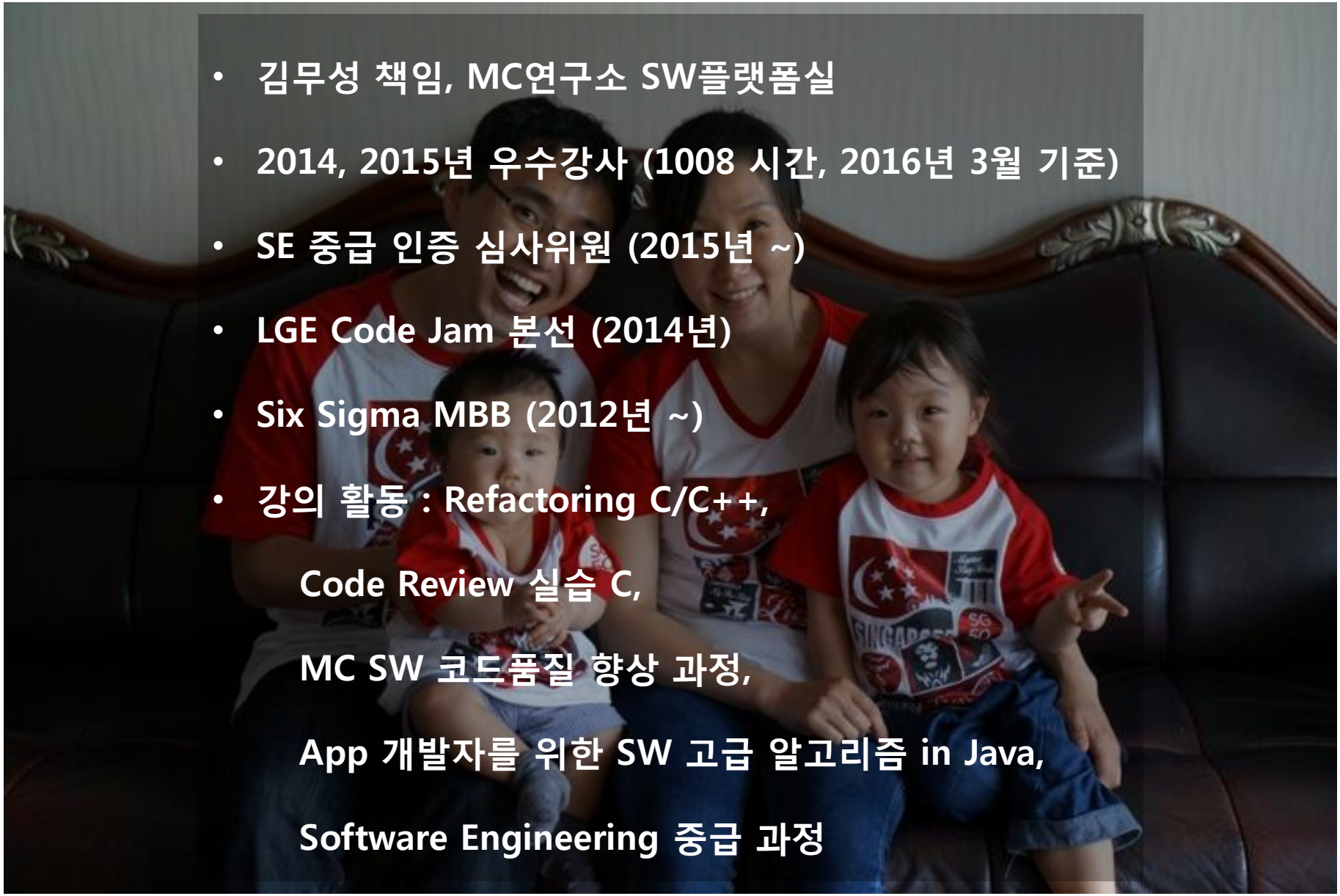
Refactoring C++



- 김무성 책임, MC연구소 SW플랫폼실
- 2014, 2015년 우수강사 (1008 시간, 2016년 3월 기준)
- SE 중급 인증 심사위원 (2015년 ~)
- LGE Code Jam 본선 (2014년)
- Six Sigma MBB (2012년 ~)
- 강의 활동 : Refactoring C/C++,
  Code Review 실습 C,
  MC SW 코드품질 향상 과정,
  App 개발자를 위한 SW 고급 알고리즘 in Java,
  Software Engineering 중급 과정

강의 개요

과정명	Refactoring (C++)	구분	필수 □ 선택 ■ 선발 □	인정학점	3
과정개요	C++ 로 프로그램을 작성할 때 Refactoring 을 적용하고 코드 품질을 향상시키는 기법을 학습한다.				
학습목표	• Technical Debt의 개념과 Refactoring과의 관계를 이해한다. • 바람직한 C++ 코드 패턴과 부적절한 C++ 코드 패턴의 차이를 구별할 수 있다. • 프로그램을 작성할 때 바람직한 코드 패턴으로 작성할 수 있다. • 다른 사람이 작성한 코드를 리뷰할 때 리뷰 포인트를 찾을 수 있다.				
대상자	C++ 개발경력 2년 이상	선수 지식	C++, SW Testing		
선수과정	C++ Programming	사후과정	n/a		
이수기준	출석률 % & 사후평가 점	테스트 유무	입과 □ 사전 □ 사후 □		
준비물		기간	3 일 / 24 시간	언어	한국어
장소		비용			

잠깐 앙케이트 합니다

1. 본인의 교육 목표
2. Refactoring 활동에 관해 평소 궁금했던 것
3. 과정 내용에 관한 궁금한 점

10분

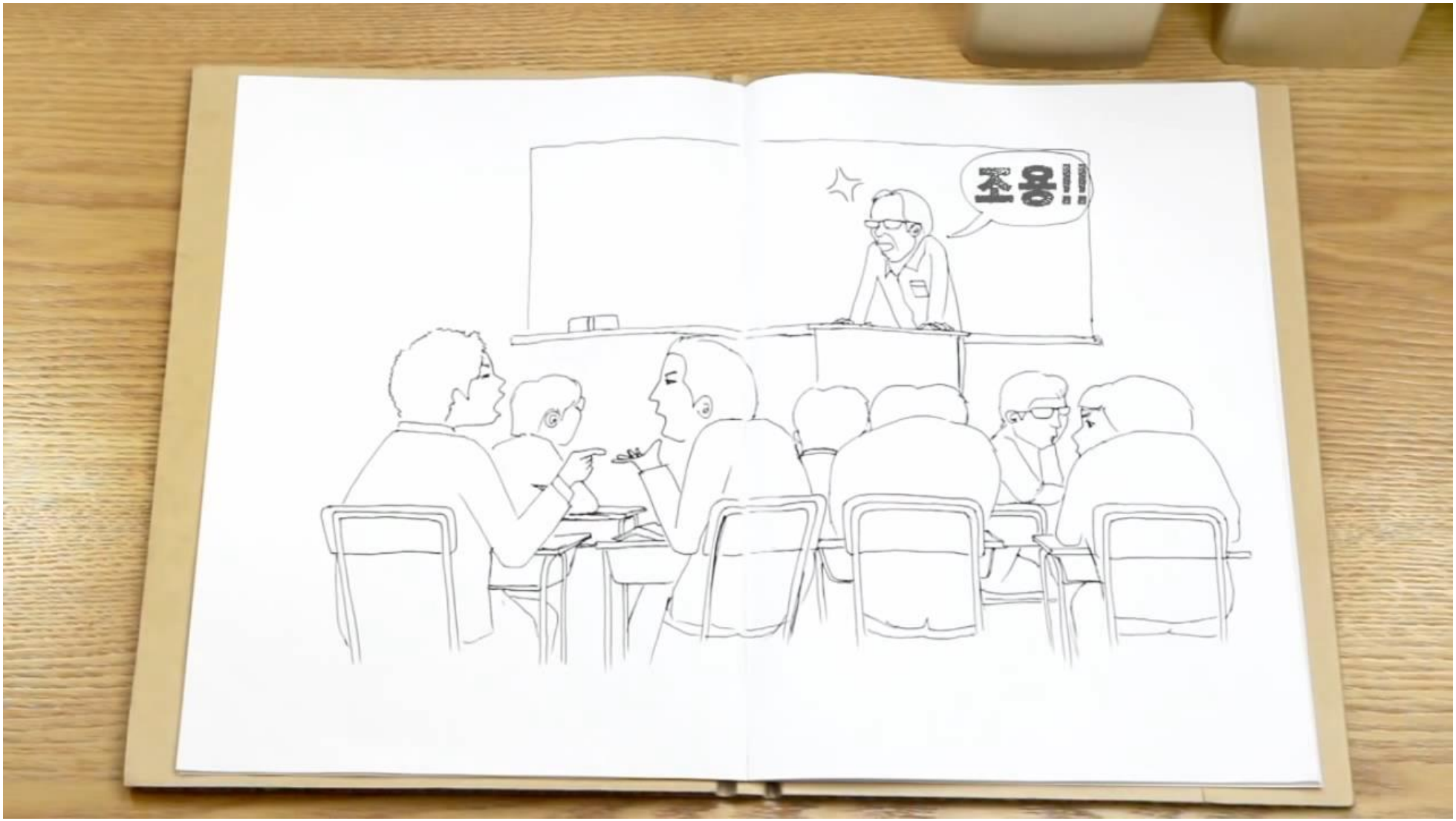
Technical Debt

- 학습목표 1
- 학습목표 2

(1) Technical Debt 정의

(2) Technical Debt 관리

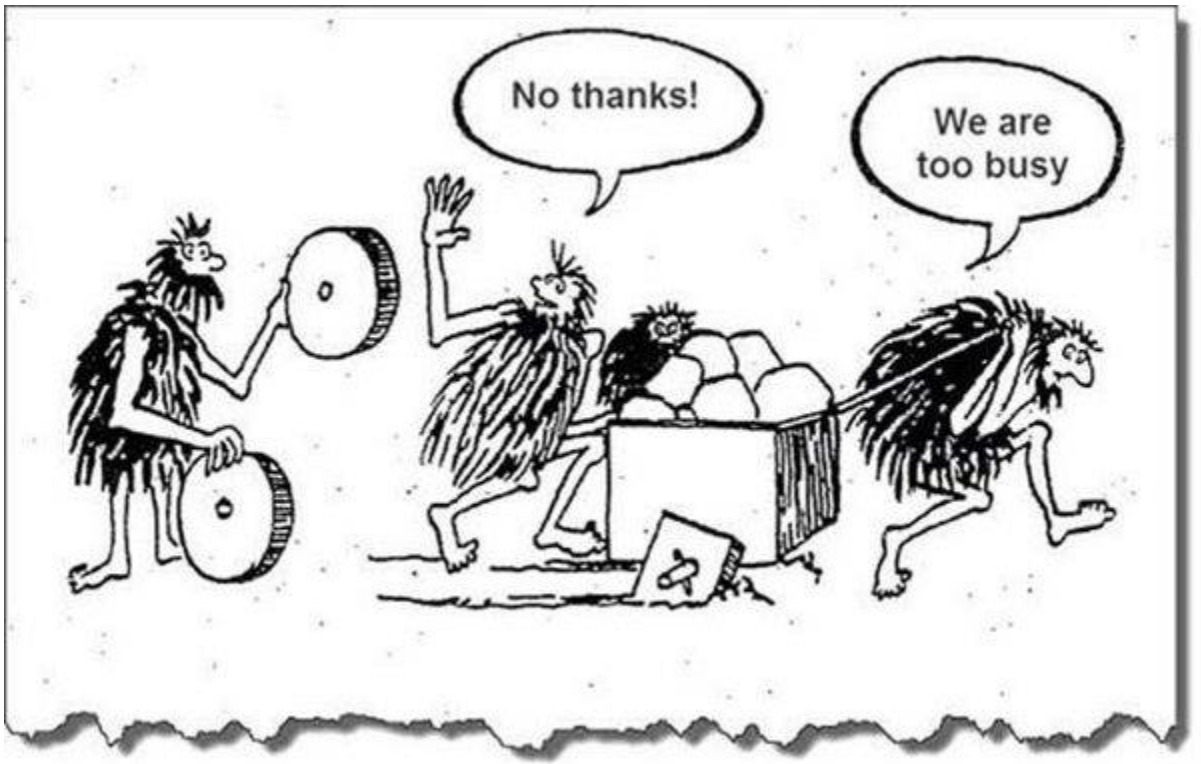
소스 코드 품질



https://www.youtube.com/watch?v=S8kVJUUrBpQ

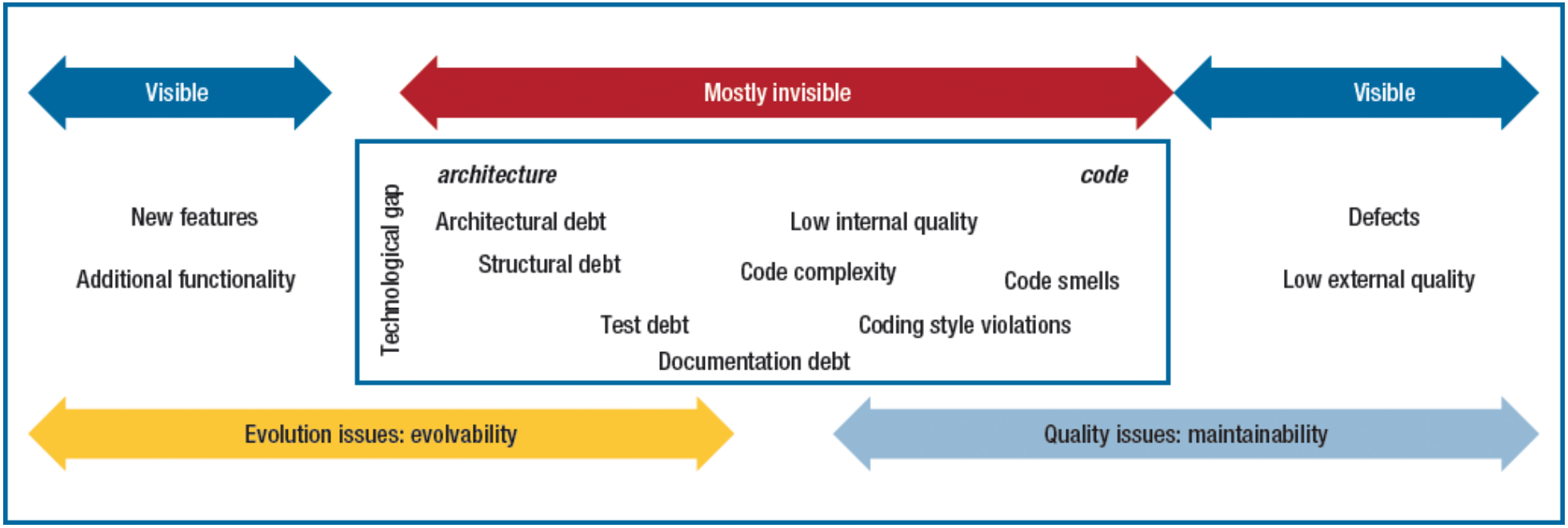
Technical Debt 정의

Technical Debt(=Design Debt, Code Debt)는 전체적으로 최상의 방법 대신에 단기적으로 구현이 쉬운 방법을 선택함으로서 발생한 나중으로 미뤄둔 작업을 나타내는 프로그래밍에서의 개념이다.



https://en.wikipedia.org/wiki/Technical_debt

Technical Debt의 종류

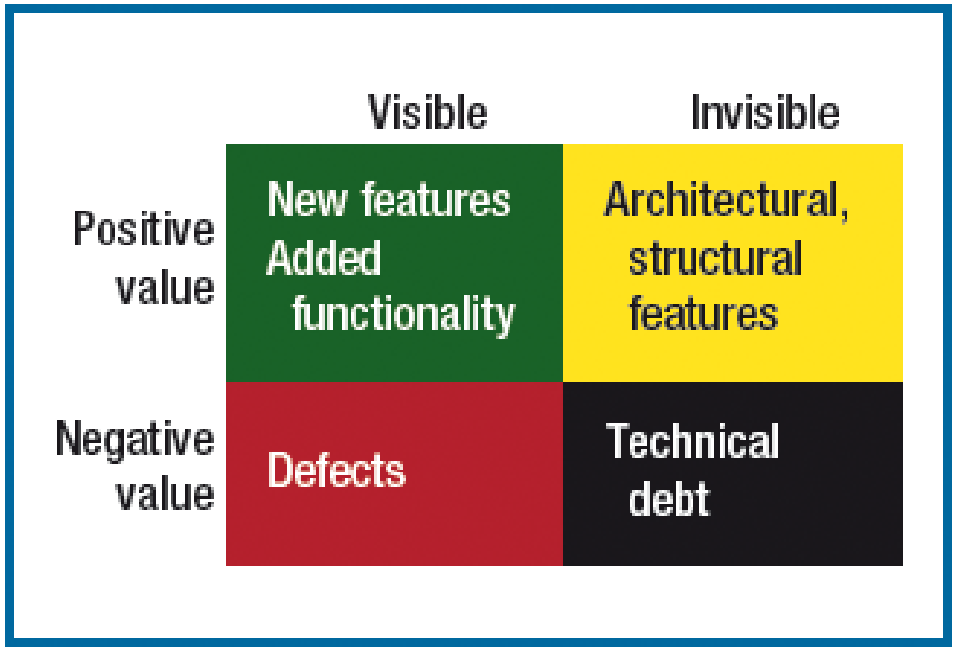


SW 개선의 종류
- 진화와 도전을 위한 새로운 기능들
- 유지보수를 위한 품질 개선

대부분의 Technical Debt는 보이지 않는다.
- 하지만 중요하다.

https://www.computer.org/csdl/mags/so/2012/06/mso2012060018.html

업무에서의 Technical Debt



- 녹색 Backlog : 새로운 기능을 추가, 프로젝트 작업의 대부분
- 노란색 Backlog : 신규 기능을 위한 설계 작업, 개발 초기에 주로 작성
- 빨간색 Backlog : (눈에 보이는) 부정적인 것을 해결하는 작업, 발생할 Defect를 해결하는 Backlog 확보
- 검은색 Backlog : Technical Debt를 해소하는 작업, 드문 케이스

https://www.computer.org/csdl/mags/so/2012/06/mso2012060018.html

Technical Debt의 원인

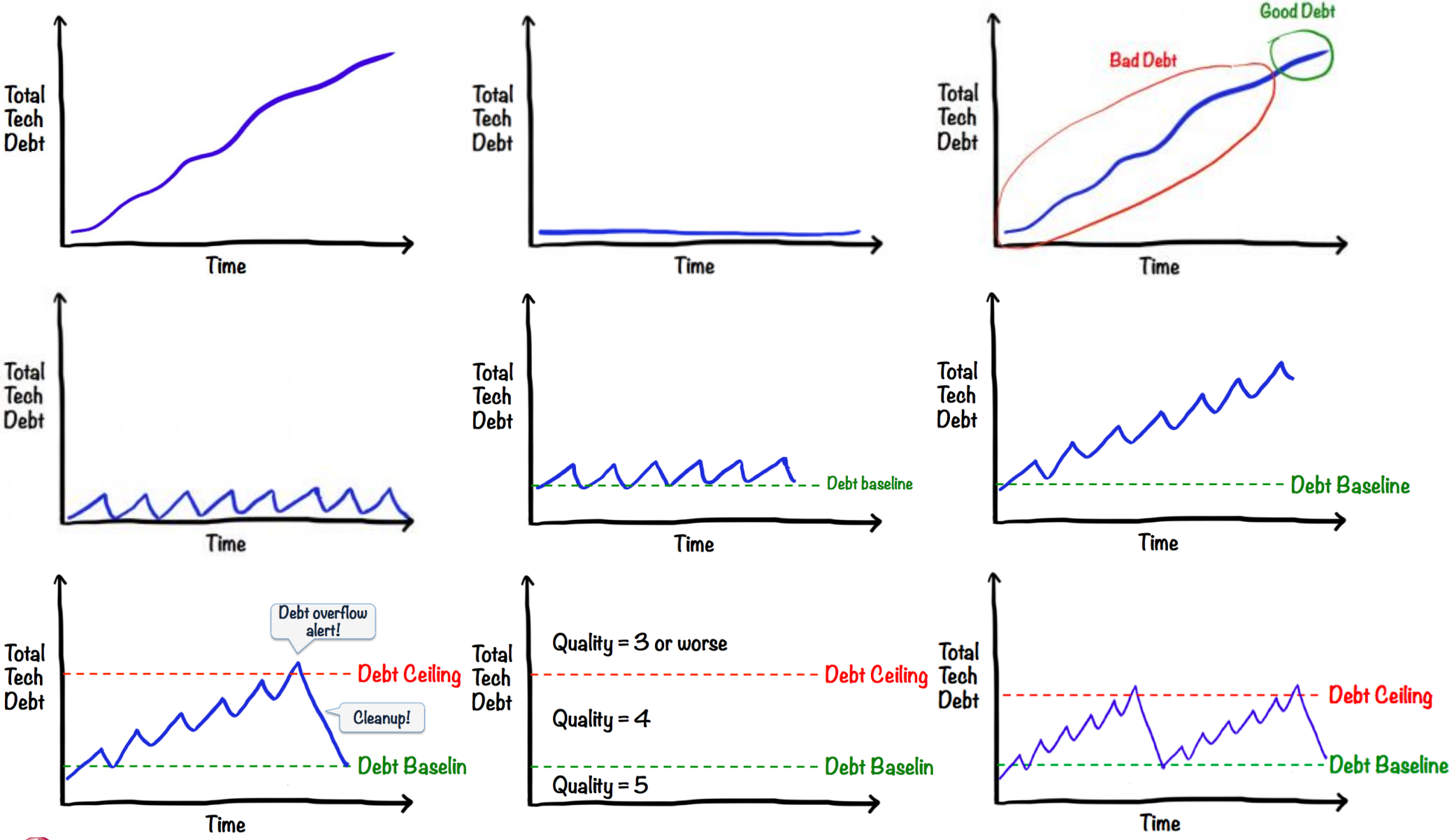
- Business pressures
- Lack of process or understanding
- Lack of building loosely coupled components
- Lack of a test suite
- Lack of documentation
- Lack of collaboration
- Parallel development
- Delayed refactoring
- Lack of alignment to standards
- Lack of knowledge
- Lack of ownership
- Poor technological leadership
- Last minute specification changes
- Scope Doping

https://en.wikipedia.org/wiki/Technical_debt

Technical Debt의 오해

- Technical debt에 대해서 오해하지 말아야 하는 점은 Technical debt가 없는 것이 좋은 프로젝트가 아니다.
- 실제 사업에서도 그렇듯이 적정 비율을 부채를 유지하면서 그 비용을 다른 곳에 투자하는 것이 오히려 많은 이익을 낼 수 있다.
- 소프트웨어 개발에서도 Technical debt를 발생시키면서 남는 잉여 리소스를 다른 곳에 투자함으로써 좋은 효과를 낼 수 있다.
- 중요한 것은 “이 Technical debt의 비율을 어느 정도로 유지할 것이냐” 이다.

Good and Bad Technical Debt



http://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt

Technical Debt를 줄이는 활동

Architecture

Test Driven Development

Testing

Code Review

Pair Programming

Refactoring

Code Inspection

Agile Scrum

Static Analysis

Continuous Integration

Technical Debt in Game

https://www.youtube.com/watch?v=AuFheQ9o7ZM

SW Testing

- 학습목표 1
- 학습목표 2

(1) SW Testing 정의

(2) Testing Method

(3) Code Coverage

교육 실습 환경

[eclipse 설치]
1. JRE : http://www.oracle.com/technetwork/java/javase/downloads/index.html
2. Eclipse IDE for C/C++ Developers : https://eclipse.org/

[cygwin 설치]
1. cygwin : http://cygwin.com/
2. 설치 중 gcc-g++, make 패키지 선택

[google test 소스 다운로드]
1. googletest : https://github.com/google/googletest
2. Download ZIP → 압축 풀기

[project 만들기]
1. eclipse → Import… → C/C++ → Existing Code as Makefile Project
2. googletest-master₩googletest₩make 폴더 입력
3. Toolchain for Indexer Settings → Cygwin GCC 선택
4. 윈도우 탐색기에서 samples 폴더를 드래그해서 project에 드랍
5. Link to files and recreate folder structure with virtual folders 선택
6. Compile & Run

Smoke Test

```
#include "gtest/gtest.h"

int Sum(int left, int right) {
    return left + right;
}

TEST(FirstTestCase, FirstTest) {
    int expected = 5;
    int actual = Sum(2, 3);
    EXPECT_EQ(expected, actual);
}
```

SW Testing 정의

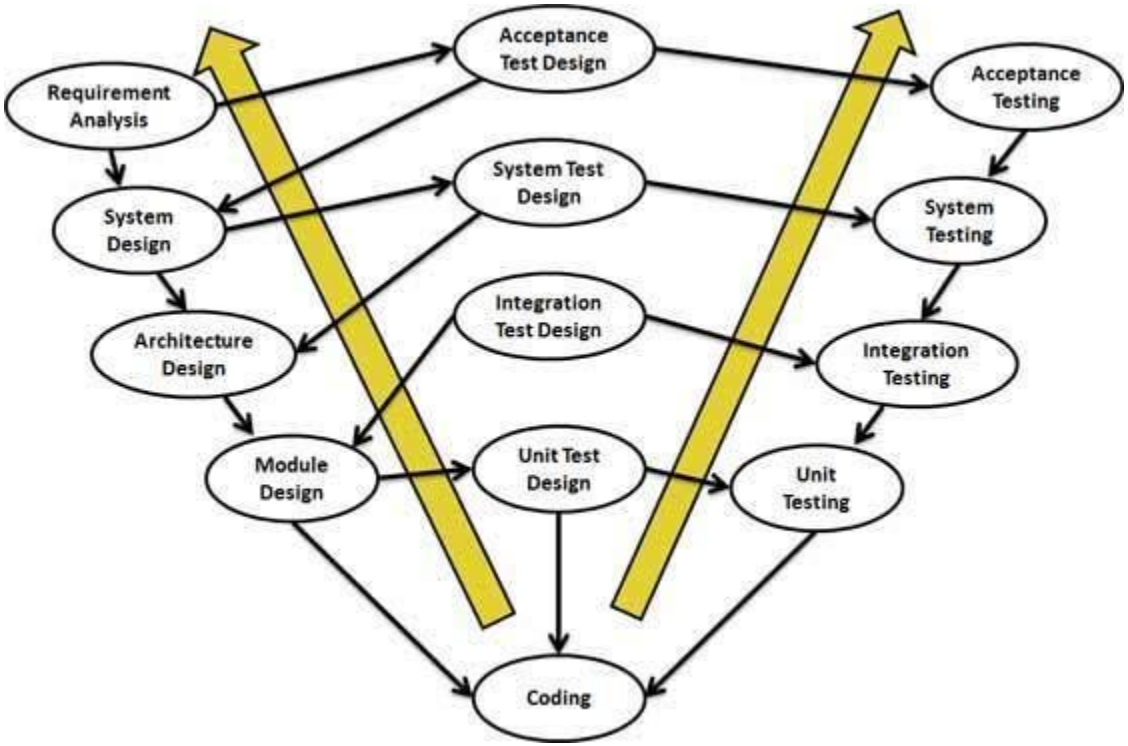
SW Testing은 이해당사자들에게 테스트를 통하여 제품이나 서비스의 품질 정보를 제공하는 조사 활동이다. 또한 소프트웨어 테스팅은 소프트웨어 구현 측면의 리스크를 확인하고 이해하는 업무를 위해 소프트웨어의 목표나 독립적인 시각을 제공할 수 있다.

Testing Levels
- Unit Testing
- Integration Testing
- Component Interface Testing
- System Testing
- Operational Acceptance Testing

Testing Methods
- Static vs Dynamic Testing
- The Box Approach
- Black-box Testing
- White-box Testing

https://en.wikipedia.org/wiki/Software_testing

Testing Levels



Unit Test는 컴퓨터 프로그래밍에서 소스 코드의 특정 모듈이 의도된 대로 정확히 동작하는지 검증하는 절차다. 즉, 모든 함수와 메소드에 대한 테스트 케이스를 작성하는 절차를 말한다.

http://www.tutorialspoint.com/sdlc/sdlc_v_model.htm
https://ko.wikipedia.org/wiki/유닛_테스트

Testing Methods

Specification Based Testing
- **Equivalence Partitioning**
- **Boundary Value Analysis**
- **State Transition Tables**
- **Decision Table Testing**
- **Use Case Testing**

Equivalence Partitioning
- **같은 방식으로 처리되고 결과값이 같은 집합(=Partition) 작성**
- **Partition에서 대표값을 선정하여 테스트 케이스 작성**
- **작은 테스트 케이스 수**

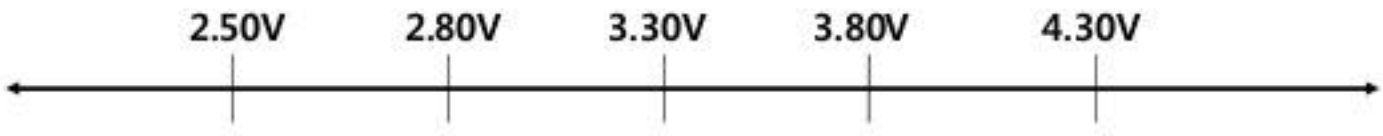
Boundary Value Analysis
- **Partition과 Partition의 경계에서 많은 결함이 발생**
- **경계값들에서 테스트 케이스 작성**
- **상세한 Spec 필요**

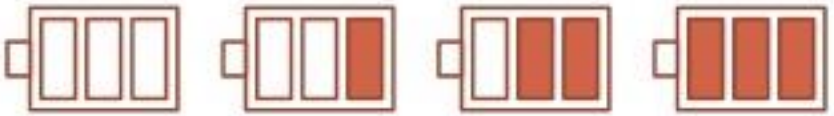
https://en.wikipedia.org/wiki/Equivalence_partitioning
https://en.wikipedia.org/wiki/Boundary_testing

Equivalence Partitioning

명세기반기법-동등분할 예제[1]

❖ 배터리 용량 표시





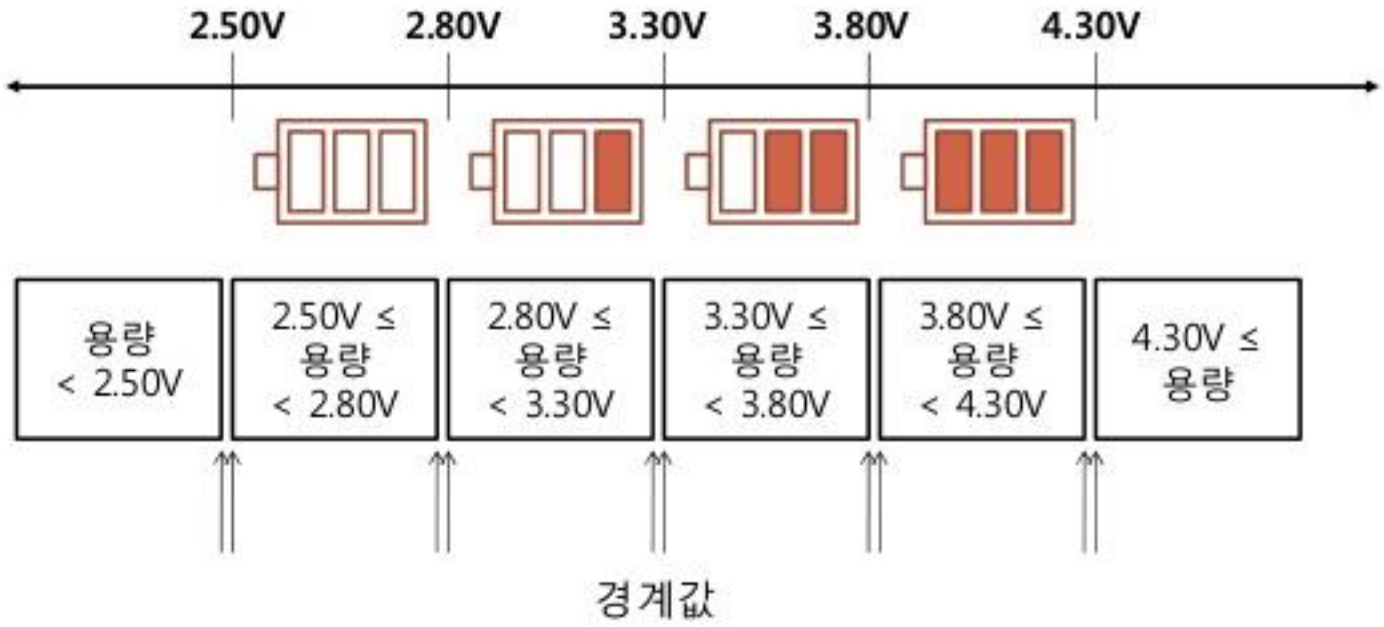
용량 < 2.50V	2.50V ≤ 용량 < 2.80V	2.80V ≤ 용량 < 3.30V	3.30V ≤ 용량 < 3.80V	3.80V ≤ 용량 < 4.30V	4.30V ≤ 용량

http://www.slideshare.net/daunjae/istqb-420151

Boundary Value Analysis

명세기반-경계값 분석 예제[1]

❖ 배터리 용량 예제



http://www.slideshare.net/daunjae/istqb-420151

Code Coverage 정의

In computer science, code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with high code coverage, measured as a percentage, has had more of its source code executed during testing which suggests it has a lower chance of containing undetected software bugs compared to a program with low code coverage. Many different metrics can be used to calculate code coverage; some of the most basic are the percent of program subroutines and the percent of program statements called during execution of the test suite.

https://en.wikipedia.org/wiki/Code_coverage

Boolean Algebra

AND 연산 (A && B = C)

A	B	C
True	True	True
True	False	False
False	True	False
False	False	False

OR 연산 (A || B = C)

A	B	C
True	True	True
True	False	True
False	True	True
False	False	False

https://en.wikipedia.org/wiki/Boolean_algebra

Decision Coverage

- 프로그램의 모든 시작점과 끝점을 적어도 한 번 이상 방문하고,
- 프로그램 내 모든 Decision Point는 적어도 한 번 이상 모든 가능한 결과값을 가져야 함

```
if (A AND B) {
    // true
} else {
    // false
}
```

Condition		Decision Point
A	B	
True	True	True
True	False	False

https://en.wikipedia.org/wiki/Modified_condition/decision_coverage

Condition Coverage

- 프로그램의 Decision Point 내 모든 Condition은 한 번 이상 모든 가능한 결과값을 가져야 함

```
if (A AND B) {
    // true
} else {
    // false
}
```

Condition		Decision Point
A	B	
True	False	False
False	True	False

https://en.wikipedia.org/wiki/Modified_condition/decision_coverage

Condition / Decision Coverage

- 프로그램의 모든 시작점과 끝점을 적어도 한 번 이상 방문하고,
- 프로그램 내 모든 Decision Point는 적어도 한 번 이상 모든 가능한 결과값을 가져야 하고,
- 프로그램의 Decision Point 내 모든 Condition은 한 번 이상 모든 가능한 결과값을 가져야 함

```
if (A AND B) {
    // true
} else {
    // false
}
```

Condition		Decision Point
A	B	
True	True	True
False	False	False

https://en.wikipedia.org/wiki/Modified_condition/decision_coverage

Modified Condition / Decision Coverage

- 프로그램의 모든 시작점과 끝점을 적어도 한 번 이상 방문하고,
- 프로그램 내 모든 Decision Point는 적어도 한 번 이상 모든 가능한 결과값을 가져야 하고,
- 프로그램의 Decision Point 내 모든 Condition은 한 번 이상 모든 가능한 결과값을 가져야 하고,
- 각 Condition이 다른 Condition에 영향을 받지 않고 전체 Decision Point의 결과에 독립적으로 영향을 줘야 한다.

```
if (A AND B) {
    // true
} else {
    // false
}
```

Condition		Decision Point
A	B	
True	True	True
True	False	False
False	True	False

https://en.wikipedia.org/wiki/Modified_condition/decision_coverage

Multiple Condition Coverage

- **Decision Point 내의 모든 Condition들의 조합을 테스트해야 한다.**

```
if (A AND B) {
    // true
} else {
    // false
}
```

Condition		Decision Point
A	B	
True	True	True
True	False	False
False	True	False
False	False	False

https://en.wikipedia.org/wiki/Code_coverage

Multiple Condition Coverage

```
if ((A OR B) AND C) {
    // true
} else {
    // false
}
```

Condition			Decision Point
A	B	C	
True	True	True	True
True	True	False	False
True	False	True	True
True	False	False	False
False	True	True	True
False	True	False	False
False	False	True	False
False	False	False	False

https://en.wikipedia.org/wiki/Code_coverage

Google Test Assertions

No	Fatal Assertion	Non Fatal Assertion	Verified
1	`ASSERT_TRUE(condition);`	`EXPECT_TRUE(condition);`	condition is true
2	`ASSERT_FALSE(condition);`	`EXPECT_FALSE(condition);`	condition is false
3	`ASSERT_EQ(val1, val2);`	`EXPECT_EQ(val1, val2);`	val1 == val2
4	`ASSERT_NE(val1, val2);`	`EXPECT_NE(val1, val2);`	val1 != val2
5	`ASSERT_LT(val1, val2);`	`EXPECT_LT(val1, val2);`	val1 < val2
6	`ASSERT_LE(val1, val2);`	`EXPECT_LE(val1, val2);`	val1 <= val2
7	`ASSERT_GT(val1, val2);`	`EXPECT_GT(val1, val2);`	val1 > val2
8	`ASSERT_GE(val1, val2);`	`EXPECT_GE(val1, val2);`	val1 >= val2
9	`ASSERT_FLOAT_EQ(val1, val2);`	`EXPECT_FLOAT_EQ(val1, val2);`	the two float values are almost equal
10	`ASSERT_DOUBLE_EQ(val1, val2);`	`EXPECT_DOUBLE_EQ(val1, val2);`	the two double values are almost equal
11	`ASSERT_NEAR(val1, val2, abs_error);`	`EXPECT_NEAR(val1, val2, abs_error);`	the difference between val1 and val2 doesn't exceed the given absolute error

https://github.com/google/googletest/blob/master/googletest/docs/Primer.md

Buildering without Safety Equipment

https://www.youtube.com/watch?v=Hy2mW29bVx8

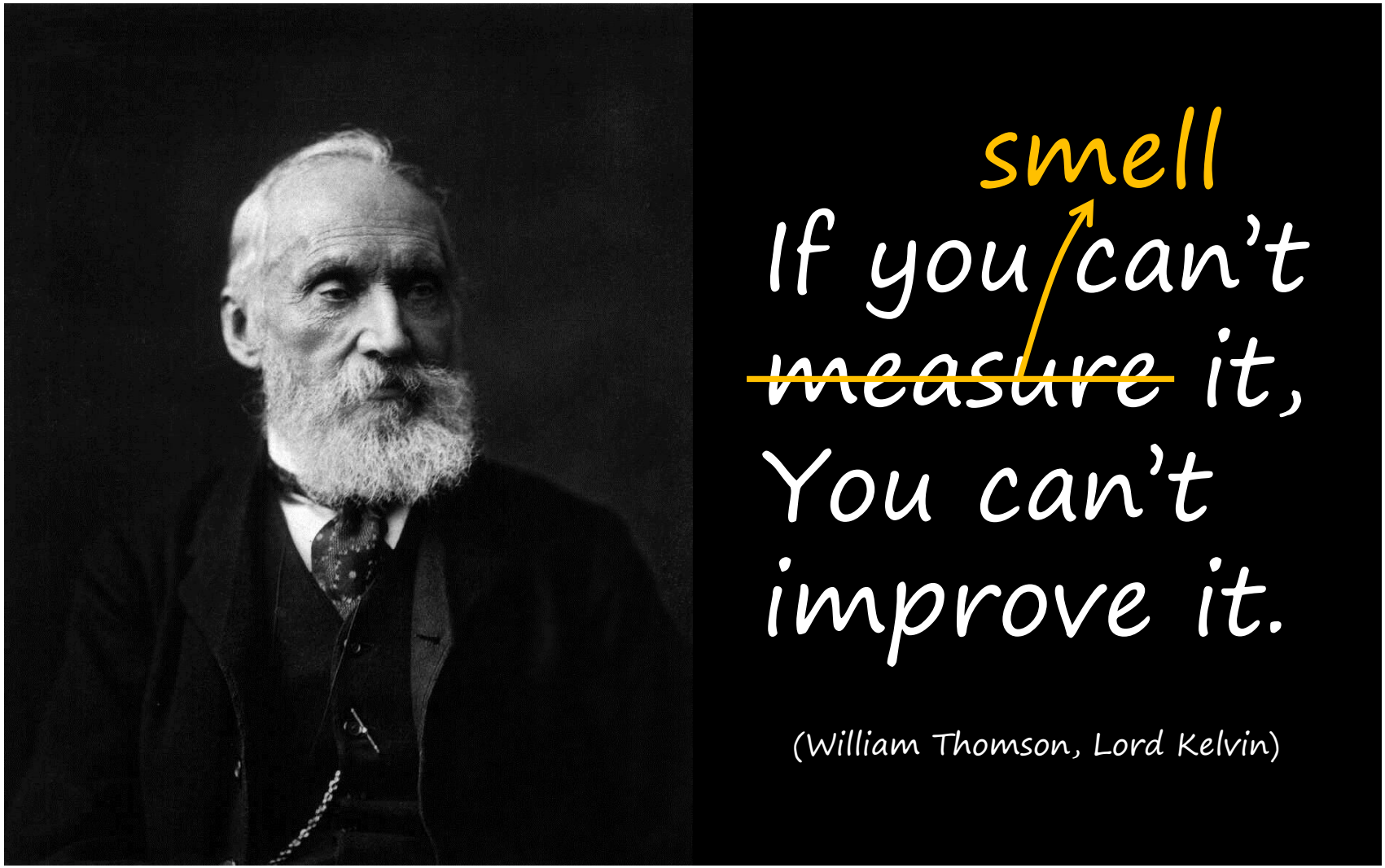
Bad Smells in Code

- 학습목표 1
- 학습목표 2

(1) If you can’t …

(2) Bad Smells in Code

If you can't ...



https://en.wikipedia.org/wiki/William_Thomson,_1st_Baron_Kelvin

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
1	Duplicated Code	중복 코드 (100)	동일한 코드가 두 군데 이상 있다.	Extract Method (142), Extract Class (187), Pull Up Method (381), Form Template Method (407)	코드	★★★★★
2	Long Method	장황한 메소드 (101)	메소드가 길다.	Extract Method (142), Replace Temp with Query (153), Replace Method with Method Object (170), Decompose Conditional (286)	코드	★★★★★
3	Large Class	방대한 클래스 (102)	클래스에 인스턴스 변수가 많고 코드가 길다.	Extract Class (187), Extract Subclass (390), Extract Interface (403), Replace Data Value with Object (215)	설계	★★★★★
4	Long Parameter List	과다한 매개변수 (103)	파라메터가 많고 자주 변경된다.	Replace Parameter with Method (347), Introduce Parameter Object (351), Preserve Whole Object (342)	코드	★
5	Divergent Change	수정의 산발 (104)	클래스가 다양한 원인으로 수정된다.	Extract Class (187)	설계	★★★
6	Shotgun Surgery	기능의 산재 (105)	수정할 때마다 수많은 자잘한 부분을 고쳐야 한다.	Move Method (178), Move Field (183), Inline Class (192)	설계	★★★★
7	Feature Envy	잘못된 소속 (105)	메소드가 자신이 속하지 않은 클래스에 더 많이 접근한다.	Move Method (178), Move Field (183), Extract Method (142)	설계	★★★
8	Data Clumps	데이터 뭉치 (106)	뭉쳐 다녀야 할 데이터가 하나의 객체로 정리되어 있지 않다.	Extract Class (187), Introduce Parameter Object (351), Preserve Whole Object (342)	설계	★★★
9	Primitive Obsession	강박적 기본 타입 사용 (107)	적절한 타입을 사용하지 않고 기본 자료형만을 사용한다.	Replace Data Value with Object (215), Extract Class (187), Introduce Parameter Object (351), Replace Array with Object (227), Replace Type Code with Class (263), Replace Type Code with Subclasses (269), Replace Type Code with State/Strategy (273)	코드	★★★
10	Switch Statements	switch 문 (108)	switch 문에 새 코드를 추가하려면 여기저기 존재하는 switch 문을 전부 찾아서 수정해야 한다.	Replace Conditional with Polymorphism (305), Replace Type Code with Subclasses (269), Replace Type Code with State/Strategy (273), Replace Parameter with Explicit Methods (338), Introduce Null Object (310)	설계	★★★★
11	Parallel Inheritance Hierarchies	평행 상속 계층 (109)	하위클래스를 만들 때마다 매번 다른 클래스의 하위클래스도 만들어야 한다.	Move Method (178), Move Field (183)	설계	★★★★
12	Lazy Class	직무유기 클래스 (109)	리팩토링 이후 기능이 축소된 클래스나 쓸모없어진 클래스가 있다.	Inline Class (192), Collapse Hierarchy (406)	설계	★★
13	Speculative Generality	막연한 범용 코드 (110)	아직은 필요 없는 기능이 구현되어 있다.	Collapse Hierarchy (406), Inline Class (192), Remove Parameter (329), Rename Method (325)	코드	★★
14	Temporary Field	임시 필드 (110)	객체 안에 인스턴스 변수가 특정 상황에서만 할당된다.	Extract Class (187), Introduce Null Object (310)	설계	★★★★
15	Message Chains	메시지 체인 (111)	어떤 객체를 사용하기 위해 여러 개의 객체에 연쇄적으로 객체를 요청해야 한다.	Hide Delegate (195)	설계	★
16	Middle Man	과잉 중개 메소드 (112)	많은 메소드가 기능을 다른 클래스에 위임하고 있다.	Remove Middle Man (198), Inline Method (150), Replace Delegation with Inheritance (419)	설계	★★
17	Inappropriate Intimacy	지나친 관여 (112)	두 개의 클래스가 서로의 private 변수를 많이 사용한다.	Move Method (178), Move Field (183), Change Bidirectional Association to Unidirectional (244), Replace Inheritance with Delegation (416), Hide Delegate (195)	설계	★★★
18	Alternative Classes with Different Interfaces	인터페이스가 다른 대용 클래스 (113)	기능은 같은데 시그니처가 다른 메소드가 있다.	Rename Method (325), Move Method (178)	코드	★★
19	Incomplete Library Class	미흡한 라이브러리 클래스 (113)	사용한 라이브러리가 적절하지 않다.	Introduce Foreign Method (201), Introduce Local Extension (203)	코드	★
20	Data Class	데이터 클래스 (114)	데이터 보관만을 담당하는 클래스가 있다.	Move Method (178), Encapsulate Field (250), Encapsulate Collection (252)	설계	★★
21	Refused Bequest	방치된 상속물 (114)	상속받은 메소드나 데이터가 하위클래스에서 더 이상 사용되지 않는다.	Replace Inheritance with Delegation (416)	구조	★
22	Comments	불필요한 주석 (115)	Bad Smell을 변명하는 주석이 있다.	Extract Method (142), Introduce Assertion (319)	코드	★

Duplicated Code

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
1	Duplicated Code	중복 코드 (100)	동일한 코드가 두 군데 이상 있다.	Extract Method (142), Extract Class (187), Pull Up Method (381), Form Template Method (407)	코드	★★★★

- 악취 퍼레이드에서 일등
- 한 클래스의 서로 다른 두 메소드 안에 같은 코드가 있는 경우
  - Extract Method 후 뽑아낸 메소드를 그 두 곳에서 호출
- 동일한 부모클래스를 갖는 두 서브클래스에 같은 코드가 있는 경우
  1. 양쪽 클래스에서 Extract Method를 사용
  2. Pull Up Method를 사용
- 코드가 비슷하기는 하지만 같지는 않은 경우
  1. 비슷한 부분은 Extract Method 사용
  2. Form Template Method 사용
- 메소드들이 같은 작업을 하지만 다른 알고리즘을 사용하는 경우
  1. 두 알고리즘 중 더 명확한 것을 선택
  2. Substitute Algorithm 사용
- 서로 관계 없는 두 클래스에서 중복 코드가 있는 경우
  - 한쪽 클래스에서 Extract Class 사용 후 양쪽에서 이 새로운 클래스를 사용
  - 중복 코드를 빼서 메소드로 만든 후 두 클래스 중 하나에 넣고 다른 클래스에서 사용, 종속성 발생 주의
  - 코드를 빼서 만든 메소드를 제 3의 클래스에 넣고 그걸 두 클래스에서 사용

Long Method 1

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
2	Long Method	장황한 메소드 (101)	메소드가 길다.	Extract Method (142), Replace Temp with Query (153), Replace Method with Method Object (170), Decompose Conditional (286)	코드	★★★★★

- 최적의 상태로 가장 오래 살아 남는 객체 프로그램은 메소드의 길이가 짧음
- 긴 프로시저는 이해하기 어려움
  - 오래된 언어에서는 서브루틴을 호출할 때 오버헤드가 이슈
  - 현대의 객체지향 언어는 호출의 오버헤드를 상당 부분 제거
  - 서브루틴이 무엇을 하는지 확인하기 위한 가독성의 오버헤드는 여전히 존재
- 짧은 메소드는 가독성 측면에서 유리
  - 개수가 많아지면 메소드 이름을 잘 지어야 함
  - 메소드의 이름을 잘 지으면 내용을 들여다 볼 필요 없이 이해 가능
  - 동의어 사전 활용 (http://thesaurus.com)
- 메소드를 분해는 훨씬 더 공격적이어야 함
  - 주석을 달아야 할 필요를 느낄 때마다 메소드 작성
  - 메소드 이름은 주석에 적을 내용으로 선정 (how보다는 what으로)
  - 한 줄의 코드일 지라도 주석을 달아야 한다면 메소드 작성 필요
- Long Method의 길이는 라인수가 아니라
  - 일과 일을 처리하는 방법 사이의 의미적 거리(semantic distance)를 의미
- 메소드의 길이를 줄이려면
  - Extract Method

Long Method 2

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
2	Long Method	장황한 메소드 (101)	메소드가 길다.	Extract Method (142), Replace Temp with Query (153), Replace Method with Method Object (170), Decompose Conditional (286)	코드	★★★★★

- 메소드에 파라미터와 임시변수가 많아서 추출이 어려운 경우
  - 임시변수 제거 : Replace Temp with Query
  - 긴 파라미터 리스트 단순화 : Introduce Parameter Object, Preserve Whole Object
- 임시변수와 파라미터가 여전히 많은 경우
  - Replace Method with Method Object
- 뽑아낼 코드 덩어리의 식별 방법
  - 코드끼리의 의미적 거리(semantic distance) 분석
  - 주석이 붙은 코드 블록
  - 조건문과 루프
    - Decompose Conditional

Large Class

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
3	Large Class	방대한 클래스 (102)	클래스에 인스턴스 변수가 많거나 코드가 길다.	Extract Class (187), Extract Subclass (390), Extract Interface (403), Replace Data Value with Object (215)	설계	★★★★

- 클래스 하나가 너무 많은 일을 하려 할 때 보통 지나치게 많은 인스턴스 변수 발생
  - 지나치게 많은 인스턴스 변수가 있는 경우 중복 코드가 존재할 확률도 높음
- 많은 변수를 묶으려면
  - Extract Class
  - 서로에게 의미있는 변수를 골라서 묶음
  - 클래스 안의 일부 변수에 공통된 접두사나 접미사가 있으면 묶음
  - 새로 만들 클래스가 서브클래스의 의미가 있으면 Extract Subclass
- 클래스가 모든 인스턴스 변수를 사용하지 않는 경우
  - Extract Class와 Extract Subclass를 여러 번 사용
- 지나치게 많은 코드를 제거하려면
  - 중복 코드 제거
  - Extract Class와 Extract Subclass
  - 클라이언트가 클래스를 어떻게 쓰게 할 것인지를 결정하고 각각의 사용 방법에 대해 Extract Interface 사용
- GUI 클래스인 경우
  - 데이터와 동작을 별도의 객체로 이동
  - Duplicate Observed Data

Long Parameter List

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
4	Long Parameter List	과다한 매개변수 (103)	파라메터가 많고 자주 변경된다.	Replace Parameter with Method (347), Introduce Parameter Object (351), Preserve Whole Object (342)	코드	★

- 프로그래밍 초창기에는 글로벌 변수보다 파라미터 전달이 좋은 방법이었음
  - OOP에서는 객체에서 직접 데이터를 가져올 수 있음
  - 파라미터는 필요한 것만 넘기고
  - 많은 정보는 메소드를 포함하고 있는 클래스에서 구함
- 긴 파라미터 리스트는
  - 이해하기도 어렵고
  - 일관성이 없거나 사용하기 어렵고
  - 다른 데이터가 필요할 때마다 계속 고쳐야 함
  - 파라미터 리스트는 짧은 게 좋음
- 이미 알고 있는 객체에 요청하여 파라미터의 데이터를 얻을 수 있는 경우
  - Replace Parameter with Method
- 한 객체로부터 주워 모은 데이터 뭉치를 그 객체 자체로 바꾸는 경우
  - Preserve Whole Object
- 객체와 관계 없는 여러 개의 데이터 아이템이 있는 경우
  - Introduce Parameter Object
  - 호출하는 객체와 큰 객체 사이의 종속성 발생 주의
    - 이럴 경우 데이터를 하나씩 풀어서 파라미터로 전달

Divergent Change

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
5	Divergent Change	수정의 산발 (104)	클래스가 다양한 원인으로 수정된다.	Extract Class (187)	설계	★★

- 소프트웨어를 변경할 때 명확히 한 곳을 집어 변경할 수 있기를 바란다.
  - 이렇게 할 수 없을 때 밀접한 관계가 있는 두 가지 냄새 중에 하나
    - Divergent Change, Shotgun Surgery
- 여러 종류의 변경 때문에 하나의 클래스가 시달리는 경우
- 특정 원인에 대해 변화해야 하는 것을 모두 찾은 다음
  - Extract Class를 사용하여 하나로 묶음

Shotgun Surgery

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
6	Shotgun Surgery	기능의 산재 (105)	수정할 때마다 수많은 자잘한 부분을 고쳐야 한다.	Move Method (178), Move Field (183), Inline Class (192)	설계	★★★

- 소프트웨어를 변경할 때 명확히 한 곳을 집어 변경할 수 있어야 됨.
  - 이렇게 할 수 없을 때 밀접한 관계가 있는 두 가지 냄새 중에 하나
    - Divergent Change, Shotgun Surgery
- 하나를 변경했을 때 많은 클래스를 고쳐야 하는 경우
  - Divergent Change과 비슷하지만 정 반대
  - 변경해야 할 것이 여기저기 널려있다면,
    - 찾기도 어렵고,
    - 변경해야 할 중요한 사항을 빼먹기도 쉬움
- Move Method와 Move Field를 사용하여
  - 변경해야 할 부분을 모두 하나의 클래스로 묶음
  - 만약 기존의 클래스 중에서 메소드나 필드가 옮겨갈 적절한 클래스가 없는 경우
    - 새로 하나를 만듬
- Inline Class를 사용하여 모든 동작을 하나로 모음
  - Divergent Change 냄새가 날 수 있음

Feature Envy

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
7	Feature Envy	잘못된 소속 (105)	메소드가 자신이 속하지 않은 클래스에 더 많이 접근한다.	Move Method (178), Move Field (183), Extract Method (142)	설계	★

- OOP의 핵심은 데이터와 데이터를 사용하는 프로세스를 하나로 묶는 기술
- 메소드가 자신이 속한 클래스보다 다른 클래스에 관심을 더 많이 가지고 있는 경우
  - 가장 흔한 욕심은 데이터에 대한 욕심
- 그 메소드는 분명히 다른 곳에 있고 싶은 것이
  - Move Method로 이동
- 메소드 내부의 특정 부분만 욕심이 있는 경우
  - 욕심이 많은 부분에 대해 Extract Method를 사용한 다음 Move Method 사용
- 한 메소드가 여러 개의 클래스 기능을 사용하는 경우
  - 어떤 클래스에 있는 데이터를 가장 많이 사용하는가를 확인
  - Extract Method로 각각 다른 장소로 옮겨질 메소드로 쪼개져 있으면 좋음

Data Clumps

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
8	Data Clumps	데이터 뭉치 (106)	뭉쳐 다녀야 할 데이터가 하나의 객체로 정리되어 있지 않다.	Extract Class (187), Introduce Parameter Object (351), Preserve Whole Object (342)	설계	★

- 데이터 아이템은 아이들과 같아서 몰려다니기를 좋아함
- 함께 몰려다니는 데이터 무리는 그들 자신의 객체로 만들어져야 함
  - 첫 번째 단계는 필드로 나타나는 덩어리들이 있는 곳을 찾는 것
    - 이 덩어리를 객체로 바꾸기 위해 Extract Class 사용
  - 메소드 시그니처에서 파라미터 리스트 단순화
    - Introduce Parameter Object, Preserve Whole Object 사용
    - 메소드 호출이 간단해짐
- 데이터 값 중 하나를 삭제했을 때 다른 것들이 의미가 없다면
  - 객체가 새로 만들어져야 함
- 새로운 객체는 Feature Envy 냄새가 날 수 있지만
  - 이것은 새로 만든 클래스로 기능을 옮길 것을 암시
  - 머지않아 이 클래스는 프로그램의 생산적인 구성원이 됨

Primitive Obsession

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
9	Primitive Obsession	강박적 기본 타입 사용 (107)	적절한 타입을 사용하지 않고 기본 자료형만을 사용한다.	Replace Data Value with Object (215), Extract Class (187), Introduce Parameter Object (351), Replace Array with Object (227), Replace Type Code with Class (263), Replace Type Code with Subclasses (269), Replace Type Code with State/Strategy (273)	코드	★★

- 프로그래밍 환경의 두 종류의 데이터 유형
  - 레코드 타입 : 데이터를 구조화한 의미 있는 그룹
  - 기본 타입 : 벽돌
- OOP는 레코드 타입도 기본 타입처럼 쉽게 만들어서 사용할 수 있음
- 기본 타입 대신에 객체로 사용하고 싶은 경우
  - Replace Data Value with Object 사용
- 데이터 값이 타입 코드이고 값이 동작에 영향을 미치지 않는 경우
  - Replace Type Code with Class 사용
- 타입 코드에 의존하는 조건문이 있는 경우
  - Replace Type Code with Subclass 사용
  - 또는 Replace Type Code with State/Strategy 사용
- 항상 몰려다녀야 할 필드 그룹이 있는 경우
  - Extract Class 사용
- 파라미터 리스트에서 이런 기본 타입을 보면
  - Introduce Parameter Object 사용
- 배열을 쪼개서 쓰고 있는 자신을 발견하면
  - Replace Array with Object 사용

Switch Statement

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
10	Switch Statements	switch 문 (108)	switch 문에 새 코드를 추가하려면 여기 저기 존재하는 switch 문을 전부 찾아서 수정해야 한다.	Replace Conditional with Polymorphism (305), Replace Type Code with Subclasses (269), Replace Type Code with State/Strategy (273), Replace Parameter with Explicit Methods (338), Introduce Null Object (310)	설계	★★★

- OOP의 가장 명확한 특징은 switch문이 비교적 적게 쓰인다는 것
- 동일한 switch문이 프로그램 여기저기에 흩어져 있어서 한 switch문에 코드를 추가하려면 중복된 모든 switch문을 찾아 바꿔줘야 하는 경우
  - OOP의 개념 중 다형성(polymorphism)으로 이런 문제를 해결
- Switch문에서 타입 코드를 사용할 경우
  - Extract Method를 사용하여 switch문을 뽑아내고 Move Method를 사용하여 다형성이 필요한 클래스로 옮김
  - Replace Type Code with Subclasses나 Replace Type Code with State/Strategy 사용
  - 상속 구조를 결정했으면 Replace Conditional with Polymorphism 사용
- 하나의 메소드에만 영향을 미치는 경우
  - 다형성 적용은 과함, 적용 안함
  - Replace Parameter with Explicit Methods 사용
- 조건 중 null이 있는 경우가 있는 경우
  - Introduce Null Object 사용

Parallel Inheritance Hierarchies

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
11	Parallel Inheritance Hierarchies	평행 상속 계층 (109)	하위클래스를 만들 때마다 매번 다른 클래스의 하위클래스도 만들어야 한다.	Move Method (178), Move Field (183)	설계	★★★

- **Shotgun Surgery의 특별한 경우**
- **한 클래스의 서브클래스를 만들면 다른 곳에서도 모두 서브클래스를 만들어주는 경우**
- **한쪽 상속 구조에서 클래스 이름의 접두어가 다른 쪽 상속 구조의 클래스 이름의 접두어와 같은 경우 의심**
- **중복을 제거하는 일반적인 전략은**
  - **한쪽 상속 구조의 인스턴스가 다른 쪽 구조의 인스턴스를 참조하게 만듬**
  - **Move Method와 Move Field를 사용**

Lazy Class

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
12	Lazy Class	직무유기 클래스 (109)	리팩토링 이후 기능이 축소된 클래스나 쓸모없어진 클래스가 있다.	Inline Class (192), Collapse Hierarchy (406)	설계	★★★

- 클래스를 생성할 때마다 그것을 유지하고 이해하기 위한 비용 발생
  - 이 비용을 감당할 만큼 충분하지 않은 일을 하는 클래스는 삭제
- 처음에는 존재 가치가 있었지만 리팩토링의 결과로 크기가 줄어들었을 수 있음
- 미리 계획된 변경사항을 반영하기 위해 추가했는데 쓰이지 않는 경우
- 별로 하는 일도 없는 클래스의 서브클래스가 있는 경우
  - Collapse Hierarchy 사용
- 거의 필요 없는 클래스에 대해서는
  - Inline Class 사용

Speculative Generality

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
13	Speculative Generality	막연한 범용 코드 (110)	아직은 필요 없는 기능이 구현되어 있다.	Collapse Hierarchy (406), Inline Class (192), Remove Parameter (329), Rename Method (325)	코드	★

- 언젠가 이런 종류의 일을 처리하기 위한 기능이 필요하다고 생각되어 구현된 기능
  - 코드는 더욱 더 이해하기 어려워지고 유지보수가 힘들어짐
  - 사용되지 않는 것은 방해가 될 뿐이므로 제거
- 별로 하는 일이 없는 추상 클래스가 있는 경우
  - Collapse Hierarchy 사용
- 불필요한 위임은
  - Inline Class로 제거
- 메소드에 사용하지 않는 파라미터가 있는 경우
  - Remove Parameter 사용
- 메소드의 이름이 이상하고 추상적인 경우
  - Rename Method 사용
- 어떤 메소드나 클래스가 테스트 케이스에서만 사용되는 경우
  - 해당 메소드, 클래스, 이들을 이용하는 테스트 케이스 삭제
  - 테스트 케이스의 helper는 예외

Temporary Field

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
14	Temporary Field	임시 필드 (110)	객체 안에 인스턴스 변수가 특정 상황에서만 할당된다.	Extract Class (187), Introduce Null Object (310)	설계	★★

- 어떤 객체 안의 인스턴스 변수가 특정 상황에서만 세팅되는 경우
  - 이런 코드는 어려운데, 왜냐하면 보통의 객체의 모든 변수가 값을 가지고 있을 거라고 기대하기 때문
  - 사용되지 않는 것처럼 보이는 변수가 왜 있는지를 이해하는 것은 매우 짜증나는 일
- 불쌍한 고아 변수들을 위한 집을 만들기 위해 Extract Class 사용
  - 그 변수를 사용하는 모든 코드를 새로 만든 클래스로 이동
- 변수의 값이 유효하지 않은 경우에 대한 대체 컴포넌트를 만들기 위해
  - Introduce Null Object를 사용하여 조건문이 포함된 코드 제거
- 임시 필드는 복잡한 알고리즘이 여러 변수를 필요로 할 때 흔히 발생
  - 필요한 변수와 메소드를 묶어 Extract Class 사용
  - 새로운 객체는 메소드 객체가 됨

Message Chains

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
15	Message Chains	메시지 체인 (111)	어떤 객체를 사용하기 위해 여러 개의 객체에 연쇄적으로 객체를 요청해야 한다.	Hide Delegate (195)	설계	★

- 클라이언트가 어떤 객체를 얻기 위해 다른 객체에게 물어보고
  - 다른 객체는 다시 또 다른 객체에게 물어보고
    - 그 객체는 다시 또 다른 객체에게 물어보고
      - ...
      - 클라이언트가 클래스 구조와 결합되어 있는 구조
      - 중간의 어떤 관계가 변하면 클라이언트 코드도 변경되어야 함
- 이 경우 Hide Deligate 사용
  - 체인의 여러 지점에서 적용
  - 원칙적으로 체인 내의 모든 객체에 적용
    - 중간의 모든 객체를 Middle Man으로 만드는 결과 초래
- 때로는 결과적으로 어떤 객체가 사용되는지를 보는 것이 나을 수 있음
- 그것을 사용하는 코드의 조각에 Extract Method를 사용할 수 있는지 보고
  - Move Method로 그것을 체인 아래로 밀어넣을 수 있는 지 확인
- 체인 중에 한 객체의 여러 클라이언트가 나머지 부분의 체인을 돌아다니고 싶어하는 경우
  - 이를 위한 메소드 추가

Middle Man

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
16	Middle Man	과잉 중개 메소드 (112)	많은 메소드가 기능을 다른 클래스에 위임하고 있다.	Remove Middle Man (198), Inline Method (150), Replace Delegation with Inheritance (419)	설계	★

- OOP의 주요 특징 중의 하나는 캡슐화(encapsulation)
- 캡슐화는 보통 위임(delegration)과 함께 사용
- 위임이 지나쳐서 메소드의 태반이 다른 클래스에 위임을 하고 있다는 경우
  - Remove Middle Man을 사용하여 그 객체에 직접 접근
- 몇몇 메소드가 많은 일을 하지 않는 경우
  - Inline Method를 사용하여 호출하는 곳에 코드를 삽입
- 추가 동작이 있는 경우
  - Replace Delegation with Inheritance를 사용하여 Middle Man을 실제 객체의 서브클래스로 바꿈
  - 위임을 하나하나 따라갈 필요 없이 기능을 확장할 수 있음

Inappropriate Intimacy

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
17	Inappropriate Intimacy	지나친 관여 (112)	두 개의 클래스가 서로의 private 변수를 많이 사용한다.	Move Method (178), Move Field (183), Change Bidirectional Association to Unidirectional (244), Replace Inheritance with Delegation (416), Hide Delegate (195)	설계	★★

- 지나치게 친밀한 클래스는 옛날에 연인을 갈라 놓은 것처럼 서로 떼어 놓아야 함
  - Move Method와 Move Field를 사용하여 조각으로 나누고 친밀함을 줄임
  - Change Bidirectional Association to Unidirectional이 적용 가능한지 확인
- 이들 클래스에 공통 관심사가 있는 경우
  - Extract Class를 사용하여 공통된 부분을 안전한 곳으로 빼내서 별도의 클래스로 작성
  - 또는 Hide Deligate를 사용하여 다른 클래스가 중개하도록 함
- 상속은 과도한 친밀을 유도
  - 서브클래스는 항상 그 부모클래스가 알려주고 싶은 것보다 많은 것을 알려고 함
  - 만약 출가(서브클래스를 부모클래스에서 분리) 할 때라면
    - Replace Inheritance with Delegation 사용

Alternative Classes with Different Interfaces

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
18	Alternative Classes with Different Interfaces	인터페이스가 다른 대용 클래스 (113)	기능은 같은데 시그니처가 다른 메소드가 있다.	Rename Method (325), Move Method (178)	코드	★★

- 같은 작업을 하지만 다른 시그니처를 가지는 메소드에 대해서는 Rename Method를 사용
- 이것으로 부족하고 여전히 충분한 작업을 하지 않는 경우
  - 프로토콜이 같아질 때까지 Move Method를 이용하여 동작을 이동
- 너무 많은 코드를 옮겨야 할 경우
  - 목적을 이루기 위해 Extract Subclass 사용

Incomplete Library Class

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
19	Incomplete Library Class	미흡한 라이브러리 클래스 (113)	사용한 라이브러리가 적절하지 않다.	Introduce Foreign Method (201), Introduce Local Extension (203)	코드	★

- 재사용은 종종 객체의 목적으로 홍보됨
  - 과대평가되었다고 생각하지만
  - 그러나, 프로그래밍 기술은 라이브러리 클래스를 기반으로 함
- 클래스 라이브러리를 만드는 사람이라고 모든 것을 다 알 수 없음
  - 라이브러리가 종종 내가 원하는 형태가 아니더라도
  - 내가 원하는 것을 하기 위해 라이브러리 클래스를 수정하는 것은 불가능
- 라이브러리 클래스가 가지고 있었으면 하는 메소드가 몇 개 있는 경우
  - Introduce Foreign Method 사용
- 별도의 동작이 잔뜩 있다면
  - Introduce Local Extension 사용

Data Class

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
20	Data Class	데이터 클래스 (114)	데이터 보관만을 담당하는 클래스가 있다.	Move Method (178), Encapsulate Field (250), Encapsulate Collection (252)	설계	★★

- 필드와 각 필드에 대한 get/set 메소드만 가지고 다른 것은 아무것도 없는 클래스
  - 멍청하게 데이터만 저장하고 거의 대부분 다른 클래스에 의해 조작
- 초기 단계에 이런 클래스는 public 필드를 가짐
  - Encapsulate Field 사용
- 컬렉션 필드를 가지고 있는 경우
  - 캡슐화가 되어 있는지 확인
  - 적절히 캡슐화가 되어 있지 않는 경우
    - Encapsulate Collection 사용
- 값이 변경되면 안되는 필드가 있는 경우
  - Remove Setting Method 사용
- get/set 메소드가 다른 클래스에서 사용되는지 찾아보고
  - 그 동작을 데이터 클래스로 옮기기 위해 Move Method 사용
- 메소드 전체를 옮길 수 없을 때는
  - Extract Method를 사용해서 옮길 수 있는 메소드를 작성
  - 잠시 후에 get/set 메소드에 대해 Hide Method 사용
- 데이터 클래스는 아이와 같음
  - 어른 클래스로 참여하기 위해서는 약간의 책임을 가질 필요 있음

Refused Bequest

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
21	Refused Bequest	방치된 상속물 (114)	상속받은 메소드나 데이터가 하위클래스에서 더 이상 사용되지 않는다.	Replace Inheritance with Delegation (416)	구조	★

- 서브클래스는 메소드와 데이터를 그 부모클래스로부터 상속
- 만약 서브클래스가 그들에게 주어진 것을 원치 않는 경우
  - 전통적으로 클래스 상속 구조가 잘못되었다는 것을 뜻함
  - 새로운 형제클래스를 만들고
  - Push Down Method와 Push Down Field를 사용해서 사용되지 않는 메소드를 모두 형제클래스로 옮김
  - 이런 방법으로 부모 클래스는 공통 부분만 가질 수 있음
  - 부모클래스는 추상클래스가 될 수 있음
- 요즘은 동작 일부를 재사용하기 위해 서브클래싱을 하는 방법을 자주 사용
  - 냄새는 나지만 강한 냄새는 아님
- 서브클래스가 동작은 재사용하지만 부모클래스의 인터페이스를 지원하지 않는 경우
  - 인터페이스를 거부하는 것은 심각한 문제
  - 클래스를 손보는 것 보다는 Replace Inheritance with Delegation 사용

Comments

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
22	Comments	불필요한 주석 (115)	Bad Smell을 변명하는 주석이 있다.	Extract Method (142), Introduce Assertion (319)	코드	★

- 주석을 쓰면 안된다는 말은 아님
  - 주석 자체는 달콤한 향기
  - 주석이 잔뜩 붙어 있는 코드를 보면 코드가 서투르기 때문에 주석이 달려있는 경우가 많음
- 주석 때문에 우리는 썩은 냄새가 진동하는 나쁜 코드를 작성하게 됨
  - 첫째로 해야할 것은 리팩토링으로 나쁜 냄새를 제거
  - 리팩토링을 마친 후에야 그 주석들이 불필요한 것이었다는 사실을 알게 됨
- 코드 블록이 무슨 작업을 설명하기 위해 주석이 필요한 경우
  - Extract Method 사용
- 메소드가 이미 추출되어 있는데도 여전히 코드가 하는 일에 대한 주석이 필요한 경우
  - Rename Method 사용
- 시스템의 필요한 상태에 대한 어떤 규칙같은 것을 설명할 필요가 있는 경우
  - Introduce Assertion 사용

Refactoring Techniques

(1) 리팩토링 정의

(2) 리팩토링 시점

(3) 리팩토링 순서

리팩토링 정의

- 겉으로 드러나는 동작의 변경 없이 소프트웨어의 내부 구조를 이해하기 쉽고 관리 비용이 적게 들도록 변경하는 작업

Refactoring p.75

리팩토링 시기

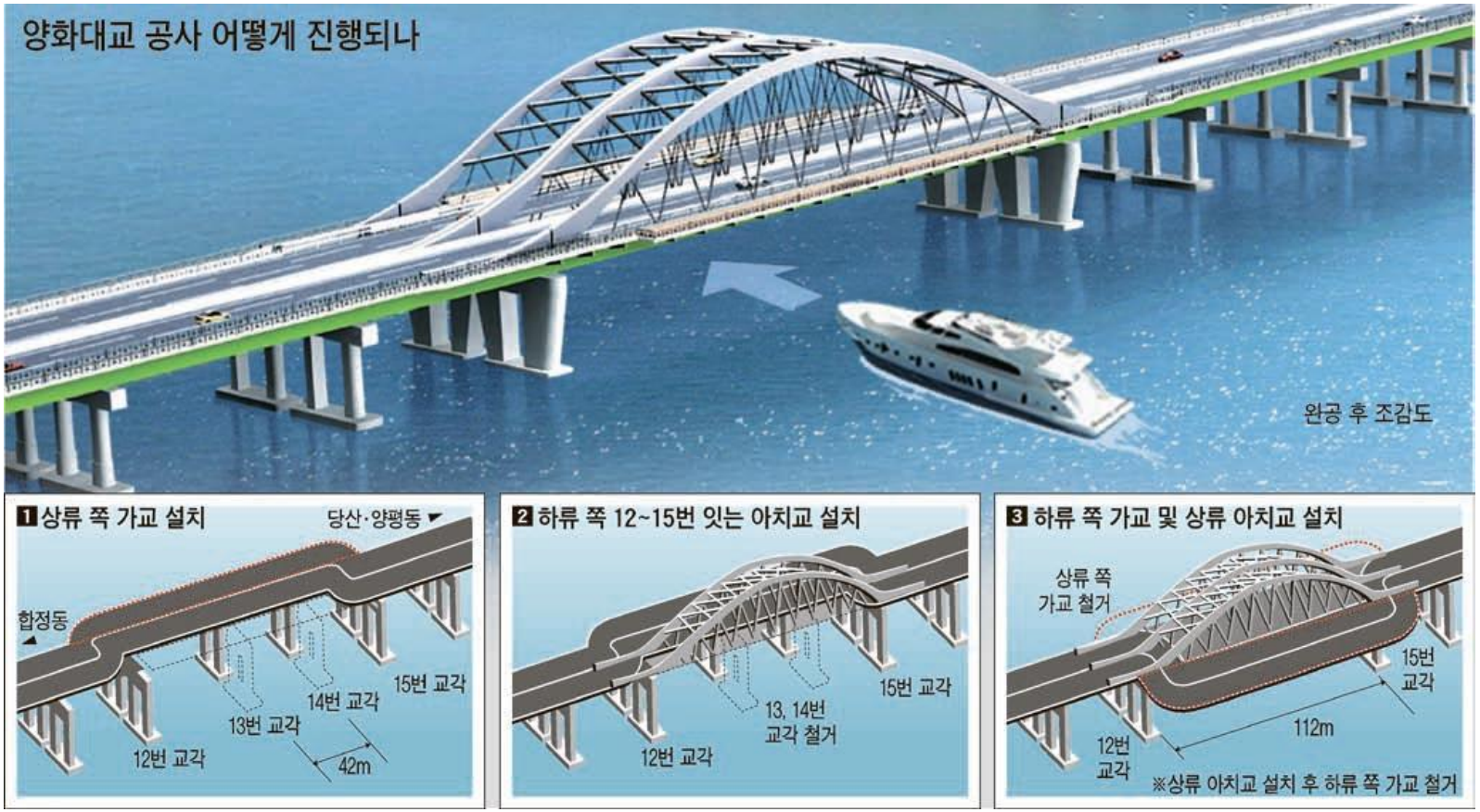
- 삼진 규칙
  - 같은 냄새를 세 번 맡았을 때
- 기능을 추가할 때
  - 수정해야 할 코드에 이해를 돕기 위해
  - 기능 추가가 쉬운 디자인으로 만들기 위해
- 버그를 수정할 때
  - 버그가 있었다는 것을 몰랐을 정도로 코드가 명확하지 않음
- 코드 리뷰를 할 때

Refactoring p.80

리팩토링 순서

- 코드가 어떤 Bad Smells에 해당하는지 확인
- 가능한 Common Refactorings 중 어떤 기법을 사용할지 결정
- 테스트가 가능한 코드인지 확인
  - 테스트 계획 수립
  - 테스트 코드 작성
- 만족스러울 때까지 반복
  - 코드 개선
  - 컴파일 & 테스트

점진적 리팩토링



- 교각을 보수할 때에는 교각의 기능을 유지한 상태로 보수한다.
- 리팩토링도 기존의 빌드상태와 동작상태를 유지한 상태로 리팩토링이 되어야 한다.
- 간단해 보이는 작업이 Mechanics(절차)이 복잡한 이유는 점진적 리팩토링 때문이다.

Duplicated Code

- 학습목표 1. Duplicated Code의 Code Smell을 판별할 수 있다.
- 학습목표 2. Duplicated Code를 제거할 수 있는 여러 가지 기법을 익힌다.

(1) Code Smell : Duplicated Code

(2) 해결 방법들

Duplicated Code (중복 코드)

1. 설명

- Number one in the stink parade is duplicated code.
- 똑같은 코드 구조가 두 군데 이상 있을 때 그 부분을 하나로 통일하자.

2. 리팩토링 기법

- 한 클래스의 두 메소드에 동일한 코드가 들어있는 경우
  - ✓ Extract Method (메소드 추출, p.142)
- 한 클래스의 두 하위 클래스에 같은 코드가 들어있는 경우
  - ✓ Extract Method (메소드 추출, p.142)를 하고난 후
  - ✓ Pull Up Method (메소드 상향, p.381)
- 코드가 똑같지 않고 비슷한 경우
  - ✓ Extract Method (메소드 추출, p.142)로 같은 부분과 다른 부분 분리
  - ✓ Form Template Method (템플릿 메소드 형성, p.407)
- 두 메소드가 알고리즘만 다르고 기능이 같은 경우
  - ✓ Substitute Algorithm (알고리즘 전환, p.174)
- 서로 상관 없는 두 클래스 안에 중복코드가 있는 경우
  - ✓ Extract Class (클래스 추출, p.187) 이나 모듈로 추출 후 다른 클래스에서 호출
  - ✓ Extract Method로 메소드로 만들고 다른 클래스에서 호출
  - ✓ 코드를 빼서 만든 메소드를 제3의 클래스에 넣고 두 클래스에서 호출

Extract Method (메소드 추출)

1. Summary
- 어떤 코드를 그룹으로 묶어도 되겠다고 판단될 땐
- 그 코드를 빼내어 목적을 잘 나타내는 직관적인 이름의 메소드로 만들자.

2. Motivation
- 가장 많이 사용되는 기법
- 메서드가 너무 길거나 코드에 주석을 달아야만 의도를 이해할 수 있을 때
- 직관적인 이름의 간결한 메소드가 좋은 이유는
  - 다른 메소드에서 쉽게 사용할 수 있다.
  - 상위 메소드에서 메소드명을 주석처럼 읽을 수 있다.
  - 재정의하기 수월하다.
- 메소드명이 메소드 내용을 잘 드러내는 것이 중요

3. Mechanics
- p.143

4. 관련 리팩토링 기법
- 둘 이상의 지역변수가 변경될 때
  - ✓ Split Temporary Variable (임시변수 분리, p.162)
- 임시변수를 제거하거나 개수를 줄이고 싶을 때
  - ✓ Replace Temp with Query (임시변수를 메소드 호출로 전환, p.153)
- 변수를 두 개 이상 반환해야할 때
  - ✓ 각기 다른 값을 반환하는 여러 개의 메소드를 만든다.
- 어떤 방법으로도 메소드 추출이 안되면
  - ✓ Replace Method with Method Object (메소드를 메소드 객체로 전환, p.170)

Split Temporary Variable (임시변수 분리)

1. Summary
- 루프 변수나 값 누적용 임시변수가 아닌 임시변수에 여러 번 값이 대입될 땐
- 각 대입마다 다른 임시변수를 사용하자.

2. Motivation
- 루프 변수 : for에서 i 같은 경우
- 값 누적용 임시변수 : 반복문에서 sum 값은 변수
- 임시변수는 긴 코드의 계산 결과를 간편히 참조할 수 있게 저장하는 용도
- 여러 용도로 사용하는 변수는 각 용도별로 다른 변수를 사용하도록 분리해야 함

3. Mechanics
- p.163

4. 관련 기법
- n/a

Replace Temp with Query (임시변수를 메소드 호출로 전환)

1. Summary
- 수식의 결과를 저장하는 임시변수가 있을 땐
- 그 수식을 빼내어 메소드로 만든 후, 임시변수 참조 부분을 전부 수식으로 교체하자.
- 새로 만든 메소드는 다른 메소드에서도 호출 가능하다.

2. Motivation
- 임시변수는 일시적이고 범위가 제한적
- 임시변수에 접근하기 위해 코드가 길어짐
- Method Extract(메소드 추출)을 쉽게 하기 위해 지역변수를 제거하기 위해 사용
- 가장 간단한 상황은 임시변수에 값을 한 번만 대입하는 경우
- 간편한 리팩토링을 위해 Separate Query from Modifier나 Inline Temp 가능

3. Mechanics
- p.154

4. 관련 기법
- 여러 번 대입되는 변수 제거해야 될 때
  - Split Temporary Variable (임시변수 분리, p.162)
- 추출 메소드가 객체를 변경하고 있을 때
  - Separate Query from Modifier (상태 변경 메소드와 값 반환 메소드를 분리 p.331)
- Inline Temp (임시변수 내용 직접 삽입, p.152)

Separate Query from Modifier (상태 변경 메소드와 값 반환 메소드를 분리)

1. Summary
- 값 반환 기능과 상태 변경 기능이 한 메소드에 들어 있을 땐
- 값 반환 메소드와 상태 변경 메소드로 분리하자.

2. Motivation
- 값을 반환하는 메소드는 눈에 보이는 부작용이 없어야 한다.
- 부작용이란 메소드 바깥의 값을 값을 바꾸는 것을 의미한다.
- 값을 반환하는 메소드가 부작용이 있으면 상태 변경 부분과 값 반환 부분을 분리해야 한다.

3. Mechanics
- p.332

4. 관련 기법
- n/a

Inline Temp (임시변수 내용 직접 삽입)

1. Summary
- 간단한 수식을 대입받는 임시변수로 인해 다른 리팩토링 기법 적용이 힘들 땐
- 그 임시변수를 참조하는 부분을 전부 수식으로 치환하자.

2. Motivation
- Replace Temp with Query(임시변수를 메소드 호출로 전환, p.153) 때 함께 사용된다.
- 메소드 호출의 결과값이 임시변수에 대입될 때 사용된다.
- 임시변수가 다른 리팩토링에 방해가 될 때 사용된다.

3. Mechanics
- p.152

4. 관련 기법
- n/a

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

Replace Method with Method Object (메소드를 메소드 객체로 전환)

1. Summary
- 지역변수 때문에 메소드 추출을 적용할 수 없는 긴 메소드가 있을 땐
- 그 메소드 자체를 객체로 전환해서 모든 지역변수를 객체의 필드로 만들자.
- 그런 다음 그 메소드를 객체 안의 여러 메소드로 쪼개면 된다.

2. Motivation
- 메소드 분해를 어렵게 하는 것은 지역변수이다.
- 일반적으로 Replace Temp with Query(임시변수를 메소드 호출로 전환, p.153)를 시도한다.
- 분해할 수 없을 땐 메소드를 메소드 객체로 변환한다.
- 이 기법을 적용하면 모든 지역변수가 메소드 객체의 속성이 된다.
- 그 후, 객체에 Extract Method를 적용해서 추가 메소드를 만든다.

3. Mechanics
- p.171

4. 관련 기법
- 기본적인 임시변수 제거는
  - ✓ Replace Temp with Query (p.153)

Long Method

- 학습목표 1. Long Method의 Code Smell을 판별할 수 있다.
- 학습목표 2. Long Method를 제거할 수 있는 여러 가지 기법을 익힌다.

(1) Code Smell : Long Method

(2) 해결 방법들

Long Method (장황한 메소드)

1. 설명

- 최적의 상태로 장수하는 프로그램은 공통적으로 메소드의 길이가 짧다.
- 메소드는 짧게 작성해야 한다.
  - 예전엔 메소드 호출의 오버헤드가 CPU 성능에 영향을 주었다. 지금은 아니다.
  - 오히려 이해의 오버헤드가 생겼다.
  - 짧은 메소드를 이해하기 쉽게 하려면 메소드 명을 잘 지어야 한다.
    - 메소드의 이름은 how(기능 수행 방식)보다는 what(기능 자체)이 좋다.
- 주석을 달아야 할 것 같은 부분은 주석 대신 메소드 추출을 한다.
  - 한 줄 밖에 안되는 코드라도 별도의 주석이 달릴 정도라면 메소드 추출을 하자.
- 임시변수가 많으면 메소드 추출이 까다로워진다.
  - 임시변수는 모두 제거한다.
  - 메소드 추출을 하면서 임시변수를 매개변수로 넘기면 안된다.
    - 가독성 측면에서 개선이 안된다.
- 조건문과 반복문도 메소드로 분리한다.

Long Method (장황한 메소드)

2. 리팩토링 기법

- 메소드 길이를 줄이려면
  - ✓ Extract Method (메소드 추출, p.142)
- 임시변수를 제거하려면
  - ✓ Replace Temp with Query (임시변수를 메소드 호출로 전환, p.153)
- 매개변수 개수가 많으면
  - ✓ Introduce Parameter Object (매개변수 세트를 객체로 전환, p.351)
  - ✓ Preserve Whole Object (객체를 통째로 전달, p. 342)
- 그래도 임시변수와 매개변수가 너무 많을 땐
  - ✓ Replace Method with Method Object (메소드를 메소드 객체로 전환, p.170)
- 조건문을 추출하려면
  - ✓ Decompose Conditional (조건문 쪼개기, p.286)

Introduce Parameter Object (매개변수 세트를 객체로 전환)

1. Summary
- 여러 개의 매개변수가 항상 붙어 다닐 땐
- 그 매개변수들을 객체로 바꾸자.

2. Motivation
- 특정 매개변수들은 늘 항상 함께 전달된다.
- 새 객체에 정의된 속성 접근 메소드로 인해 코드의 일관성이 개선된다.
  - 결과적으로 코드를 알아보거나 수정하기 쉬워진다.
- 매개변수를 한 덩이로 만들면 기능을 새 클래스로 옮길 수 있다.
  - 메소드 안에 매개변수 값에 대한 공통적인 조작을 넣는 경우가 많다.
  - 이 동작을 새 객체로 옮기면 상당량의 중복 코드를 없앨 수 있다.

3. Mechanics
- p.351

4. 관련 기법
- 새 데이터 뭉치에 매개변수를 추가할 때
  - Add Parameter (매개변수 추가, p.327)
- 매개변수 객체에 기능을 옮길 때
  - Move Method (메소드 이동, p178)

Add Parameter (매개변수 추가)

1. Summary
- 메소드가 자신을 호출한 부분의 정보를 더 많이 알아야 할 땐
- 객체에 그 정보를 전달할 수 있는 매개변수를 추가하자.

2. Motivation
- 메소드를 수정해야 하는데 기존에 전달하지 않은 정보가 있을 때 사용한다.
- 매개변수를 추가하지 않을 대안이 있는 경우 대안을 선택한다.
  - 매개변수 길이가 길어지면 Code Smell이 생긴다.
- 새로운 파라메터를 추가해야할 때
  - 파라메터 중 하나에 필요한 정보를 요청할 수 있나?
  - 파라메터 중 하나에 필요한 정보를 가져오는 메소드를 추가하는 것이 합리적인가?
  - 필요한 정보는 무슨 용도로 사용되나?
  - 정보가 들어있어야 하는 다른 객체가 있나?
  - Introduce Parameter Object (매개변수 세트를 객체로 전환, p.351)을 적용해야 하나?

3. Mechanics
- p.328

4. 관련 기법
- n/a

Move Method (메소드 이동)

1. Summary
- 메소드가 자신이 속한 클래스보다 다른 클래스의 기능을 더 많이 이용할 땐
- 그 메소드가 제일 많이 이용하는 클래스 안에서 비슷한 내용의 새 메소드를 작성하자.
- 기존 메소드는 간단한 대리 메소드로 전환하든지 아예 삭제하자.

2. Motivation
- 클래스에 기능이 너무 많을 때 사용
- 클래스가 다른 클래스와 과하게 연동되어 의존성이 지나칠 때 사용
- 일부 속성을 옮길 때 이 작업을 같이 하는 것이 좋다.

3. Mechanics
- p.179

4. 관련 기법
- n/a

Preserve Whole Object (객체를 통째로 전달)

1. Summary
- 객체에서 가져온 여러 값을 메소드 호출에서 매개변수로 전달할 땐
- 그 객체를 통째로 전달하게 수정하자.

2. Motivation
- 객체가 한 객체에 든 여러 값을 메소드 호출할 때 매개변수로 전달하고 있다면 사용
  - 호출된 객체가 새 데이터가 필요로 할 때마다 메소드 수정하는 문제 발생
  - 데이타를 넘겨주는 객체 자체를 전달하자.
- 매개변수 세트 변경의 편이성이 높아지고 코드 가독성이 좋아진다.
- 파라메터로 전달하는 객체와 의존성이 생기는 단점이 생긴다.
- 통 객체에서 가져올 값이 하나인 경우
  - 코드 명료성 입장에서는 통으로 가져오나 값으로 가져오나 차이 없다.
  - 성능은 약간 떨어질 수 있으나 무시할 수 있다.
  - 불필요한 의존성이 생길 수 있다.
- 대안으로 Move Method (메소드 이동, p.178)를 사용할 수 있다.

3. Mechanics
- p.343

4. 관련 기법
- n/a

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```


리팩토링 p.342

Decompose Conditional (조건문 쪼개기)

1. Summary (요약)
- 복잡한 조건문(if-then-else)이 있을 땐
- If, then, else 부분을 각각 메소드로 빼내자.

2. Motivation (동기)
- 프로그램에서 가장 복잡한 부분은 주로 복잡한 조건문이다.
- 조건 검사 코드와 액션 코드의 원리는 알 수 있지만 왜 그렇게 되는지 알기 힘들다.
- 큰 덩어리의 조건을 잘게 쪼개고 각 코드 조각을 용도에 맞는 이름의 메소드로 바꾼다.
- 조건이 눈에 잘 들어오고 갈라지는 로직 흐름을 알아보기 쉬워진다.

3. Mechanics (방법)
- p.286

4. 관련 기법
- 조건문이 여러 겹일 땐
  - ✓ Replace Nested Conditional with Guard Clauses (여러 겹의 조건문을 감시 절로 전환, p.299)

Replace Nested Conditional with Guard Clauses (여러 겹의 조건문을 감시 절로 전환)

1. Summary (요약)

- 메소드에 조건문이 있어서 정상적인 실행 경로를 파악하기 힘들 땐
- 모든 특수한 경우에 감시 절을 사용하자.

2. Motivation (동기)

- 조건문이 특이한 조건이라면 그 조건을 검사해서 조건이 true인 경우 return 한다.
  - 이런 식의 검사를 Guard Clause(감시절)이라고 한다.
- Guard Clause(감시절)은 "이것은 드문 경우이니 이 경우가 발생하면 작업을 수행한 후 빠져나와라 " 라고 하는 방법이다.
- Exit point가 하나인 로직은 복잡해진다. Exit point를 하나만 고집하지 말자.

3. Mechanics (방법)

- p.300

4. 관련 기법

- 모든 감시절의 결과가 같다면
  - ✓ Consolidate Conditional Expressions (중복 조건식 통합, p.288)

Consolidate Conditional Expression (중복 조건식 통합)

1. Summary (요약)
- 여러 조건 검사식의 결과가 같을 땐
- 하나의 조건문으로 합친 후 메소드로 빼내자.

2. Motivation (동기)
- 조건식의 결과가 모두 같을 때 AND와 OR을 사용해서 하나로 합친다.
- 식이 간결해지고 Extract Method가 쉬워진다.
- 조건 검사식이 독립적이고 하나의 검사로 인식되면 안될 땐 실시하면 안된다.

3. Mechanics (방법)
- p.289

4. 관련 기법
- Extract Method (메소드 추출, p.142)

Large Class

- 학습목표 1. Large Class의 Code Smell을 판별할 수 있다.
- 학습목표 2. Large Class를 제거할 수 있는 여러 가지 기법을 익힌다.

(1) Code Smell : Large Class

(2) 해결 방법들

Large Class (방대한 클래스)

1. 설명

- 너무 많은 일을 하는 클래스는 보통 인스턴스 변수가 많이 있다.
  - 인스턴스 변수가 많으면 중복코드가 반드시 있게 마련이다.

2. 리팩토링 기법

- 인스턴스 변수를 하나로 묶으려면
  - ✓ Extract Class (클래스 추출, p.187)을 한다.
  - ✓ 서로 연관된 변수를 골라서 클래스로 빼낸다.
  - ✓ 접두어와 접미어가 같은 경우에 클래스로 뺄 가능성이 높다.
    - ✓ depositAmount, depositCurrency
- 클래스가 커서 중복코드를 제거하고 싶으면
  - ✓ Extract Class (클래스 추출, p.187)
  - ✓ Extract Subclass (하위클래스 추출, p.390)
  - ✓ 필요에 따라 Extract Interface (인터페이스 추출, p.403)를 실시한다.
- 방대한 클래스가 GUI 클래스인 경우 data와 behavior를 분리해야될 때
  - ✓ 중복 데이터는 놔두고 그 데이터와 싱크를 유지해야 한다면,
  - ✓ Duplicate Observed Data (관측 데이터 복제, p.231)을 한다.

Extract Class (클래스 추출)

1. Summary
- 두 클래스가 처리해야 할 기능이 하나의 클래스에 들어 있을 땐
- 새 클래스를 만들고 기존 클래스의 관련 필드와 메소드를 새 클래스로 옮기자.

2. Motivation
- 클래스는 완벽히 추상화되어야 한다.
  - 두세 가지의 명확한 기능을 담당해야 한다.
- 시간이 갈수록 담당하는 기능이 많아진다.
  - 별도의 클래스로 만들기엔 사소하지만 기능들도 점점 추가된다.
  - 그런 클래스는 보통 메소드와 데이터가 많아서 이해하기 힘들다.
- 데이터의 일부분과 메소드의 일부분이 한 덩이이거나
  - 주로 함께 변화하거나 서로 유난히 의존적인 데이타의 일부분도
  - 클래스로 떼어내기 쉽다.
- 데이터나 메소드로 하나 제거하면 어떻게 될지,
  - 다른 필드와 메소드를 추가하는 건 합리적이지 않은지 자문하여 확인한다.

3. Mechanics
- p.187

4. 관련 기법
- Move Field (필드 이동, p.183)
- Move Method (메소드 이동, p.178)

Move Field (필드 이동)

1. Summary
- 어떤 필드가 자신이 속한 클래스보다 다른 클래스에서 더 많이 사용될 때는
- 대상 클래스 안에 새 필드를 선언하고 그 필드 참조 부분을 전부 새 필드 참조로 수정하자.

2. Motivation
- 한 클래스에서 다른 클래스로 상태와 기능을 옮기는 것은 리팩토링의 기본이다.
  - 지금은 합리적이고 올바르다고 판단되는 설계라도 나중에는 그렇지 않을 수 있다.
- 어떤 필드가 자신이 속한 클래스보다 다른 클래스에 있는 메소드를 더 많이 참조하면
  - 그 필드를 옮긴다.

3. Mechanics
- p.183

4. 관련 기법
- Extract Class (클래스 추출, p.187)
- Self Encapsulate Field (필드 자체 캡슐화, p.211)

Shotgun Surgery

Shotgun Surgery

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
6	Shotgun Surgery	기능의 산재 (105)	수정할 때마다 수많은 자잘한 부분을 고쳐야 한다.	Move Method (178), Move Field (183), Inline Class (192)	설계	★★★

- 소프트웨어를 변경할 때 명확히 한 곳을 집어 변경할 수 있어야 됨.
  - 이렇게 할 수 없을 때 밀접한 관계가 있는 두 가지 냄새 중에 하나
    - Divergent Change, Shotgun Surgery
- 하나를 변경했을 때 많은 클래스를 고쳐야 하는 경우
  - Divergent Change과 비슷하지만 정 반대
  - 변경해야 할 것이 여기저기 널려있다면,
    - 찾기도 어렵고,
    - 변경해야 할 중요한 사항을 빼먹기도 쉬움
- Move Method와 Move Field를 사용하여
  - 변경해야 할 부분을 모두 하나의 클래스로 묶음
  - 만약 기존의 클래스 중에서 메소드나 필드가 옮겨갈 적절한 클래스가 없는 경우
    - 새로 하나를 만듬
- Inline Class를 사용하여 모든 동작을 하나로 모음
  - Divergent Change 냄새가 날 수 있음

Inline Class

1. Summary
- 클래스에 기능이 너무 적을 땐
- 그 클래스의 모든 기능을 다른 클래스로 합쳐 놓고 원래의 클래스는 삭제하자.

2. Motivation
- Inline Class는 Extract Class와 반대다.
- 클래스가 더 이상 제 역할을 수행하지 못하여 존재할 이유가 없을 때 실시.
- 리팩토링 후 남은 기능이 없는 경우
  - 이 작은 클래스를 가장 많이 사용하는 클래스를 하나 고른 후 합침.

3. Mechanics
- p.192

4. 관련 기법
- Extract Interface
- Move Method
- Move Field

Extract Interface

1. Summary
- 클래스 인터페이스의 같은 부분을 여러 클라이언트가 사용하거나
- 두 클래스에 인터페이스의 일부분이 공통으로 들어 있을 땐
- 공통 부분을 인터페이스로 빼내자.

2. Motivation
- 클래스를 사용한다는 건 주로 클래스의 기능 전반을 사용한다는 뜻
  - 다른 경우는 여러 클라이언트가 특정 부분만 사용하는 경우
  - 한 클래스가 여러 클래스와 함께 사용하는 경우
- 마지막 두 가지의 경우 책임진 부분을 독립시켜 어떤 부분을 사용하는지 명확하게 하는 것이 좋음.
- 유사한 Extract Superclass, 공통된 코드를 추출
- 중복된 코드 냄새를 초래할 수 있음
  - Extract Class를 사용해서 동작을 컴포넌트로 만들어서 위임
  - Extract Superclass는 뺄 수 있는 클래스가 단 한 개

3. Mechanics
- p.404

4. 관련 기법
- Extract Superclass

Extract Superclass

1. Summary
- 클래스 인터페이스의 같은 부분을 여러 클라이언트가 사용하거나
- 두 클래스에 인터페이스의 일부분이 공통으로 들어 있을 땐
- 공통 부분을 인터페이스로 빼내자.

2. Motivation
- 클래스를 사용한다는 건 주로 클래스의 기능 전반을 사용한다는 뜻
  - 다른 경우는 여러 클라이언트가 특정 부분만 사용하는 경우
  - 한 클래스가 여러 클래스와 함께 사용하는 경우
- 마지막 두 가지의 경우 책임진 부분을 독립시켜 어떤 부분을 사용하는지 명확하게 하는 것이 좋음.
- 유사한 Extract Superclass, 공통된 코드를 추출
- 중복된 코드 냄새를 초래할 수 있음
  - Extract Class를 사용해서 동작을 컴포넌트로 만들어서 위임
  - Extract Superclass는 뺄 수 있는 클래스가 단 한 개

3. Mechanics
- p.192

4. 관련 기법
- Extract Superclass

Temporary Field

Temporary Field

No	Bad Smells	Bad Smells 한글	의미	Common Refactorings	분류	중요도
14	Temporary Field	임시 필드 (110)	객체 안에 인스턴스 변수가 특정 상황에서만 할당된다.	Extract Class (187), Introduce Null Object (310)	설계	★★

- 어떤 객체 안의 인스턴스 변수가 특정 상황에서만 세팅되는 경우
  - 이런 코드는 어려운데, 왜냐하면 보통의 객체의 모든 변수가 값을 가지고 있을 거라고 기대하기 때문
  - 사용되지 않는 것처럼 보이는 변수가 왜 있는지를 이해하는 것은 매우 짜증나는 일
- 불쌍한 고아 변수들을 위한 집을 만들기 위해 Extract Class 사용
  - 그 변수를 사용하는 모든 코드를 새로 만든 클래스로 이동
- 변수의 값이 유효하지 않은 경우에 대한 대체 컴포넌트를 만들기 위해
  - Introduce Null Object를 사용하여 조건문이 포함된 코드 제거
- 임시 필드는 복잡한 알고리즘이 여러 변수를 필요로 할 때 흔히 발생
  - 필요한 변수와 메소드를 묶어 Extract Class 사용
  - 새로운 객체는 메소드 객체가 됨

Introduce Null Object (Null 검사를 널 객체에 위임)

1. Summary
- null 값을 검사하는 코드가 계속 나올 땐
- null 값을 널 객체로 만들자.

2. Motivation
- Polymorphism의 본질은 객체에 일일이 물어서 실행하는 것이 아니라
  - 묻지도 따지지도 않고 기능을 바로 호출하는 것이다.
  - 객체가 null인 경우 이 동작은 직관적이지 못하게 된다..

3. Mechanics
- p.306

4. 관련 기법

조건문 간결화

- 학습목표 1. 복잡한 조건문의 문제점을 파악할 수 있다.
- 학습목표 2. 복잡한 조건문을 개선할 수 있는 여러 가지 기법을 익힌다.

(1) Code Smell : 복잡한 조건문

(2) 해결 방법들

복잡한 조건문

1. 설명

- 프로그램이 커질수록 조건문이 복잡해질 가능성이 높다.

2. 리팩토링 기법

- 조건문을 여러 개로 나누고 싶을 때
  - ✓ Decompose Conditional(조건문 쪼개기, p.286)
- 여러 조건 검사가 있는데 결과가 모두 같을 때
  - ✓ Consolidate Conditional Expression (중복 조건식 통합, p.288)
- 조건문 안의 중복 코드를 제거하고 싶을 때
  - ✓ Consolidate Duplicate Conditional Fragments (조건문의 공통 실행 코드 빼내기, p.291)
- 특수한 case 조건문을 명확히 하고 싶을 때
  - ✓ Replace Nested Conditional with Guard Clauses (여러 겹의 조건문을 감시 절로 전환, p299)
- 복잡한 제어 플래그를 제거하고 싶을 때
  - ✓ Remove Control Flag (제어 플래그 제거, p.293)
- Switch-case 문을 재정의로 전환하고 싶을 때
  - ✓ Replace Conditional with Polymorphism (조건문을 재정의로 전환, p.305)
- Null 값을 검사하는 코드가 계속 나올 땐
  - ✓ Introduce Null Object (Null 검사를 널 객체에 위임, p.310)

Consolidate Duplicate Conditional Fragments (조건문의 공통 실행 코드 빼내기)

1. Summary (요약)

- 조건문의 모든 절에 같은 실행 코드가 있을 땐
- 같은 부분을 조건문 밖으로 빼자.

2. Motivation (동기)

- 공통적으로 실행될 기능과 다르게 실행할 기능을 구분하기 쉬워진다.

3. Mechanics (방법)

- p.291

4. 관련 기법

- n/a

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

Remove Control Flag (제어 플래그 제거)

1. Summary (요약)
- 논리 연산식의 제어 플래그 역할을 하는 변수가 있을 땐
- 그 변수를 break 문이나 return 문으로 바꾸자.

2. Motivation (동기)
- 특정 플래그가 ON 될 때 로직을 빠져나오게 하는 변수
  ex) found 변수 같은 것
- break, continue, return으로 코드를 단순화한다.

3. Mechanics (방법)
- p.293

4. 관련 기법
- Extract Method

메소드 정리

- 학습목표 1..
- 학습목표 2. 복잡한 조건문을 개선할 수 있는 여러 가지 기법을 익힌다.

(1) Code Smell : 복잡한 조건문

(2) 해결 방법들

메소드 정리

1. 설명
- 거의 모든 문제점은 장황한 메소드로 인해 생긴다.
- 메소드의 많은 정보를 가독성이 높게 잘 정리해야 한다.

2. 관련 리팩토링 기법
- Extract Method (메소드 추출, p.142)
- Inline Method (메소드 내용 직접 삽입, p.150)
- Replace Temp with Query (임시변수를 메소드 호출로 전환, p.153)
- Introduce Explaining Variable (직관적 임시변수 사용, p.157)
- Split Temporary Variable (임시변수 분리, p.162)
- Substitute Algorithm (알고리즘 전환, p.174)

Inline Method (메소드 내용 직접 삽입)

1. Summary (요약)

- 메소드 기능이 너무 단순해서 메소드명만 봐도 너무 뻔할 땐
- 그 메소드의 기능을 호출하는 메소드에 넣어버리고 그 메소드는 삭제하자.

2. Motivation (동기)

- 메소드명에 모든 기능이 반영될 정도로 메소드 기능이 지나치게 단순할 땐 그 메소드를 없앤다.
- 잘못 쪼개진 메소드는 전부 하나의 큰 메소드로 합친 후 다시 각각의 작은 메소드로 Method Extract 한다.

3. Mechanics (방법)

- p.151

4. 관련 기법

- Extract Method

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Introduce Explaining Variable (직관적 임시변수 사용)

1. Summary (요약)
- 사용된 수식이 복잡할 땐
- 수식의 결과나 수식의 일부분을 용도에 부합하는 직관적인 이름의 임시변수에 대입하자.

2. Motivation (동기)
- 수식이 너무 복잡해서 이해하기 힘들 때 임시변수를 사용해서 수식을 쪼갠다.
- 조건문에서 각 조건 절을 가져와서 직관적인 이름의 임시변수를 사용해 그 조건의 의미를 설명한다.
- 긴 알고리즘에서 임시변수를 사용해서 계산의 각 단계를 설명한다.
- 일반적으로 Method Extract가 더 좋은 방법이다. 지역변수로 힘든 상황에 사용한다.

3. Mechanics (방법)
- p.158

4. 관련 기법
- Extract Method

Substitute Algorithm (알고리즘 전환)

1. Summary (요약)
- 알고리즘을 더 분명한 것으로 교체해야 할 땐
- 해당 메소드의 내용을 새 알고리즘으로 바꾸자.

2. Motivation (동기)
- 적용된 알고리즘보다 더 좋은 알고리즘이나 라이브러리를 찾었을 때
- 어떤 작업을 약간 다르게 처리해야할 때
- 메소드가 잘게 쪼개져 있어야 수정하기 쉽다.

3. Mechanics (방법)
- P.175

4. 관련 기법
- n/a

데이터 체계화

- 학습목표 1. 복잡한 데이타의 문제점을 파악할 수 있다.
- 학습목표 2. 복잡한 데이타 개선할 수 있는 여러 가지 기법을 익힌다.

(1) Code Smell : 복잡한 데이타

(2) 해결 방법들

데이터 체계화

1. 설명

- 리팩토링을 통해 데이터 관리를 잘 할 수 있도록 한다.

2. 관련 리팩토링 기법

- Replace Array with Structure (배열을 구조체로 전환, p.227)
- Replace Magic Number with Symbolic Constant (마법 숫자를 기호 상수로 전환, p.248)

Replace Array with Object (배열을 객체로 전환)

1. Summary (요약)
- 배열을 구성하는 특정 원소가 별의별 의미를 지닐 땐
- 그 배열을 각 원소마다 필드가 하나씩 든 객체로 전환하자.

2. Motivation (동기)
- 배열은 비슷한 유형 데이터를 저장하는 용도로 사용된다.
- "배열의 첫 원소는 이름이다"라는 것과 같이 사용될 경우 객체로 전환한다.

3. Mechanics (방법)
- p.227

4. 관련 기법
- n/a

Replace Magic Number with Symbolic Constant (마법 숫자를 기호 상수로 전환)

1. Summary (요약)

- 특수 의미를 지닌 리터럴 숫자가 있을 땐
- 의미를 살린 이름의 상수를 작성한 후 리터럴 숫자를 그 상수로 교체하자.

2. Motivation (동기)

- Magic number(=마법 숫자)는 특정한 값을 갖는 숫자
- Magic number 대신 상수를 사용하면 단점이나 부작용 없이 가독성이 향상된다.

3. Mechanics (방법)

- p.249

4. 관련 기법

- n/a

메소드 호출 단순화

- 학습목표 1. 복잡한 메소드 호출 관계의 문제점을 파악할 수 있다.
- 학습목표 2. 복잡한 메소드 호출 관계를 개선할 수 있는 여러 가지 기법을 익힌다.

(1) Code Smell : 복잡한 호출 관계

(2) 해결 방법들

메소드 호출 단순화

1. 설명

- 리팩토링을 통해 메소드 호출을 잘 할 수 있도록 한다.

2. 관련 리팩토링 기법

- Rename Method (메소드명 변경, p.325)
- Add Parameter (매개변수 추가, p.327)
- Remove Parameter (매겨변수 제거, p.329)
- Parameterize Method (메소드를 매개변수로 전환, p.336)
- Replace Parameter with Explicit Methods (매개변수를 메소드로 전환, p.338)
- Introduce Parameter Object (매개변수 세트를 객체로 전환, p.351)

Rename Method (메소드명 변경)

1. Summary (요약)
- 메소드명을 봐도 기능을 알 수 없을 땐
- 메소드 명을 직관적인 이름으로 바꾸자.

2. Motivation (동기)
- 복잡한 과정은 작은 메소드로 쪼갠다.
- 작은 메소드의 역할을 파악하기 힘들어지는 것을 방지하기 위해 메소드 명을 잘 지어야 한다.
- 메소드명을 설명하기 위한 주석을 떠올린 후 주석을 메소드명으로 바꾼다.
- 매개변수명도 중요하다.

3. Mechanics (방법)
- p.325

4. 관련 기법
- Add Parameter (매개변수 추가, p.327)
- Remove Parameter (매개변수 제거, p.329)

Remove Parameter (매개변수 제거)

1. Summary (요약)
- 메소드가 어떤 매개변수를 더 이상 사용하지 않을 땐
- 그 매개변수를 삭제하자.

2. Motivation (동기)
- 나중에 필요할 거라는 이유로 사용하지 않는 파라메터를 남겨두지 말자.
- 불필요한 파라메터는 그 파라메터를 사용하는 불필요한 작업을 만들 수 있다.

3. Mechanics (방법)
- P.330

4. 관련 기법
- Add Parameter (매개변수 추가, p.327)

Parameterize Method (메소드를 매개변수로 전환)

1. Summary (요약)

- 여러 메소드가 기능은 비슷하고 안에 든 값만 다를 땐
- 서로 다른 값을 하나의 매개변수로 전달받는 메소드를 하나 작성하자.

2. Motivation (동기)

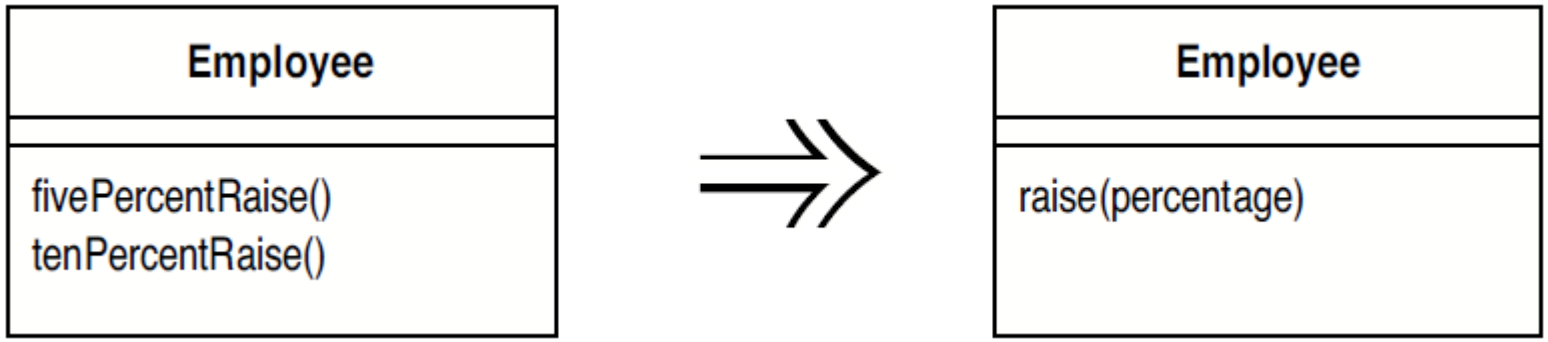
- 기능은 비슷하지만 몇 가지 값에 따라 결과가 달라지는 메소드가 여러 개 있을 때 하나의 메소드로 합친다.
- 중복 코드가 제거되고 다양한 것을 처리할 수 있는 유연성이 커진다.

3. Mechanics (방법)

- P.336

4. 관련 기법

- n/a



Replace Parameter with Explicit Methods (매개변수를 메소드로 전환)

1. Summary (요약)

- 매개변수로 전달된 값에 따라 메소드가 다른 코드를 실행할 땐
- 그 매개변수로 전달될 수 있는 모든 값에 대응하는 메소드를 각각 작성하자.

2. Motivation (동기)

- 매개변수를 판단하는 조건문이 없어진다.
- 쪼개진 메소드는 컴파일할 때 검사가 된다.
- 매개변수 값이 많이 변할 가능성이 있을 때는 적용하면 안된다.

3. Mechanics (방법)

- P.339

4. 관련 기법

- n/a

```
void setValue (String name, int value) {
    if (name.equals("height")) {
        _height = value;
        return;
    }
    if (name.equals("width")) {
        _width = value;
        return;
    }
    Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {
      _height = arg;
}
void setWidth (int arg) {
      _width = arg;
}
```


리팩토링 p.338

일반화 처리

- 학습목표 1. 일반화 처리가 필요한 경우를 알 수 있다.
- 학습목표 2. 일반화 처리를 할 수 있는 여러 가지 기법을 익힌다.

(1) Code Smell : 일반화 처리

(2) 해결 방법들

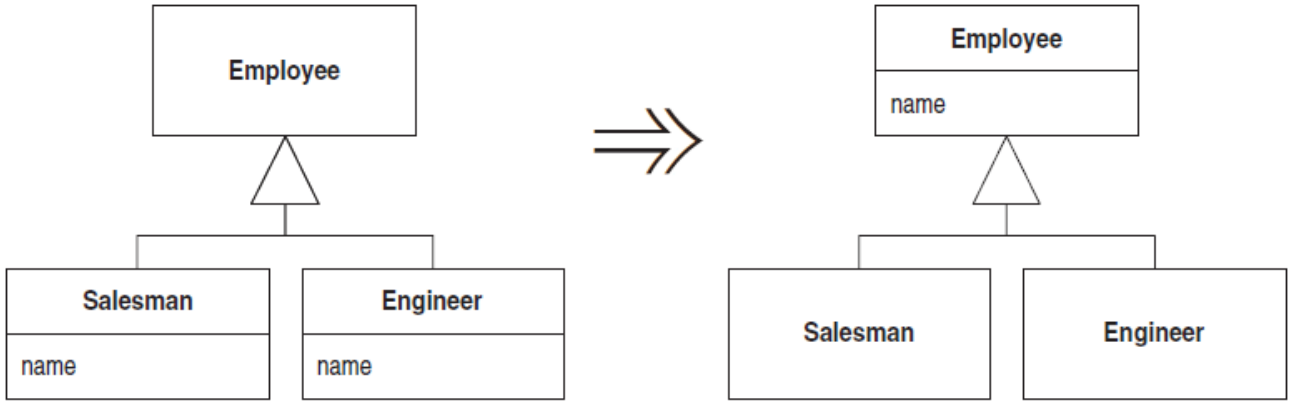
Pull Up Field (필드 상향)

1. Summary
- 두 하위클래스에 같은 필드가 들어있을 땐
- 필드를 상위클래스로 옮기자.

2. Motivation
- 각각의 하위클래스를 따로 개발 중이라면,
  - 여러 하위클래스에 중복된 기능이 들어있는 경우가 많다.
- 필드 이름이 비슷할 때 용도를 파악하기 위해
  - 다른 메소드에 어떤 식으로 사용되는지를 확인한다.
- 중복된 필드가 서로 비슷한 방식으로 사용되면
  - 그 필드를 일반화한다.
- 중복이 줄어든다.
  - 데이터 선언의 중복이 없어진다.
  - 필드를 사용하는 기능이 하위클래스에서 상위클래스로 옮겨진다.

3. Mechanics
- p.380

4. 관련 기법
- n/a



리팩토링 p.380

Pull Up Method (메소드 상향)

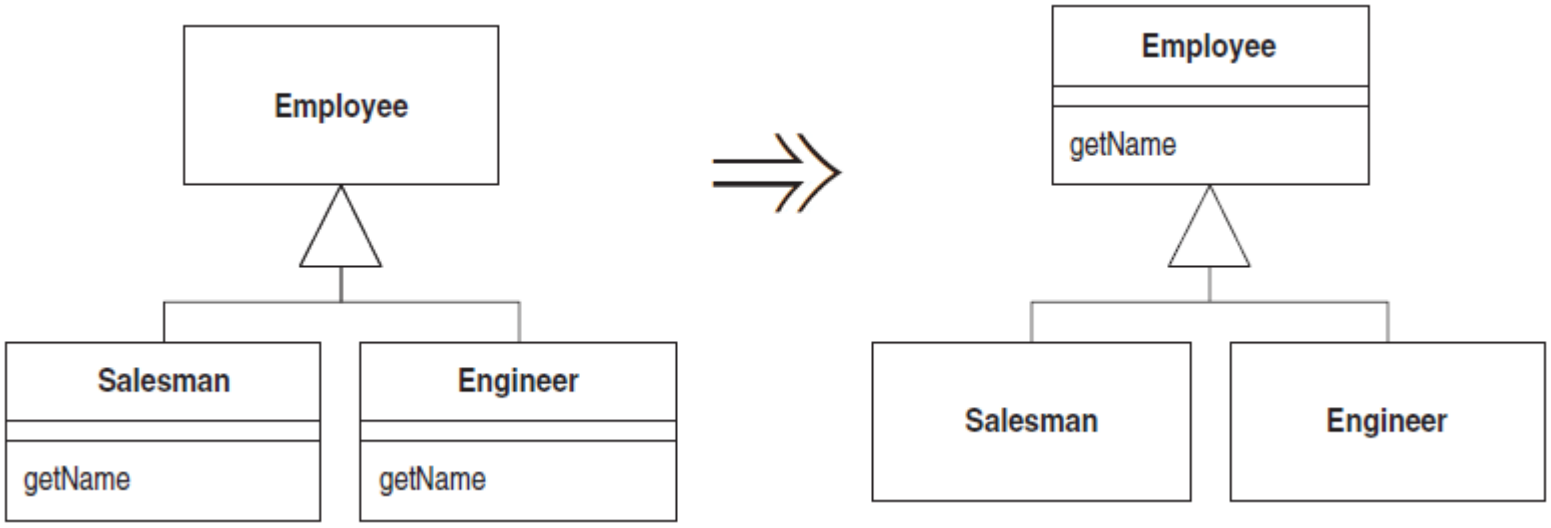
1. Summary
- 기능이 같은 메소드가 여러 하위클래스에 들어 있을 땐
- 그 메서드를 상위클래스로 옮기자.

2. Motivation
- 중복된 코드는 한 쪽만 수정하고 다른 쪽은 수정하지 않는 위험한 경우가 많다.
- Pull Up Method를 적용해야 할 가장 단순한 상황은 중복 코드 발견이다.
- Pull Up Method는 다른 리팩토링 단계를 마친 후 적용하는 게 일반적이다.
- 하위클래스에는 있고 상위클래스에는 없는 기능을 참고할 때에는
  - 그 기능을 상위클래스에 옮기거나
  - 메서드의 시그니처를 수정하거나 위임하는 메서드를 작성할 수 있다.
- 두 메소드가 비슷한 부분이 있다면
  - Form Template Method (템플릿 메소드 형성, p.407) 실시한다.

3. Mechanics
- p.382

4. 관련 기법
- 메소드가 하위 클래스의 필드를 사용한다면
  - Pull Up Field (필드 상향, p.380)
  - Encapsulate Field (필드 자체 캡슐화, p.211)을 적용

Pull Up Method (메소드 상향)



상위클래스로 옮기기 전 구조

상위클래스로 옮긴 후 구조

리팩토링 p.381

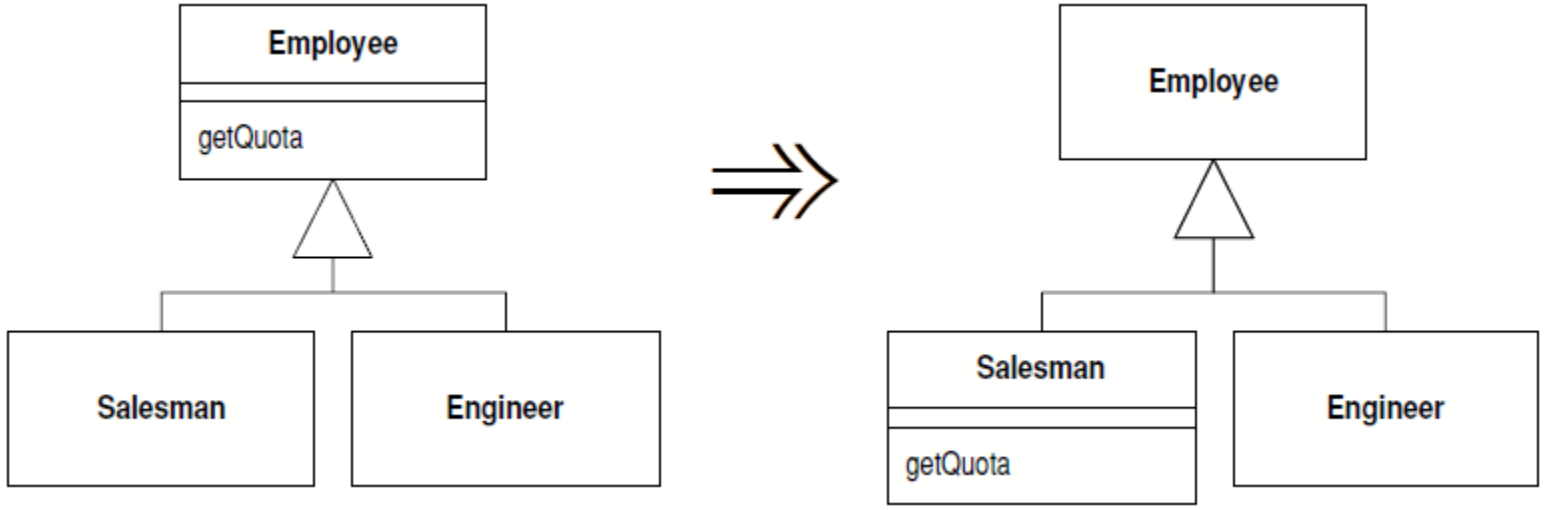
Push Down Method (메소드 하향)

1. Summary
- 상위클래스에 있는 기능을 일부 하위클래스만 사용할 땐
- 그 기능을 관련된 하위클래스 안으로 옮기자.

2. Motivation
- Pull Up Method (메소드 상향, p.381)과 반대되는 기법이다.
- Extract Subclass (하위클래스 추출, p.390)을 실시할 때 흔히 사용한다.

3. Mechanics
- p.388

4. 관련 기법
- n/a



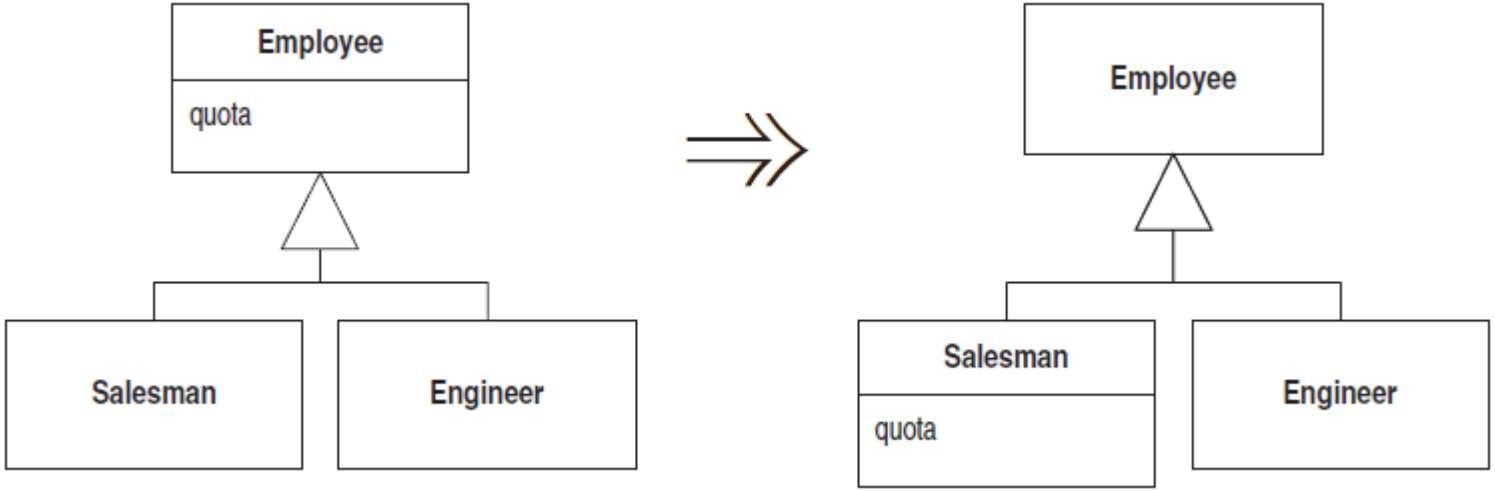
Push Down Field (필드 하향)

1. Summary
- 일부 하위클래스만이 사용하는 필드가 있을 땐
- 그 필드를 사용하는 하위클래스로 옮기자.

2. Motivation
- Pull Up Field (필드 상향, p.380)과 반대되는 기법이다.
- 필드가 상위클래스엔 필요 없고 하위클래스만 필요할 때 사용한다.

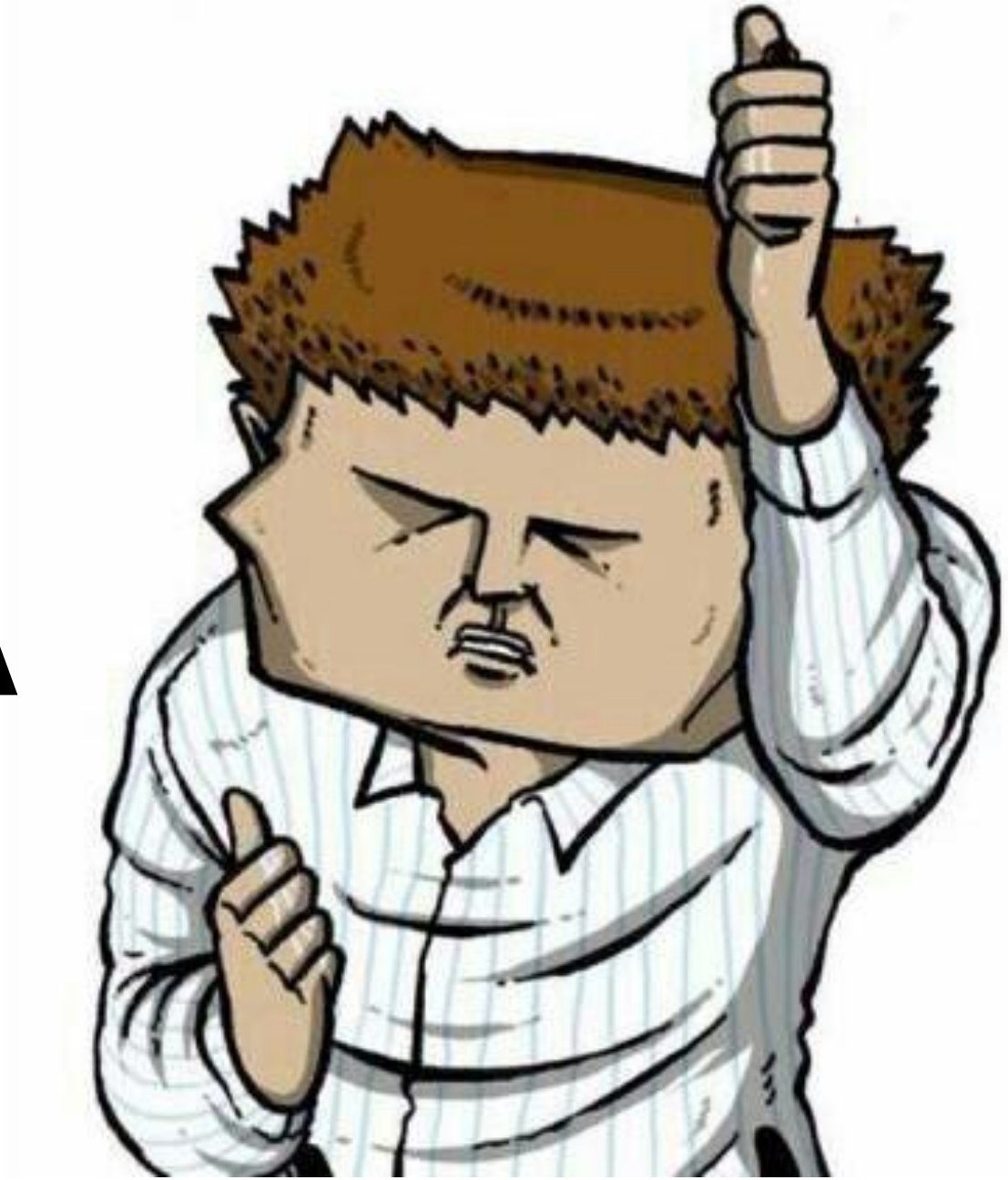
3. Mechanics
- p.389

4. 관련 기법
- n/a



리팩토링 p.389

Q&A




That's all Folks!