라는 글자로 method 이름을 시작해야 한다는 제약 해소
- 3) 좀 더 유연한 픽스처
 - @BeforeClass, @AfterClass, @Before, @After
- 4) 예외 테스트
 - @Test(expected=NumberFormatException.class)
- 5) 시간 제한 테스트
 - @Test(timeout=1000)

■ JUnit 4 특징

6) 테스트 무시

- @Ignore("this method isn't working yet")

7) 배열 지원

- assertEquals([message], expected, actual);

8) @RunWith(클래스이름.class)

- JUnit Test 클래스를 실행하기 위한 러너(Runner)를 명시적으로 지정

9) @SuiteClasses(Class[])

- 보통 여러 개의 테스트 클래스를 수행하기 위해 사용

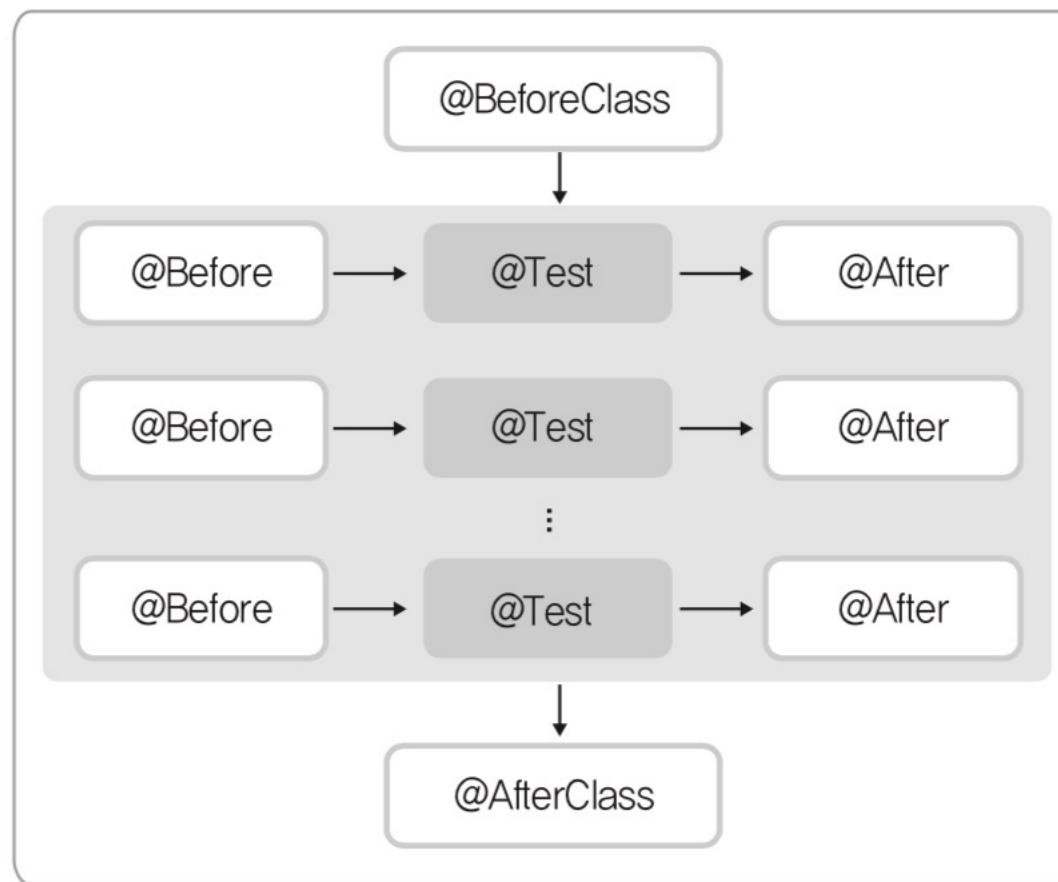
10) 파라미터를 이용한 테스트

- @RunWith(Parameterized.class)

@Parameters

▪ 테스트 퍽스처 메소드 추가 지원

TestClassB



@BeforeClass

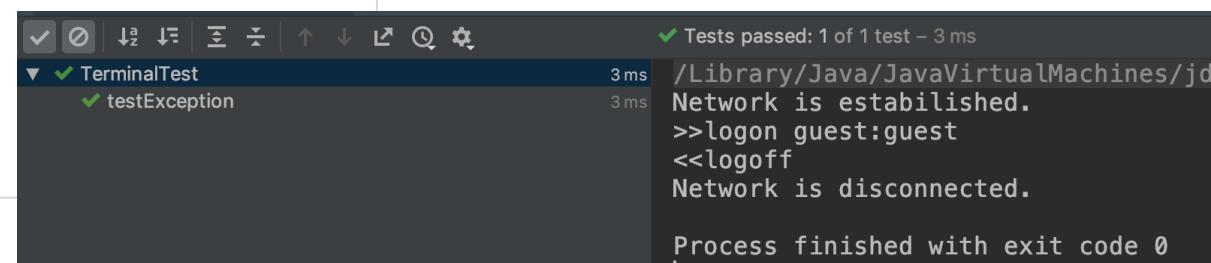
```
public static void setUpBeforeClass() throws Exception {  
    term = new Terminal();  
    term.netConnect(); // 터미널에 접속한다.  
}
```

@AfterClass

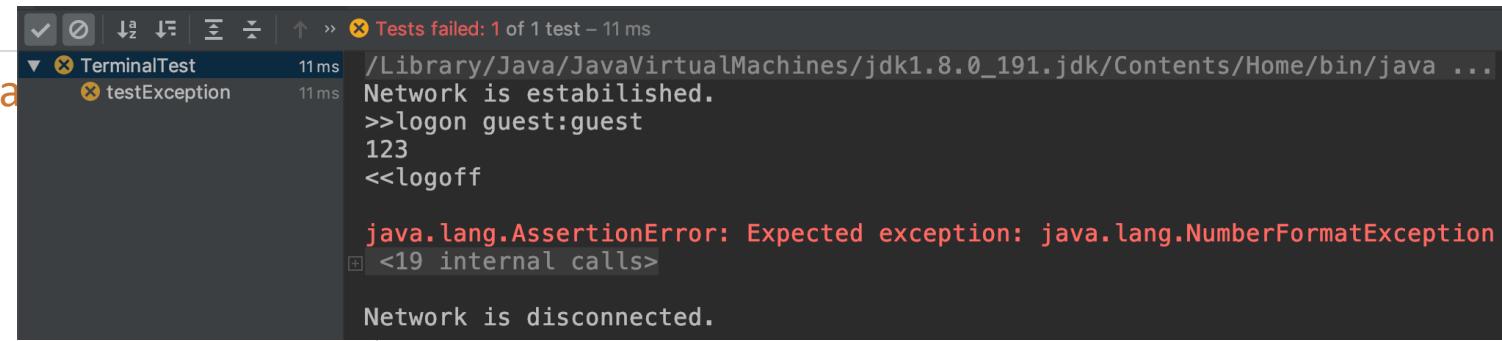
```
public static void tearDownAfterClass() throws Exception {  
    term.netDisconnect(); // 터미널과의 연결을 해제한다.  
}
```

■ 예외 테스트

```
public void testException(){
    String value = "test";
    try {
        System.out.println(Integer.parseInt(value));
        assertTrue(false);
    } catch (NumberFormatException nfe){
        assertTrue(true);
    }
}
```

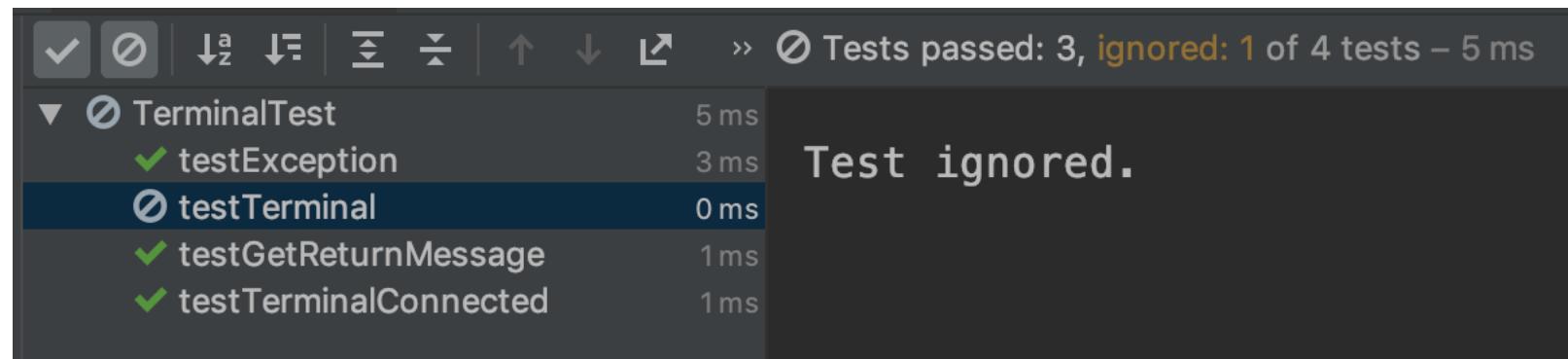


```
@Test (expected=NumberFormatException.class)
public void testException(){
    String value = "test";
    System.out.println(Integer.parseInt(value));
}
```



▪ 테스트 무시

```
@Ignore  
@Test  
public void testTerminal() throws Exception {  
    assertTrue (term.isLogon());  
    System.out.println("== logon test");  
}
```



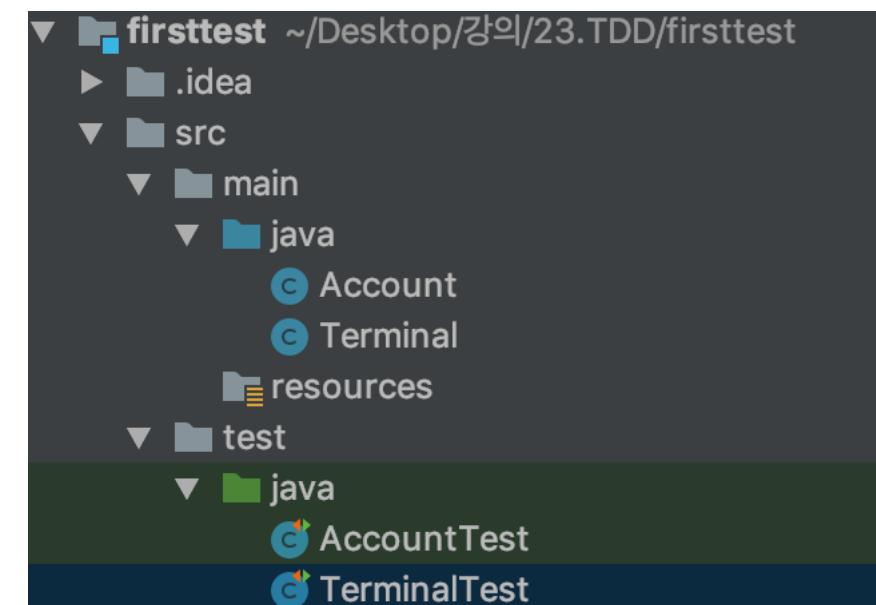
TDD 테스트 케이스 작성

- **테스트 클래스 위치**

- 1) 테스트 대상 소스와 테스트 클래스를 같은 곳에 저장
- 2) 테스트 클래스를 하위 패키지에 저장
- 3) 최상위 패키지를 분리

4) 소스 폴더와 컴파일 된 클래스 위치 분리, 같은 패키지 사용

- Maven Project
 - /src/main/java
 - /src/main/resources/
 - /src/test/java
 - /src/test/resources/



TDD 테스트 케이스 작성

■ 테스트 메소드 작성 방식

1) 테스트 대상 메소드와 이름을 1:1로 일치

```
public int getBalance() { ... }
```

```
@Test  
public void testGetBalance(){ ... }
```

2) 테스트 대상 메소드의 이름 뒤에 추가적인 정보 기재

```
public void withdraw(int money) { ... }
```

```
@Test  
public void testWithdraw_마이너스통장인출(){ ... }  
  
@Test  
public void testWithdraw_잔고가0원일때(){ ... }
```

3) 테스트 시나리오 표시

특정한 메소드를 대상으로 하기보다는 테스트 시나리오가 대상이 된다.

```
@Test  
public void VIP고객이_인출할때_수수료계산(){ ... }
```

TDD 테스트 케이스 작성

▪ 테스트 케이스 작성 접근 방식

- 1) 설계자와 개발자가 분리되어 있는 경우
- 2) 개발자가 설계와 개발을 함께 하는 경우

Backlog Item #34

09. 기능 공통 : 공통 : 첨부파일

파일 업로드/다운로드 기능을 구현한다.

스토리
포인트

8

스프린트
#1

테스트 케이스

- hwp, doc, xls, ppt만 첨부 가능하다.
- exe, bat, com 등의 확장자를 등록하려 할 때 보안 모듈을
호출한다.
- 10M 이상의 파일은 업로드할 수 없다.

Practice #1

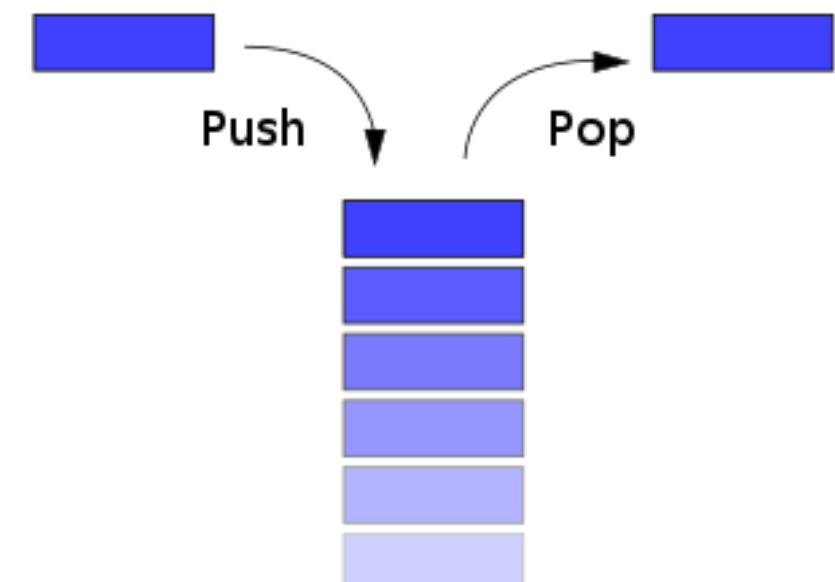
- ***Reverse Polish Calculator***

- 역 폴란드 표기법을 이용한 계산기
- 컴퓨터 프로그래밍으로 계산기 구현 시 내부 연산으로 사용
- 수식을 앞에서부터 읽어 나가면서 **Stack**에 저장
- 중위 표기법에서는 연산자의 **우선순위가 모호**해서 괄호가 필요

ex1) $1 + 2 \rightarrow 1\ 2\ +$

ex2) $(2 + 3) * 4 \rightarrow 2\ 3\ +\ 4\ *$

ex3) $(3 + 5) * (4 + 2) \rightarrow ??$



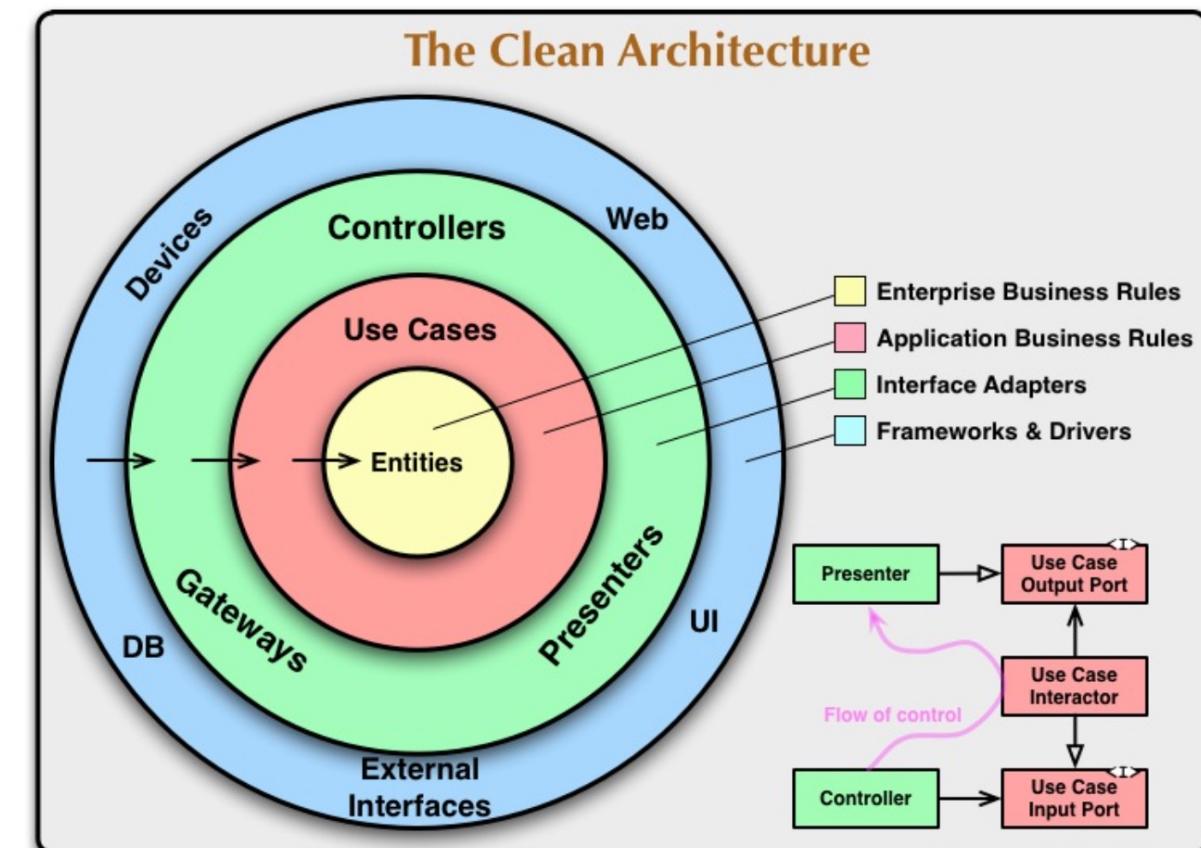


Clean Architecture

Clean Architecture?

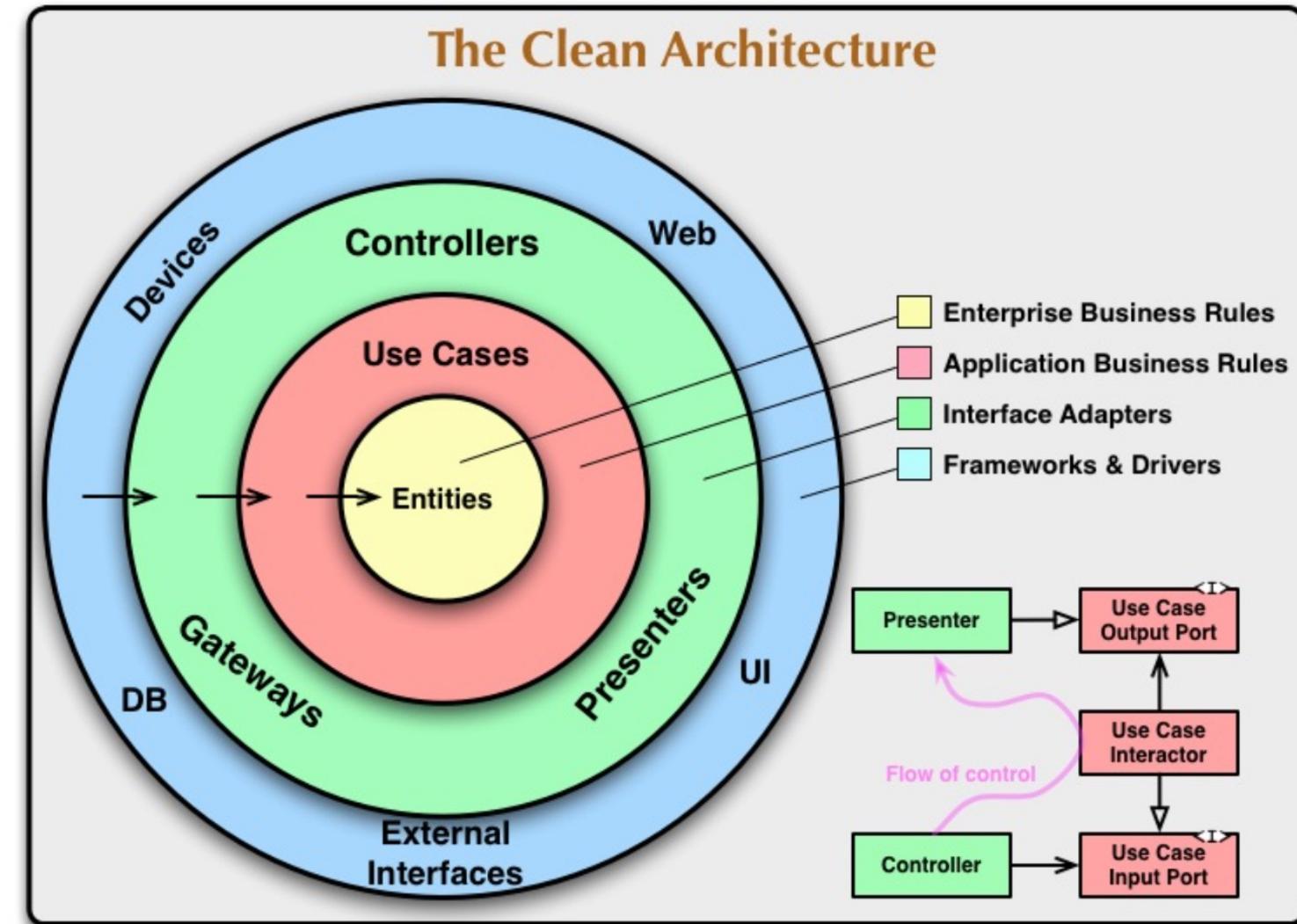
- ***The Clean Architecture*** (<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>)

- Robert C. Martin
- SOLID(객체지향 설계 원칙)
- 관심의 분리와 테스트 가능성을 요구 → 추상화 개념으로 관심사를 분리시키고 의존도를 낮추는 것에 목적은 둔 아키텍처



Clean Architecture?

- *Entities*
- *Use cases*
- *Interface Adapter*
- *Frameworks & Drivers*



Clean Architecture?

■ Dependency rule

- 종속성의 흐름을 제어
- 캡슐화를 통한 재사용

SOLID software design principles

	PRINCIPLE	DESCRIPTION
S	Single responsibility	A class should have one, and only one, reason to change, which means it should have only one function.
O	Open/closed	Software objects should be open to extension, but closed for modification.
L	Liskov substitution	Objects of the same type should be replaceable with others from the same category without altering the function of the program.
I	Interface segregation	No client should be forced to depend on methods it does not use. The program's interfaces should always be kept smaller and separate from one another.
D	Dependency inversion	High-level modules should not depend on low-level modules, but both should depend on abstractions. While abstractions should not depend on details, details should depend on abstractions.

종속성의 흐름을
제어

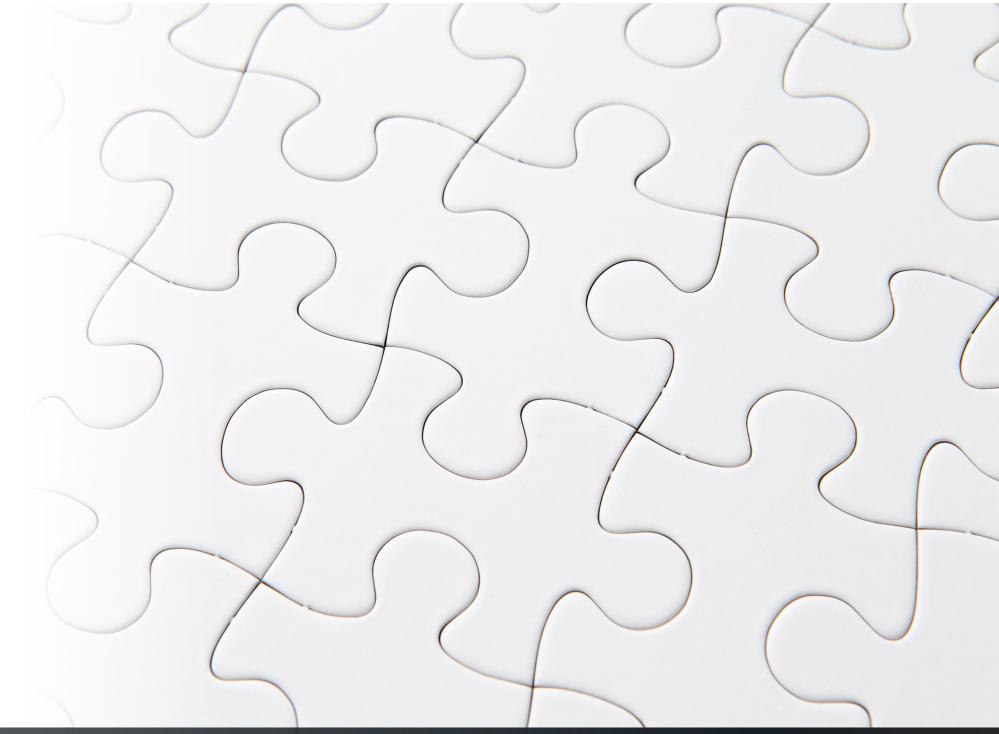
부 사항을 구현

■ Abstraction

- 가장 안쪽에

■ SOLID

- 확장가능하며



Thank you