00 리팩토링이란

이 장에서는 다음의 순서로 리팩토링의 개요를 배웁니다.

- 리팩토링의 정의
- 리팩토링과 '코드의 악취(Bad Smells in Code)'
- 리팩토링 카탈로그
- 리팩토링의 에센스(essence)
- 리팩토링 Q&A

리팩토링이란

리팩토링의 정의

리팩토링(refactoring)이라는 것은 외부에서 본 프로그램의 동작은 변하지 않고 프로그램 내부의 구조를 개선하는 것입니다. 이 정의의 요점은 다음의 두 가지입니다.

- 리팩토링을 하더라도 외부에서 보이는 프로그램의 동작에는 변함이 없다.
- 리팩토링을 하면 프로그램의 내부의 구조는 개선된다.

예를 한 가지 들어 보겠습니다. 앞으로 6장에서 우리들은 **클래스 추출**이라는 리팩토링을 공부하게 됩니다. 이것은 크고 다루기 힘든 클래스를, 정리되어 있는 필드와 메소드를 뽑아서 새로운 클래스로 만드는 것입니다.

내부적으로 새로운 클래스를 만들어도 외부에서 보이는 프로그램의 동작에는 변함이 없습니다. 하지만 거대한 클래스를 제대로 분할하는 것은 프로그램 내부구조의 개선이라고 할 수 있습니다.

구체적인 리팩토링은 1장 이후에서 자세히 설명하겠지만, 우선은 "외부에서 보는 프로그램의 움직임은 변함없이 프로그램 내부의 구조를 개선한다"라는 정의를 머릿속에 새겨 두기 바랍니다.

리팩토링 퀴즈

앞에서 "리팩토링의 정의"를 읽었습니까?

이제부터 리팩토링의 정의에 관해 이해했는지 퀴즈 형식으로 확인해 보겠습니다.

1. '디버깅'은 리팩토링인가?

질문 : 디버깅은 리팩토링일까요?
대답 : 아닙니다.

디버깅을 하면 외부에서 보이는 프로그램의 동작은 달라집니다. 그러나 리팩토링의 경우에는 외부에서 볼 때 프로그램의 동작이 변하는 것은 아니기 때문에 디버깅은 리팩토링이 아닙니다.

2. '기능 추가'는 리팩토링일까?

질문 : '기능의 추가'는 리팩토링일까요?
대답 : 아닙니다.

기능을 추가하면 외부에서 볼 때 프로그램의 동작이 변하게 됩니다. 리팩토링을 하더라도 외부에서 볼 때 프로그램의 동작은 변하는 것이 아니기 때문에 '기능의 추가' 역시 리팩토링이 아닙니다.

3. '소스코드의 정리'는 리팩토링일까?

질문 : '소스코드의 정리'는 리팩토링일까요?
대답 : 반드시 그렇지는 않습니다.

소스코드를 정리하면 소스코드가 읽기 쉬워집니다. 이것은 분명 개선의 방법 중 하나지만 리팩토링이라고 말할 수는 없습니다. 그 이유는 소스코드를 정리했다는 것만으로도 프로그램의 동작이 달라질 수 있기 때문입니다.

리팩토링은 단순히 소스코드를 정리하는 것은 아닙니다. "반드시 외부에서 보는 프로그램의 움직임에는 변화가 없다"라는 것을 확인해가면서 진행하는 것입니다.

리팩토링과 유닛테스트(unit test)

리팩토링을 하더라도 외부에서 보는 프로그램의 동작에는 변함이 없습니다. 그럼 프로그램 동작이 정말 변화가 없다라는 것은 어떻게 하면 확인할 수 있을까요?

프로그램의 동작이 정말 변하지 않았다는 것을 확인하려면 테스트를 이용합니다. 테스트하는 방법에는 여러 가지 종류가 있는데, 적어도 유닛테스트(unit test)라고 하는 것이 필요합니다. 유닛

테스트는 단일 테스트라고도 합니다. 유닛테스트는 일반적으로 개발자 스스로 실행하는 최소의 테스트로서 클래스나 패키지 하나를 테스트 대상으로 하는 것이 보통입니다.

리팩토링을 할 때에는 그 전후에 테스트를 합니다.

- 리팩토링하기 전에 테스트를 합니다.
- 리팩토링을 합니다.
- 리팩토링한 후 다시 한 번 테스트를 합니다.

이렇게 해서 리팩토링을 하더라도 프로그램 동작에 변화가 없다는 것을 확인하는 것입니다.

또한 리팩토링의 각 단계에서도 테스트를 실시합니다.

모든 동작을 빠짐 없이 체크할 수는 없기 때문에 테스트도 완전한 것은 아닙니다. 하지만 그렇다고 해서 테스트를 준비하지 않는 이유가 되는 것은 아닙니다. 리팩토링은 테스트가 필수라고 생각하는 것이 좋습니다. 만약 리팩토링하려고 생각하는 프로그램에 테스트가 없다면, 먼저 테스트를 만드는 것을 검토합시다.

자바에서 유닛테스트를 실행하는데 가장 많이 이용되는 테스트 프레임워크는 JUnit입니다.

JUnit에 대해서는 "부록 B"를 참고합시다.

리팩토링의 목적

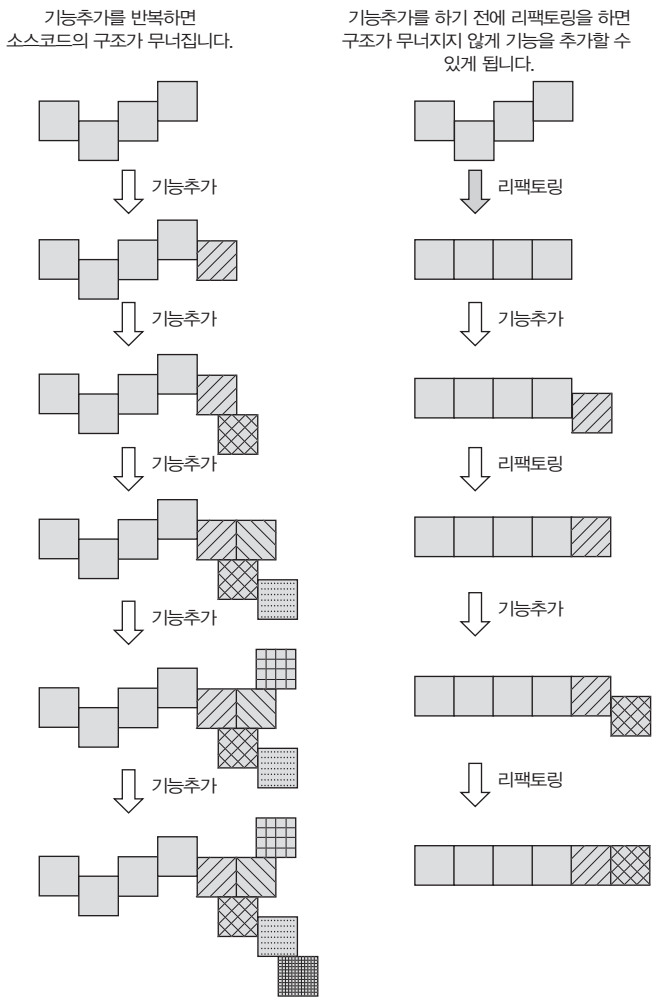
'디버깅'이나 '기능 추가'는 리팩토링이 아닙니다. 그럼 리팩토링의 목적은 무엇일까요?

리팩토링에는 다음과 같은 목적이 있습니다.

버그를 찾아내기 쉽게 한다

'디버깅' 자체는 리팩토링이 아닙니다. 하지만 리팩토링을 하면 프로그램이 정리되어 숨어 있는 버그를 찾아내기 쉬워집니다.

일반적으로 혼란스러운 코드는 디버깅하기 어렵습니다. 디버거로 따라가 보아도 무엇이 어떻게 되어 있는 것인지 파악하기 힘들고, 이 때문에 버그를 찾기 어려워집니다.



[그림 0-1] 기능추가와 리팩토링의 이미지 다이어그램

기능을 추가하기 쉽게 한다

'기능추가' 자체는 리팩토링이 아닙니다. 하지만 리팩토링을 실행하면 프로그램에 새로운 기능을 추가하기 쉬워집니다.

일반적으로 기능을 추가하면 소스코드는 점점 더 복잡해지게 됩니다. 특히 시간에 쫓기는 상황이라면 기능추가는 필요 이상으로 코드가 들어가 구조가 무너진 소스가 됩니다. 이런 상태를 소스코드가 '지저분하다'라고 합니다.

지저분해진 소스코드에 기능을 계속해서 추가하면 소스코드는 점점 더 지저분해져서, 결국 더 이상 기능을 추가할 수 없는 지경에까지 이르게 되어 시간에 더욱 쫓기게 됩니다. 그대로 방치한다면 이렇게 상황이 나빠지는 것이 연속 되기만 할 뿐입니다.

리팩토링을 하면 구조가 무너져 복잡하고 지저분한 코드가 정리되어, 깨끗한 소스코드가 될 것입니다. 결국 기능추가도 편해지겠죠.

리뷰하는 것이 쉬워진다

리팩토링을 하면 코드리뷰가 쉬워집니다.

프로그램의 품질을 올릴 때 여러 기술자가 코드를 읽고 검토하는 코드리뷰는 중요한 역할을 합니다. 리팩토링하여 깨끗해진 코드는 읽기 쉽고 이해하기 쉬워집니다.

리팩토링의 한계

리팩토링은 언제라도 가능한 것은 아닙니다. 리팩토링을 적용할 수 없는 경우도 있습니다.

프로그램이 아직 동작하지 않는 경우

만들고 있는 중이라 아직 동작하지 않는 프로그램은 리팩토링할 수 없습니다. 리팩토링하기 전에 먼저 동작하는 프로그램을 만들어야만 합니다.

또한 설계와 코딩이 좋지 않아 버그투성이인, 사용할 수 없는 프로그램에 대해서도 리팩토링은 할 수 없습니다.

시간이 얼마 남지 않았을 경우(마감일에 가까워 졌을 때)

납기 기간이 매우 엄격한 코드를 리팩토링하는 것은 현명한 방법은 아닙니다.

모름지기 리팩토링이라는 것은 시간이 지나면서 차차 효과가 나타나는 것입니다. 또한 납기 직전에 커다란 리팩토링은 하지 않도록 합시다.

리팩토링과 코드의 악취

코드의 악취란

리팩토링은 외부에서 본 프로그램의 동작은 변함없이 내부구조를 개선하는 것입니다. 그럼 구체적으로 프로그램의 '어느 부분'을 개선하면 좋을까요?

프로그램에서 리팩토링이 필요한 부분을 '코드의 악취(Bad Smells in Code)'라고 부릅니다.

이것은 마틴 파울러의 책 『Refactoring』에 쓰여져 있는 감각적이고 좋은 비유입니다.

'코드의 악취'라는 것은 프로그램 내에서 다음과 같은 부분을 비유하는 말합니다.

- 이해하기 어렵다.
- 수정하기 어렵다.
- 확장이 어렵다.

여러분이 코드를 읽고 있는데 "왠지 이해하기 어렵다"라고 느낀다면 그것은 코드 속에 악취가 숨어있는 경우일 것입니다. 다시 말해 그 부분을 '리팩토링해야 하는 곳이다'라고 생각하는 것이 좋습니다.

마틴 파울러의 『Refactoring』에는 [표 0-1]같은 '코드의 악취(Bad Smells in Code)'가 22가지 소개되어 있습니다.

[표 0-1] 코드의 악취(Bad Smells in Code)

{중복된 코드} (Duplicated Code)	코드가 여기저기 겹쳐 있다.
{너무 긴 메소드} (Long Method)	메소드가 너무 길다.
{거대한 클래스} (Large Class)	클래스의 파일이나 메소드가 너무 많다.
{너무 많은 인수} (Long Parameter List)	메소드에 전달하는 인수의 수가 너무 많다.
{변경의 발산} (Divergent Change)	사양변경이 발생한 경우 수정할 곳이 여기저기 흩어져 있다.

{변경의 분산} (Shotgun Surgery)	어떤 클래스를 수정하면 다른 클래스도 수정하지 않으면 안 된다.
{속성, 조작의 부적절한 관계} (Feature Envy)	언제나 다른 클래스의 속성을 건드리고 있다.
{데이터 덩어리} Data Clump	정리해서 다룰 수밖에 없는 여러 개의 데이터가 하나의 클래스에 정리되어 있지 않다.
{기본 데이터형의 집착} (Primitive Obsession)	클래스를 만들지 않고 int같은 기본 데이터형만을 사용한다.
{switch문} (Switch Statements)	switch문이나 if문을 사용하여 동작을 분할하고 있다.
{평행 상속 구조} (Parallel Inheritance Hierarchies)	서브클래스를 만들면 클래스 계층에 따로 서브클래스를 만들어야 한다.
{게으름뱅이 클래스} (Lazy Class)	클래스가 별로 하는 일이 없다.
{추측성 일반화} (Speculative Generality)	언젠가 이렇게 확장하겠지 하고 기대하는 지나친 일반화
{일시적 속성} (Temporary Field)	일시적으로 사용할 필드가 있다.
{메시지의 연쇄} (Message Chains)	메소드가 호출하는 연쇄가 너무 많다.
{중개자} (Middle Man)	위양(권리를 위임하고)자신이 하는 일은 없는 클래스가 있다.
{부적절한 관계} (Inappropriate Intimacy)	필요 없는 쌍방향 링크가 걸려 있거나 IS–A 관계가 아니면서 상속을 사용한다.
{클래스의 인터페이스 불일치} (Alternative Classes with Different Interface)	API가 부적절하다.
{미숙한 클래스 라이브러리} (Incomplete Library Class)	기존의 클래스라이브러리가 사용하기 힘들다.
{데이터 클래스} (Data Class)	필드와 getter 메소드와 setter 메소드만 가지고 있는 클래스가 있다.
{상속거부} (Refused Bequest)	상속하고 있는 메소드면서 그것을 호출하면 문제가 발생한다.
{코멘트} (Comments)	코드의 부족을 보충하기 위해 상세한 코멘트가 있다.

이 책에서는 코드의 악취를 {중복된 코드}처럼 { }로 묶고 있습니다. 또한 리팩토링을 **메소드의 추출**처럼 볼드로 표기합니다.

코드의 악취를 나타내는 표현

[표 0-1]에서 본 '코드의 악취'는 중요하지만 처음부터 22개를 모두 외우는 것은 무리라고 생각합니다. 여기서 '코드의 악취를 표현하는 대사'를 6개 소개합니다. 소스코드를 읽다가 이 대사가 입에 붙는다면, 붙이고 싶어진다면 '리팩토링이 필요하겠구나!'라고 생각해 주세요.

- 겹쳐 있다!
- 너무 길다!
- 너무 많다!
- 이름이 어울리지 않는다!
- 너무 공개했다!
- 객체지향적이지 않다!

겹쳐 있다!

"어? 여기 코드랑 저쪽 코드가 닮았잖아, 겹쳐 있다!"라는 것은 {중복된 코드}라는 [Bad Smells in Code]의 특징입니다.

겹쳐 있다는 것은 복수의 장소에 비슷한 코드가 산재해 있는 상태입니다. 프로그램을 복사하고 붙여넣기해서 만들면 중복된 코드가 여러 곳에 존재할 것입니다.

겹쳐 있는 코드는 수정하기 어렵습니다. 발견한 버그를 잡을 때도 여러 곳을 고쳐야 하기 때문입니다.

겹쳐 있는 코드를 발견하면 모음을 찾아내서 **메소드의 추출**이나 **클래스 추출**이라는 리팩토링을 검토하는 것이 좋습니다.

null 체크가 여기저기 발견되면 **NULL 오브젝트 도입**이라는 리팩토링이 적당합니다.

또한 오류 체크가 많을 때는 **오류 코드를 예외로 치환하기**를 검토해 봅시다.

너무 길다!

"이 메소드, 도대체 몇 줄인거야? 너무 길다!"라는 것은 {너무 긴 메소드}라는 [Bad Smells in Code]의 특징입니다.

메소드가 너무 길면 이해하는 것이 어렵습니다. "이 메소드에서 하고 있는 것은 OO이다"라고 한 마디로 표현하지 못하고 '이 메소드에서는 OO랑 XX이랑…'과 같이 설명까지 길어집니다.

{너무 긴 메소드}를 발견하면 **메소드 추출**이라는 리팩토링을 하는 것이 좋습니다. 코드의 정리를 찾아서 새로운 메소드로 하는 리팩토링입니다.

또한 혹시 여러분의 프로그램을 읽다가 "이 클래스 도대체 메소드가 몇 개인거야. 너무 크다!"라고 느낀다면 그것은 {거대한 클래스}라는 'Bad Smells in Code'입니다.

클래스가 너무 크다는 것은 그 클래스가 담당하고 있는 책임이 너무 많다는 것입니다.

이런 느낌이 든다면 책임들을 클래스로 빼내는 **클래스의 추출**이라고 하는 리팩토링을 검토해야 할 것입니다.

너무 길다면 짧게 합시다. 너무 크면 작게 합시다. 이러한 것이 리팩토링의 기본입니다.

너무 많다!

클래스가 너무 크기 때문에라고 하더라도 **클래스의 추출**을 지나치게 많이 하면 안됩니다. 왜냐하면 **클래스의 추출**을 지나치게 하면 이번에는 클래스가 너무 많아져서 "이 패키지 도대체 몇 개의 클래스가 있는 거야? 너무 많다!"라는 상태가 되기 때문입니다. 아무리 책임을 줄이는 것이 좋다고 하더라도 일에는 정도가 있는 것입니다.

커다란 클래스는 이해하기 어렵지만 클래스의 수가 너무 많은 것도 이해하는데 지장을 줍니다. 클래스가 너무 많다고 느껴진다면 **중개자의 삭제**라는 리팩토링이나 **클래스의 인라인화** 또는 **메소드의 인라인화** 등의 리팩토링을 검토합시다.

이름이 어울리지 않는다!

"이 setItem이라는 메소드, 아무것도 세팅되어 있지 않네. 이름이 어울리지 않는다!"

이름은 중요합니다. 프로그램에서는 많은 이름들이 등장합니다. 변수명, 필드명, 메소드명, 클래

스명, 패키지명……. 이들의 이름은 프로그램을 읽는 사람에게 적절한 정보를 전달하는 역할을 합니다. 그러므로 코드를 쓸 때 프로그래머는 적절한 이름을 붙여주는 것에 주의를 기울여야 합니다.

또한 이름을 한 번 붙인다고 그것으로 끝이 아닙니다. 프로그램은 살아 움직이는 것입니다. 수정을 하다 보면 처음 붙였던 이름이 부적절한 경우가 많을 것입니다.

표현하고 싶은 개념과 이름이 어울리지 않는다는 것을 발견하면 영향을 끼치는 범위를 최소한으로 하고 이름을 변경합니다.

처리의 정리에 이름을 붙이는 **메소드의 추출**이나 메소드명을 적절히 변경하는 **메소드명의 변경**은 대표적인 리팩토링입니다. 또한 이해하기 힘든 식에 이름을 붙이는 **설명용 변수의 도입**, 같은 변수를 다시 사용하는 것을 막는 **임시변수의 분리**도 관련된 리팩토링입니다.

지나치게 공개했다

"이 메소드, public으로 되어 있어도 괜찮을까? 지나치게 공개했다!"

메소드가 private로 되어 있으면 외부 클래스에서 메소드를 호출할 수 없습니다. 하지만 호출할 수 없어 불편하다고 모두 public으로 선언하는 것은 잘못된 것입니다. 그것은 클래스의 구현을 너무 공개하는 것입니다. public으로 선언된 메소드는 다른 것에서 호출될 가능성이 있습니다. 프로그램을 수정하려고 생각했을 때 "이 메소드 누군가 사용하고 있을지도 모르겠네. 지우면 난처 하려나"라고 고민하게 될 것입니다.

메소드만이 아니고 필드 역시 너무 많이 공개해서는 안 됩니다. 필드의 캡슐화라는 리팩토링으로 필드를 감춥시다.

클래스의 이름 역시 너무 많이 공개해서는 안 됩니다. new를 사용해서 인스턴스를 생성할 때 new 다음에는 구체적인 클래스명을 씁니다. 이 경우 클래스명을 변경하는 것은 어렵습니다. 왜냐하면 프로그램에서 인스턴스를 new로 하고 있는 부분을 모두 변경해야 하기 때문입니다. 이런 경우에는 **생성자를 Factory Method로 치환하기**라고 하는 리팩토링을 검토합시다. Factory Mothod로 클래스의 이름을 숨기는 것입니다.

"클래스 간의 관계도 너무 공개했다!"라고 말하고 싶어질 때가 있습니다. 자세한 것은 14장 **위임의 은폐**에서 소개합니다.

'100'같은 구체적인 값(매직넘버)이 보이는 것도 좋지 않은 것입니다. 이것에 관해서는 1장 **매**

직넘버를 심볼릭 정수로 치환하기에서 이야기합시다.

적절히 정보를 숨기는 것을 일반적으로 정보은폐(information hiding)라고 합니다. 정보은폐는 대단히 중요한 것입니다.

객체지향적이지 않다

자. 'Bad Smells in Code'의 마지막입니다.

"프로그램의 여기저기서 instanceof를 사용하고 있다. 객체지향적이지 않다!"

객체지향적이지 않다라는 것은 이해하기 어려울지도 모르지만, 예를 들어 다음과 같은 상황입니다.

- switch문이나 if문을 사용해서 처리를 분기만 한다.
- instanceof를 사용해서 오브젝트가 속해있는 클래스를 찾기만 한다.
- int만 너무 많이 사용하고, 전용 클래스를 만들지 않았다.

switch문이나 if문을 사용해서 다른 동작으로 분기시키는 것은 좋지 않습니다. 이 코드의 악취는 [표 0-1]의 {switch문}이라는 것입니다. switch문이나 if문 대신에 다형성을 사용하는 것을 검토합시다. 다형성을 잘 사용하면 switch문이나 if문을 줄이고 심플한 메소드를 호출하는 것이 가능합니다.

객체지향에서는 오브젝트 자신이 자신의 동작을 알고 있습니다. 그러므로 instanceof를 사용해서 "이 오브젝트는 어떤 클래스의 인스턴트였더라?"라고 일일이 조사할 필요는 없는 것이 보통입니다. 만약 instanceof를 많이 사용하고 있다면 그것은 객체지향적이지 않은 코드가 되는 것입니다.

익숙하다고 해서 int만을 사용하는 것은 문제가 있습니다. 언어에 준비되어 있는 형체크가 활용되지 못하고 생각지도 못한 실수를 낳을 위험성이 있기 때문입니다. 그것은 {기본 데이터형에의 집착}이라고 불리는 코드의 악취입니다.

이렇듯 객체지향적이지 않은 상황을 발견한다면 **타입코드를 클래스로 치환하기** 또는 **타입코드를 서브클래스로 치환하기** 등의 리팩토링을 검토합시다.

여기서 중요한 주의점 한 마디. '객체지향적이지 않은 코드'가 있으면 절대 리팩토링하지 않으면 안 된다라고 단정해서는 안됩니다. 객체지향적이지 않은 코드는 어디까지나 "여기는 리팩토링하는 것이 좋지 않을까?"라고 검토하는 것에 지나지 않는다는 것입니다. 어찌되었든 간에 리팩토

링을 하고 싶어하는 것은 리팩토링을 배운 사람들이 빠져드는 함정이므로 주의하기 바랍니다.

리팩토링 카탈로그

리팩토링 카탈로그란

지금까지 **메소드의 추출**이나 **클래스의 추출**과 같은 리팩토링의 이름이 몇 개 등장했습니다. 이 책에서는 리팩토링의 이름을 모두 볼드로 나타내고 있습니다.

리팩토링에는 여러 가지 종류가 있습니다. 마틴 파울러의 『Refactoring』에서는 각 리팩토링의 목적이나 순서를 카탈로그화 하여 정리하고 있습니다. 이 카탈로그를 리팩토링 카탈로그라고 합니다. 또한 웹페이지 http;//www.refactoring.com/에는 100개 가까이의 리팩토링이 소개되어 있습니다.

이 책에서는 마틴 파울러의 『Refactoring』에 게재되어 있는 리팩토링의 에센스만을 뽑아 "부록 A"에 정리해 두었습니다. 이 부록을 참고하면 어떤 리팩토링들이 있는지 짧은 시간에 파악할 수 있을 것입니다.

소개되어 있는 리팩토링을 모두 외워야만 하는 것은 아닙니다. 먼저 여러분이 알고 있는 것부터 적용해 나가면 되는 것입니다.

또한 이 책에서 기본적인 리팩토링의 코치를 받으면 다른 리팩토링을 배우는 것이 편해질 것입니다. 이제 곧 여러분이 스스로 유익한 리팩토링을 만들어 낼 수도 있을 것입니다.

조직적 수정

리팩토링에서는 맘이 닿는 대로 프로그램을 수정해 나가는 것은 아닙니다. 카탈로그에 따라서 조직적으로 코드를 변경합니다. 실제로 숙달된 프로그래머에게는 재미없을만큼 코드를 변경하는 순서가 잘 정리되어 있습니다.

하지만 이 순서는 프로그래머의 실수를 가능한 한 줄이고 실수가 있을 경우에는 가능한 한 빨리 검출할 수 있게 고안된 것입니다.

만약 "뭐야, 이런 거라면 나도 매일 하고 있어!"라고 느끼는 독자가 있을지도 모르만 그것은 당

연합니다. 리팩토링은 신기한 기법이 아니라 많은 설계자와 프로그래머들의 경험을 정리한 것이기 때문입니다.

리팩토링의 에센스

다음 장부터 우리는 구체적인 리팩토링을 배워나갈 것입니다. 자세한 이야기에 들어가기 전에 모든 리팩토링의 근원인 리팩토링의 에센스를 소개합니다. 꼭 마음 속에 새겨두기 바랍니다. 그것은 다음과 같이 한 마디로 표현할 수 있습니다.

Step by Step

이것은 '한 걸음 한 걸음'의 의미입니다. 리팩토링에 관해서는 '한 번에 한 개씩'이라고 생각하면 되겠습니다. 이 말은 이 책의 전반에 걸쳐 가장 중요한 단어입니다.

다음의 스텝 바이 스텝의 의미에 관해 생각해 봅시다.

Step by Step: 한 번에 2개씩 수정하지 않는다

우리들은 눈앞에 있는 코드에 매혹되어 한꺼번에 여러 개를 수정하려고 합니다. 예를 들어 어떤 클래스에서 다른 클래스로 메소드를 이동하는 중에 "아무래도, 이 메소드의 이름은 좋지 않는 것 같다"라고 생각해 메소드의 이름을 변경한 적이 있을 것입니다.

하지만 이것은 스텝 바이 스텝이 아닙니다. 메소드의 이동을 완료하고 컴파일러와 테스트가 끝난 다음 "자, 그럼 메소드명을 고치는 것으로 다음 단계를 진행하자"라는 것이 올바른 태도입니다.

A와 B의 작업을 할 때에는 다음과 같이 2종류의 작업을 혼재시면 안됩니다.

A1→B1→A2→A3→B2→B3→A4→B4

그렇게 되면 실패할 확률이 매우 높아집니다. 올바른 진행 방법은 먼저 다음과 같이 A 작업을 모두 끝냅니다.

A1→A2→A3→A4→작업 A 확인

그런 다음 B 작업을 합니다.

B1→B2→B3→B4→작업 B 확인

이렇게 하나씩 해결하라는 말을 들으면 "나는 이 정도로 간단한 프로그램은 절대 실패하지 않습니다"라고 말하고 싶어질 것입니다. 필자도 마찬가지입니다. 그리고 몇 번의 실패를 했습니다. 실패까지는 아니더라도 2개의 수정이 뒤죽박죽 되어 헛된 시간을 보내버린 적도 몇 번이나 있습니다.

리팩토링은 여러 개의 작업을 동시에 할 수 있는 힘을 자만하기 위한 것이 아닙니다. 작더라도 확실한 한 걸음을 반복해서 개선하는 기술인 것입니다. 즉 스텝 바이 스텝입니다.

Step by Step: 후에 돌아오기 쉽게

한 단계씩 진행하는 가장 큰 목적은 '마지막 스텝을 되돌리는 것'이 편해지기 때문입니다. 리팩토링을 하더라도 문제가 발생하여 수정 전으로 되돌리고 싶을 때가 있습니다. 그럴 때 step by step으로 진행한다면 돌아오는 것이 편해집니다. 진행하는 것과 역순으로 Step by Step으로 돌아가면 되기 때문입니다.

작업 A만 지우고 싶은데 작업 A랑 작업 B의 결과가 소스코드 안에 혼재하고 있다면 어떨까요. 작업 B에는 손을 대지 않고 작업 A만 되돌아오게 하는 것은 매우 귀찮은 작업입니다.

Step by Step: 단계마다 확인

한 단계씩 작업을 할 때에는 각 스텝을 올바르게 진행했는지 반드시 확인해야 합니다. 그 때 가능하면 컴퓨터를 사용해서 확인하는 것이 좋습니다.

예를 들어 새로운 클래스를 만들었다고 합시다. 그 클래스를 누구도 사용하지 않더라도 만든 단계에서 컴파일해 둡시다. 그렇게 하면 문법적으로 문제 없는 것을 확인할 수 있습니다. 이것은 각 스텝을 하나씩 모두 확인한 예입니다. 컴파일하여 확인한 것입니다.

다음으로 지금 만든 클래스를 프로그램의 안에 넣는다고 합시다. 이번에는 컴파일하는 것만이 아니고 테스트도 합시다. 그렇게 하면 그 클래스를 넣더라도 프로그램의 동작에는 변함이 없다는 것을 확인할 수 있을 것입니다. 이것 역시 '스텝마다 확인'하는 것의 한 예입니다. 이번에는 컴

파일러와 테스트의 도움으로 확인한 것이 됩니다.

리팩토링 카탈롤그의 안에서는 자주 이런 표현이 나옵니다.

- 컴파일하기
- 컴파일해서 테스트하기

이것은 틀림없이 스텝마다의 확인이 중요하다는 것을 말하는 것입니다.

step by step: 오래된 것을 새로운 것으로 바꿔가자

우리들은 갑자기 모든 것을 한 번에 변경하고 싶어집니다. 하지만 하나씩 바꿔가는 것이 안전합니다.

"예전 것을 뒤엎고 모두 다시 고치자"라는 것이 아니고 "움직이는 상태를 보존한 채로 새로운 코드를 추가해서, 오래된 것이 모두 새롭게 되었을 때에 예전 것을 없애자"라는 순서를 밟는 것입니다. 처음에는 귀찮아 보이지만 이렇게 하는 것이 실패 위험이 적어집니다.

작은 강아지 집을 짓는 것이라면 모두 망가뜨리고 새로 지으면 되겠지만 큰 건물을 지으려고 한다면 뼈대를 튼튼히 하고 건물을 짓고 그 다음에 뼈대를 빼내는 것입니다.

자. 스텝 바이 스텝으로 진행하는 것의 중요성이 이해가 되셨는지요?

이 스텝 바이 스텝에 의해 리팩토링의 에센스가 여러분의 마음 속에 남는 것이 필자의 바램입니다.

리팩토링 Q&A

Q&A 형식으로 리팩토링을 공부해 봅시다.

리팩토링은 만병통치약인가

Q. 리팩토링은 소프트웨어 문제를 모두 해결합니까?

A. 아닙니다.

리팩토링은 소프트웨어의 체질개선을 의미하는 것이지만, 모든 문제를 해결하는 만병통치약은 아닙니다. 사실 리팩토링자체는 버그를 잡거나 기능을 늘리거나 하는 것은 아니며 움직이지 않는 코드를 움직이게 해주는 것도 아닙니다.

리팩토링은 내부의 구조를 개선하고 버그를 잡기 쉽게 하고 기능을 추가하기 쉽게 하며 읽기 쉬운 코드를 만드는 것뿐입니다.

리팩토링을 배우는 것은 가치가 있는가

Q. 리팩토링은 당연한 것만을 진술하고 있습니다. 명백하게 공부할 가치가 있는 것입니까?

A. 예, 그렇습니다.

경험을 쌓은 프로그래머는 소개되고 있는 리팩토링을 '당연하다'라고 느끼는 경우가 많은 듯합니다. 이것은 당연합니다. 왜냐하면 리팩토링은 경험을 쌓은 프로그래머가 무의식 중에 하고 있는 것을 의식적으로, 그리고 확실히 하기 위한 것이기 때문입니다.

경험을 쌓은 프로그래머가 리팩토링을 배우면 자신의 경험을 재정리, 다시 확인하게 됩니다. 또한 신인 프로그래머가 리팩토링을 배우면 앞서 경험한 프로그래머들의 지식을 효율적으로 몸에 익힐 수 있는 것입니다.

움직이는 코드에 손을 대는 것은 좋은 것인가

Q. 움직이고 있는 코드에 손을 대야만 하는 것은 아니라고 생각합니다. 움직이고 있는 코드를 바꾸는 리팩토링은 문제는 없습니까?

A. 아니오.

이야기를 정리해봅시다.

분명 현재 움직이고 있는 코드를 이후 전혀 수정할 것이 아니라면 리팩토링을 할 가치는 없습니다.

하지만 실제로는 그렇지 않습니다. 오랜 기간에 걸쳐서 버그를 수정하고, 기능을 추가해야 하는 것이 소프트웨어의 현실입니다. 리팩토링은 버그를 수정하거나 기능을 추가하기 쉽게 하기 위해 내부의 구조를 개선하는 것입니다.

움직이고 있는 코드에 손을 대면 안 되는 것은 새로운 버그를 낳기 때문입니다. 리팩토링을 할 때

는 새로운 버그를 낳는 위험성을 줄이기 위해 유닛테스트를 준비하는 것이 일반적입니다.

단, 릴리스 직전에 타이밍이나 기간적으로 좋지 않은 상황에서는 리팩토링을 해야만 하는 것은 아닙니다.

어디까지가 '외부'인가

Q. 리팩토링의 정의에 "외부에서 보는 프로그램의 동작은 변함없이 내부의 구조를 개선한다"라는 표현이 있지만 프로그램의 어디까지가 외부이며, 어디가 내부인 것일까요?

A. 경우에 따라 다릅니다.

메소드의 본체에 쓰여있는 처리내용을 리팩토링하려고 생각한다면 그 메소드의 본체가 '내부'가 되며 그 메소드를 부르는 부분이 '외부'가 되는 것입니다. 이런 경우 "외부에서 본 동작을 바꾸지 않는다"라는 표현은 "메소드의 처리내용을 바꾸더라도 메소드를 호출하고 있는 측에서는 영향을 미치지 않는다"라고 바꾸어 말할 수 있는 것입니다.

클래스 A의 안에 선언된 여러 개의 메소드를 리팩토링 대상으로 한다면 클래스 A가 '내부'가 되고, 클래스 A를 이용하고 있는 다른 클래스 B,C,D,...가 '외부'가 될 것입니다. 리팩토링의 결과 클래스 A의 내부에서만 사용되고 있는 몇 개인가의 메소드의 동작이 변할지도 모릅니다. 하지만 외부의 클래스 B,C,D,...에서 호출되고 있는 메소드마저 동작을 바꾸지 않는다면 '외부에서 본 동작'은 변하지 않는 것이 됩니다.

리팩토링에는 여러 개의 클래스가 영향을 받는 경우도 있습니다. 이런 경우도 생각하는 방법은 같습니다. 리팩토링의 대상범위가 '내부'가 되고 이용자측이 '외부'가 되는 것입니다.

이렇게 생각해보면 리팩토링에 관한 테스트의 역할이 보다 명확해집니다. 리팩토링에 관한 테스트는 그 리팩토링에 의해 영향을 받는 부분과 영향을 받지 않는 부분을 명시하고 있습니다. 테스트는 "이 리팩토링에 의한 영향은, 그 범위의 외부에는 끼치지 않는다"라고 주장하는 것입니다.

초기 설계를 제대로 해두는 것이 유효하지 않을까?

Q. 리팩토링이라 해서 코드를 조금씩 고치는 것보다 개발의 상류공정에서 제대로 초기설계를 하는 편이 중요하지 않을까요?

A. 초기설계는 분명 중요한 것이지만 리팩토링은 설계를 개선하는 한 부분이기도 합니다.

리팩토링은 '코드의 개선'이라는 시점에서 설명되는 것이 많지만 그것을 이해하기 쉽게 하기 위해 말을 단순화시킨 것입니다. 실제 리팩토링은 '설계의 개선'이라고 할 수도 있는 것입니다.

소프트웨어는 우리들이 상상하는 것 이상으로 복잡합니다. 설계의 단계에서 보이고 있는 것만으로 한정 지을 수는 없습니다. 구현하는 단계에서 초기에 설계할 때는 생각하지 못한 상황이 발견되는 경우도 자주 있습니다.

특히 소프트웨어에 대한 요구는 자주 변하는 것에도 주목합시다. 소프트웨어는 살아 있습니다.

초기설계를 가능한 한 바꾸지 않도록 기능추가를 하면 그것은 다음에 자연스럽지 않은 코드가 생겨 추가기능에도 시간이 걸리게 될 것입니다. 초기 설계를 절대 바뀌지 않도록 하면 오히려 고객의 요구에 대응하지 못할 가능성도 있습니다.

리팩토링에서 클래스의 관계를 조절해서 가는 것은 초기설계를 조금씩 현실의 요구에 가깝게 가는 것입니다. 즉 이것은 '설계의 개선'인 것입니다.

적절한 리팩토링을 발견하는 것은

Q. 어떻게 하면 적절한 리팩토링을 찾는 것이 가능할까요?
A. 우선 '코드의 악취'를 느끼는 것입니다.

리팩토링은 닥치는 대로 적용하는 것이 아닙니다. 코드 속에서 악취를 느끼고 숨겨져 있는 문제점을 발견해 내는 것입니다.

해결해야만 하는 문제를 확실히 한 다음부터 적절한 리팩토링을 찾는 것이 좋습니다.

모두 기억할 필요가 있는가

Q. 리팩토링 카탈로그를 모두 외울 필요가 있습니까?
A. 처음부터 모두 기억할 필요는 없습니다.

우선 이 책을 읽고 기본적인 리팩토링의 감을 잡도록 합시다. 그런 후에 참고 문서를 살피면서 서적이나 웹사이트에 보다 넓고 깊은 공부를 하는 것이 좋을 것입니다.

익스트림 프로그래밍과 관계가 있는가

Q. 리팩토링은 엑스트림 프로그래밍(XP)와 관련이 있습니까?

A. 예

리팩토링은 익스트림 프로그래밍(XP)과 깊은 관련이 있습니다.

실제 XP가 제공하고 있는 연습의 하나가 '리팩토링'이 포함되어 있습니다. 그것은 리팩토링이 가지고 있는 "현실에 맞춰 조금씩 설계해 간다"라는 성질과 관련되어 있는 것입니다.

디자인 패턴과 관계가 있는가

Q. 리팩토링은 디자인 패턴과 관계가 있습니까?

A. 예, '디자인 패턴은 리팩토링의 이정표'라고 불립니다.

디자인 패턴을 점이라고 한다면 리팩토링은 점과 점을 연결하는 화살표와 같은 것입니다. 현재의 코드를 보다 좋은 코드로 변화시켜가는 것이 리팩토링이고, 리팩토링이 향하는 '보다 좋은 코드의 본연의 자세'가 디자인 패턴이라고 생각해도 좋을 듯합니다. 디자인 패턴과 리팩토링의 관계에 대해서는 『패턴지향 리팩토링 입문』(케리에브스키)에서 자세히 설명하고 있습니다.

지금의 현장 업무에서 리팩토링은 어렵다?

Q. 개선에는 관심이 있습니다. 하지만 예산과 시간을 생각하면 지금의 현장에서 리팩토링을 도입하는 것이 어려운데 어떻게 하면 좋을까요?

A. 모두를 해결할 수 있는 방법은 없지만, 제안 하나를 해 보겠습니다.

리팩토링을 위해 예산과 시간을 새롭게 확보하는 것이 어려운 경우가 많습니다. 또한 많은 사람들이 함께 하는 프로젝트에서는 더더욱 그러할 것입니다.

하지만 리팩토링을 단계적으로 도입하는 것은 어떻습니까?

프로그램 전제에 리팩토링을 도입하기 전에 적은 수의 사람만으로 리팩토링을 시험적으로 도입해 보는 것입니다. 자신이 쓴 범위에서 '코드의 악취'가 생기면 그것을 바로 리패토링하는 것입

니다(테스트도 잊지 말길 바랍니다). 전체적으로 리팩토링의 공정을 준비하는 것이 아니라 코딩의 일부로서 하는 것이 포인트입니다. 이것은 책상 주위가 어질러져 있다면 정리하는 것과 같은 발상입니다.

적은 수의 사람으로 실행 가능한 리팩토링은 작은 것밖에 없을지도 모르지만, 더욱 큰 개선이 가능할지도 모릅니다. 그렇지만 그렇게 해서 리팩토링을 실제 프로젝트에 실행해보면 다음 스텝으로 진행할 수 있는 계기를 만드는 것이 아닐까요.

리팩토링은 단계적으로 코드를 수정합니다. 리팩토링도 단계적으로 도입하는 것이 좋겠죠!

연습문제

1. 다음의 문장에서 올바른 것에 O, 잘못된 것에는 X를 하세요. [기초지식 확인]

(1) 버그를 수정하는 것은 리팩토링이 아닙니다.
(2) 리팩토링을 하면 외부에서 본 프로그램의 동작이 개선됩니다.
(3) 올바르게 리팩토링 되었는지 확인하려면 리팩토링이 끝난다음 테스트를 만듭니다.