SAMSUNG SDS
Realize your vision

Foresee

Techtonic 2019

Disrupt

Partner

2019.11.14 • SAMSUNG SDS Tower B1F
{ Magellan Hall / Pascal Hall }

Track 2 | Clean Code

클린코드를 위한 리팩토링 체험

이동석 프로 (코드품질그룹) / 삼성SDS
최재원 프로 (코드품질그룹) / 삼성SDS

우리는 알기 바랍니다…

- 깨끗한 코드를 작성해야 하는 이유를…
- 리팩토링의 다양한 방법과 기법을…
- 리팩토링이 가져다 주는 이점을…
- 리팩토링 과정의 고난과 희열을…

AGENDA

1. Clean Code
2. Refactoring
3. Legacy code 소개
4. Test case 작성
5. Refactoring 체험

1

Clean Code

Clean Code란 ?

“모든 팀원이 이해(understandability)하기 쉽도록 작성된 코드”

- 의존성이 적어 단순하고, 테스트코드로 검증된 코드
- 코드를 해석하는 시간과 수정하는 시간의 비율은 10(read) : 1 (write)
- 대부분의 결함은 기존 코드 수정 시에 발생되므로 이해하기 쉬운 코드는 오류의 위험을 최소화함

Readability, Changeability, Extensibility, Maintainability…

Clean Code의 주요 영역

Naming

Comments

Data Structure

Method

Style

Class

Generics

Concurrency

Error Handling

Annotation

Performance

Unit Test

Architecture

2

Refactoring

Refactoring 이란?

기능은 그대로 동작하도록 두고, 소프트웨어 내부를 수정하는 작업

- 제대로 실행되나 구조가 완전하지 못한 코드를 대상으로 작업
- 코드를 이해하고 수정하기 더 쉽도록 소프트웨어 내부를 변경
- 겉으로 드러나는 기능에는 영향을 주지 않음
- **기능이 변하지 않는다는 것을 증명해주는 테스트 코드가 반드시 필요!**

왜 리팩토링을 하는가?

1. **소프트웨어 설계가 개선되니까** 중복된 코드가 없어지면서 설계가 개선된다.
2. **소프트웨어 이해가 더 쉬워지니까** 코드가 깔끔해져 빠르게 파악하고 수정할 수 있다.
3. **버그를 찾기가 더 쉬워지니까** 프로그램 구조의 단순·명료화로 버그 원인을 쉽게 찾는다.
4. **프로그래밍 속도가 빨라지니까** 깔끔한 설계로 소프트웨어 개발속도가 높아진다.

언제 리팩토링을 하는가?

- **작정하고 따로 시간을 내서 하는 것이 아닌 일상적으로 하는 것이 좋다.**
  - 기능을 추가할 때
  - 버그를 수정할 때
  - 코드리뷰를 할 때

- **개발 중에 능률을 높이기 위해서 틈틈히 하는 것이다.**
  - 코드를 알아보기 힘들 때
  - 중복된 로직이 들어있을 때
  - 추가 기능을 넣어야 해서 레거시 코드를 변경해야 할 때
  - 조건문 구조가 복잡할 때

어떻게 리팩토링을 하는가?

Find some code that **"smells"**

↓

Determine how to **simplify** this code

↓

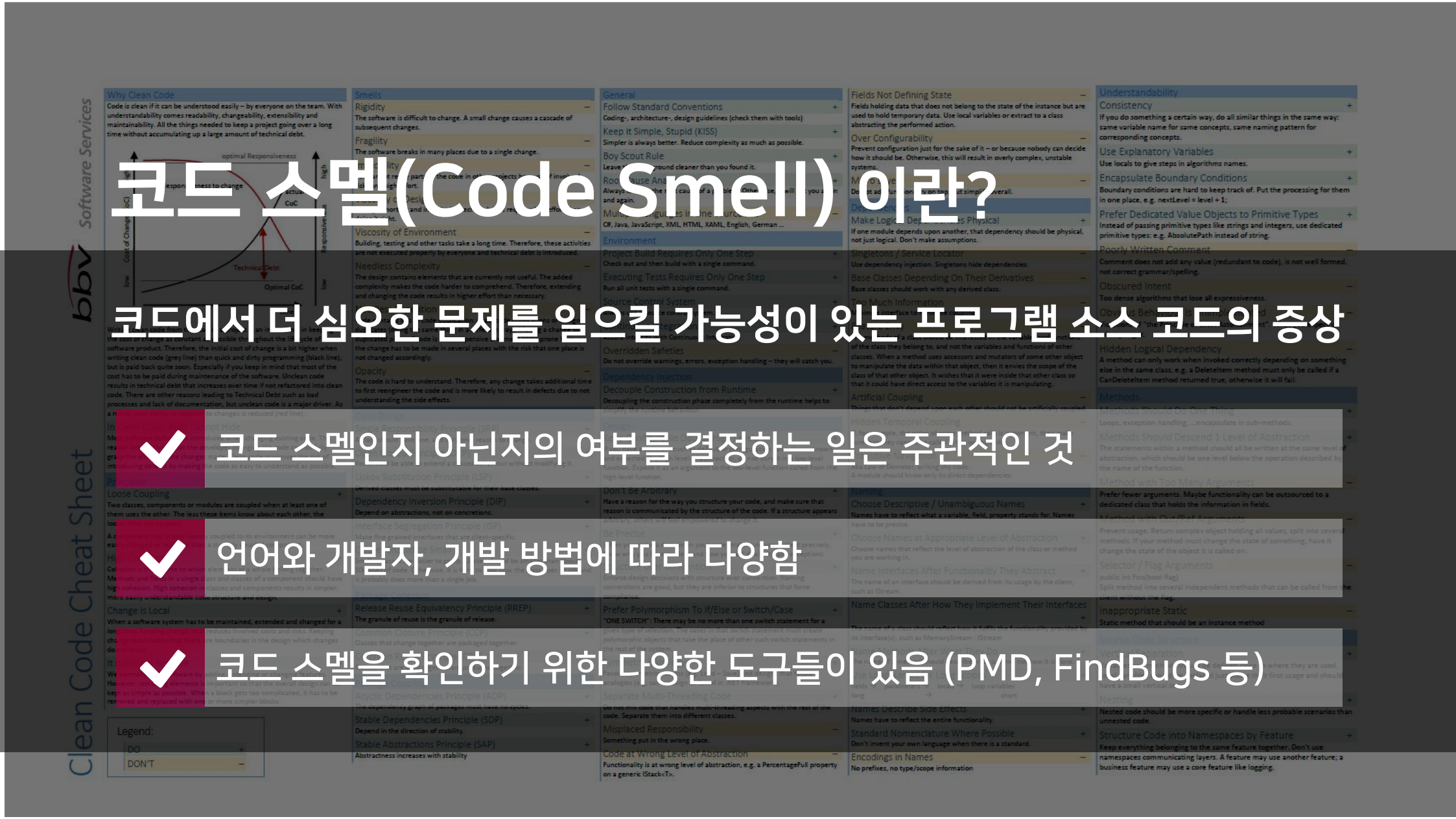
Make the **simplifications**

↓

Run tests to ensure things still work correctly

↓

Repeat the simplify/test cycle until the smell is gone



코드 스멜(Code Smell) 이란?

코드에서 더 심오한 문제를 일으킬 가능성이 있는 프로그램 소스 코드의 증상

- ✔ 코드 스멜인지 아닌지의 여부를 결정하는 일은 주관적인 것
- ✔ 언어와 개발자, 개발 방법에 따라 다양함
- ✔ 코드 스멜을 확인하기 위한 다양한 도구들이 있음 (PMD, FindBugs 등)

Code Smell

Group name	Smells in group
The Bloaters	-Long Method -Large Class -Primitive Obsession -Long Parameter List -Data Clumps
The Object-Orientation Abusers	-Switch Statements -Temporary Field -Refused Bequest -Alternative Classes with Different Interfaces
The Change Preventers	-Divergent Change -Shotgun Surgery -Parallel Inheritance Hierarchies
The Dispensables	-Lazy class -Data class -Duplicate Code -Dead Code -Speculative Generality
The Couplers	-Feature Envy -Inappropriate Intimacy -Message Chains -Middle Man

3

Legacy code 소개

Gilded Rose

시스템소개

Gilded Rose 는 Allison 이 운영하는 작은 여관입니다.
이 여관은 유명한 도시의 요지에 자리잡고 있습니다.
우리는 이 여관에서 finest goods를 사고 팝니다.

상품의 유통기한(sellIn)이 다가올수록 상품들의 품질(quality)은 지속적으로 떨어집니다.

- 모든 아이템에는 유통기한(sellIn)이 있습니다.
  - 유통기한은 아이템을 팔아야 하는 날까지 남아 있는 일 수로 표시합니다.

- 모든 아이템에는 품질값(quality)이 있습니다.
  - 품질값은 아이템이 얼마나 가치가 있는지를 나타냅니다.

- 시스템은 매일 자정에 모든 아이템의 값들을 갱신합니다.

Gilded Rose

Gilded Rose

시스템소개

- 유통기한이 지나면, 품질은 두 배씩 빨리 떨어집니다.
- 아이템의 품질은 음수가 될 수 없습니다.
- 아이템의 품질은 50보다 클 수 없습니다.

- "Aged Brie"는 시간이 지날수록 품질이 증가합니다.

- "Sulfuras"는 전설의 아이템입니다.
- 절대 팔지도 않고 품질이 떨어지지도 않습니다.

- "Backstage passes" 는 유통기한이 다가올수록 품질이 증가합니다.
- 유통기한이 10일 이하일 때 품질은 2씩 증가하고,
- 유통기한이 5일 이하일 때는 3씩 증가합니다.
- 콘서트가 끝나고 유통기한이 지나면 품질은 0이 됩니다.

Gilded Rose

<Aged Brie>

<Backstage Pass>

<Sulfuras>

NORMAL

<Normal>

Gilded Rose

코드를 살펴보아요

- Item
- GildedRose
- GildedRoseTest

Gilded Rose

생각해 보아요

<Aged Brie>

<Backstage Pass>

<Sulfuras>

NORMAL

<Normal>

새로운 아이템이 추가될 경우에 어떤 어려움이 예상될까요?

Gilded Rose

업무별로 분기만 잘 되어 있어도…

분기문이 좀 적었으면…

기존 코드의 동작을 보장하는 TestCase가 있었으면…

4

Test case 작성

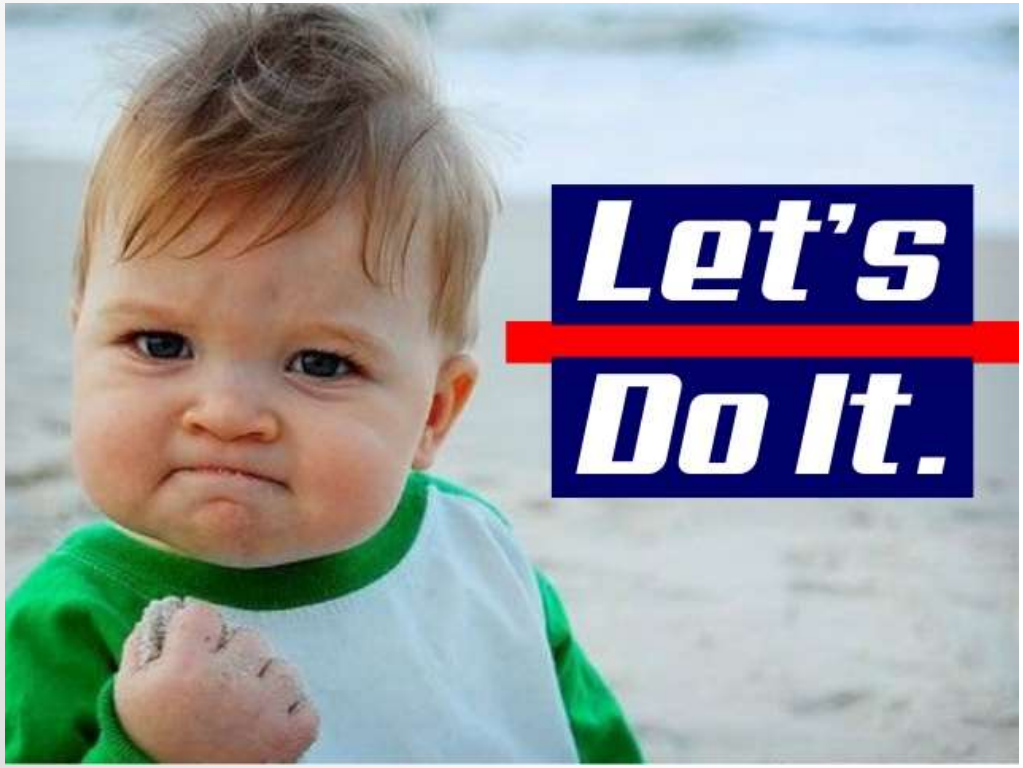
Test Case

Test Case의 효과

1. 수정시의 생산성 향상
2. 버그 잡기가 빨라진다.
3. 버그 잡기가 쉬워진다.
4. 시스템 구조가 좋아진다.
5. 리팩토링의 조건이 된다.
6. 회귀테스트를 제공한다.
7. 하위호환성 보증의 방법을 제공한다.
8. 전체 시스템의 이해 없이 부분의 수정이 가능하다.
9. 샘플로 활용된다.
10. 코드리뷰시에 부담감이 준다.
11. CI가 제대로 활용된다.
12. 설계와 구현을 분리할 수 있다.

Test Case

Coverage를 100% 만족하도록 Test Case 작성



5

Refactoring 체험

Conditional Complexity Refactoring (1/4)

Refactoring	Before	After

Invert-if	If (!A) {
	  B
	} else {
	  C
	}
		If (A) {
		  C
		} else {
		  B
		}

Split Condition	If (A || B) {
	  C
	} else {
	  D
	}
		If (A) {
		  C
		} else if (B) {
		  C
		} else {
		  D
		}

Conditional Complexity Refactoring (2/4)

Refactoring	Before	After

Add else block	If (A) {
	  B
	}
		If (A) {
		  B
		} else {
		}

Join if-statement with inner if-statement	If (A) {
	  if (B) {
	  C
	  }
	}
		If (A && B) {
		  C
		}

Conditional Complexity Refactoring (3/4)

Refactoring	Before	After

Exchange conditions for inner and outer if-statement	If (A) {
	  if (B) {
	  C
	  } else {
	  D
	  }
	}
		If (B) {
		  if (A) {
		  C
		  }
		} else {
		  if (A) {
		  D
		  }
		}

Combine outer else and inner if-statement	If (A) {
	  B
	} else {
	  if (C) {
	  D
	  } else {
	  E
	  }
	}
		If (A) {
		  B
		} else if (C) {
		  D
		} else {
		  E
		}

Conditional Complexity Refactoring (4/4)

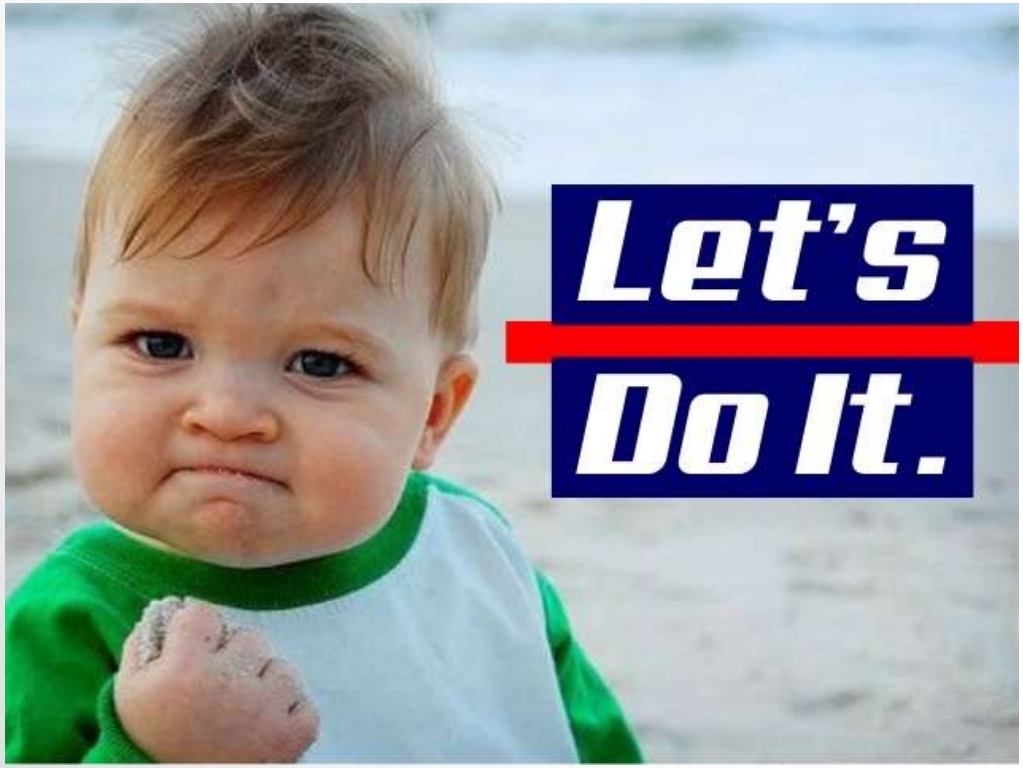
Refactoring	Before	After

Consolidate Duplicate Conditional Fragments	If (A) {
	  B
	  D
	} else {
	  C
	  D
	}
		If (A) {
		  B
		} else {
		  C
		}
		D

Duplicate Statements into if-statement	Z
	
	If (A) {
	  B
	} else {
	  C
	}
	
	D
		If (A) {
		  Z
		  B
		  D
		} else {
		  Z
		  C
		  D
		}

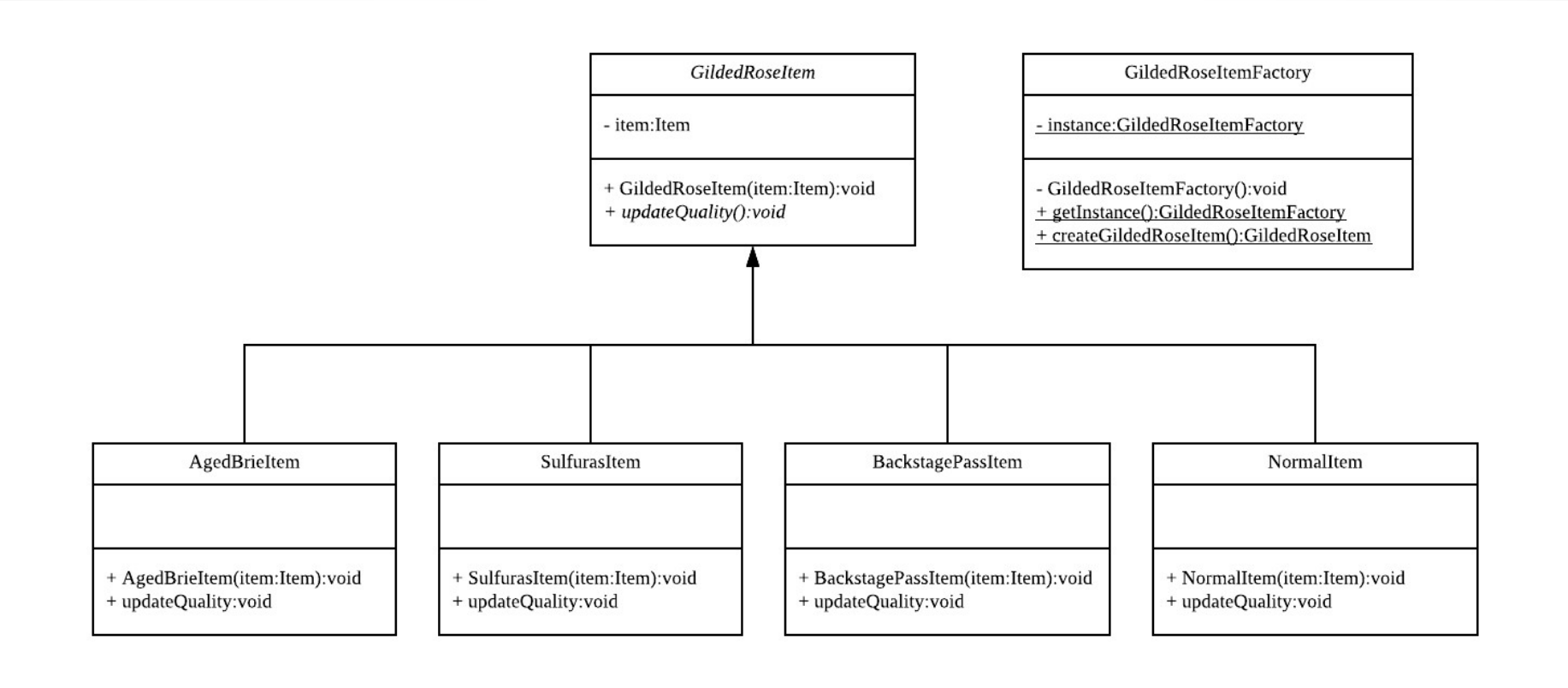
Refactoring

If 문을 정복해 보자



Refactoring

Beyond…



끝맺음

클린코드를 위하여…

- 객체지향언어의 이해(클래스, 추상화, 캡슐화, 상속, 다형성)
- 객체지향언어의 설계원칙(SOLID)
- Code Smell
- 테스트기법(단위테스트와 Mock, 테스트 커버리지)
- IDE 활용
- 디자인 패턴
- 아키텍처 패턴, 코드품질지표

Thank You


Q & A
Partner Disrupt Foresee