< 2023.08.29 >

**< 1 교시 > 좋은 프로그램**

코드를 오류 없이 짜면서도 깔끔,안정적으로 짤 수 있는 기법

// 1. 방어적 코딩 – Validation

if(svcListOutDto != null

&& vcListOutDto.get(0) != null) {

outDto.setInstCd(svcListOutDto.get(0).getInstCd());}

우리는 여기서 점 하나를 찍는 효과에 대해서 충분히 고려해 보아야 한다

깨진 창문이론을 기억하자.

// 2. 방어적 코딩 - Assertion

if(svcListOutDto == null ) {

throw new BizApplicationException

("svcListOutDto 값이 Null이 들어왔어요!!");

}

if(svcListOutDto.get(0) == null ) {

throw new BizApplicationException

("svcListOutDto.get(0) 값이 Null이 들어왔어요!!");

}

- 프로그램에 에러가 있어요 : Error , 버그보다 무거운 상황의 느낌. Exception 예상보다 훨씬 큰값이 들어오거나, 메모리가 부족하거나, 코드적인 결함이 있어서 정상적으로 실행되지 않음.

- 프로그램에 결함이 있어요 : 요구사항을 만족시키지 못함. 1+1=3 이 나오게 해주세요 요구사항이 있다면 3이 아니라면 결함이 된다.

=======================================================

**< 2 교시 > 코드 리뷰 기법**

- 정적 프로그램 분석

예) PMD

PMD는 잘못된 코드부분이나 위반사항을 지적해준다.

하지만, PMD에 의해 보고 된 문제는 다소 비효율적인 코드일 수 있고,

잘못된 프로그래밍 습관으로 이후에 프로그램의 성능과 유지 보수성을 저하시킬 수도 있다.

- 동적 프로그램 분석

실제 또는 가상 프로그램을 실행하여 분석

코드 커버리지와 같은 소프트웨어 테스트 기법활용예) APM, 단위 테스트, 통합 테스트, 인수 테스트, Post Man

- 코드 리뷰 기법

1. Code Inspection

* 가장 정형화된 패턴의 기법
* 코드리뷰팀이 시스템이 어느정도 구현된 단계에서 일정한 패턴을 가지고 코드를 분석

1. Team Review

* 코드 인스펙션보다 좀 덜 정형화 되었지만,그래도 일정한 계획과 프로세스를 따름
* 발표자(코드를 만든 사람)가 코드에 대한 설명을 하고, 팀원은 결함이나 개선안을 찾는다.
* Moderator(PM, PL, Senior개발자 등)는 리뷰의 주제를 선정하여 리뷰를 진행하고, 리뷰에서 나온 의견을 정리해서 Action Item으로 기록한다.

1. Walkthrough

* 단체로 하는 코드 리뷰 기법 중에서 가장 비정형적인 방법
* 발표자가 리뷰의 주제와 시간을 정해서 발표를 하고 동료들로부터 의견이나 아이디어를 들음
* 주로 사례에 대한 정보 공유나, 아이디어 수집을 위해서 사용될 수 있다.
* 개발을 위한 프로세스에서 보다는 “Bug 사례에 대한 회의”와 같은 정보 공유성격에 유리

1. Peer Review (2~3명, 주로 2명 리뷰)

* 코드의 작성자가 모니터를 보면서 코드를 설명하고 다른 한 사람이 설명을 들으면서 아이디어를 제안하거나 Defect를 발견하는 방법
* 사전 준비등이 거의 필요 없고, 필요할 때 마다 자주 사용가능
* 주로 Senior 개발자(사수)가 Junior 개발자를 멘토링할 때 사용
* Senior 개발자의 리뷰 역량에 따라서 결과물의 품질이 달라질 수 있음을 유의
* Senior 개발자의 시간 투여량이 많은 만큼, Senior의 참여도가 떨어질 수 있음. (또는 형식적)

1. Passaround (돌려보기)

* 작성자가 리뷰를 할 부분을 메일이나 시스템을 통해서 등록하면, 참석자들이 메일을 통해서 각자의 의견을 개진하는 방식
* 시간과 장소에 구애를 받지 않음. 리모트 작업환경인 경우 유리
* Ownership이 애매함에 유의

=======================================================

**< 3 교시 > 코드 리팩토링 기법**

- 리팩토링(Refactoring)이란?

* 기존 소스 코드의 의미나 행동에 영향을 주지 않고 가독성, 재사용성 또는 구조를 개선하기 위해 기존 소스 코드를 다시 작성한다.
* 코드는 작동하지만 생산 품질이 되기 전에 리팩터링해야한다.

- 리팩토링 명사적 의미

소프트웨어를 보다 쉽게 이해할 수 있고, 적은 비용으로 수정할 수 있도록, 겉으로 보이는 동작의 변화 없이 내부 구조를 변경하는 것.

- 리팩토링 동사적 의미

일련의 리팩토링을 적용하여 겉으로 보이는 동작의 변화 없이 소프트웨어의 구조를 바꾸다.

- 리팩토링이 필요할 때

* 프로그램의 코드가 새로운 기능을 추가하기 쉽도록 구조화되어 있지 않은 경우, 먼저 리팩토링을 해서 프로그램에 기능을 추가하기 쉽게 하고, 그 다음에 기능을 추가한다.

- 리팩토링의 준비

1. 리팩토링을 시작하기 전에 견고한 테스트 세트를 가지고 있는지 확인하라.

수정이 진행되는 과정에서도 계속해서 테스트를 하라.

2. 리팩토링이 끝날 때까지 테스트에 의지하여 수행한다.

- 리팩토링의 기본

1. 리팩토링은 작은 단계로 나누어 프로그램을 변경한다.

2. 실수를 하게 되더라도 쉽게 버그를 찾을 수 있다.

- 리팩토링의 효과

1. 소프트웨어의 디자인을 개선시킨다.

2. 소프트웨어를 더 이해하기 쉽게 만든다.

3. 버그를 찾도록 도와준다.

4. 프로그램을 빨리 작성하도록 도와준다.

- 리팩토링의 실행시기

1. 기능을 추가할 떄

* 새로운 기능을 추가하고 싶을 때, 기존 코드 리팩토링을 먼저 수행한다.

2. 버그를 수정해야 할 때

3. 코드 검토 (Code Review)를 할 때

4. 인디렉션 할 때

* 인디렉션(Indirection)
  + 큰 객체를 여러 개의 작은 객체로 쪼개고, 긴 메소드를 여러 개의 작은 메소드로 나누는 과정
  + 인디렉션으로 관리포인트가 늘어날 수 있음

=======================================================

**< 4 교시 > 코드 속의 나쁜 냄새**

1. 주석(Comments)

* 좋은 주석을 다는 방법?
* 주석을 써야 할 것 같은 생각이 들면, 먼저 리팩토링 하여 주석이 불필요하도록 한다.
* 주석을 사용하기 좋을 때는, 무엇(What)을 해야 할지를 모를 때
* 주석은 무엇이 진행되고 있는 지를 설명할 뿐이다.
* 어떤 것을 왜(Why) 이렇게 했는지를 표시하는데 주석을 써도 좋다.
* 특히 잊어버리기 쉬운 것을 표시할 때는 좋다.

2. 가독성(Readability)

세로 밀집도

* 생각과 생각 사이에 빈 행을 넣어 분리하라.

수직 거리

* 서로 밀접한 개념은 가까이 있어야 한다.

가로 밀집도

* 밀접한 개념은 붙여서, 느슨한 개념은 띄어쓰기.

변수 선언

* 최대한 가까운 곳에 선언하라.
* 지역 변수는 사용하는 시점에 최대한 가까이 선언하라.
* 인스턴스 변수는 클래스 맨 처음에 선언하고, 중간에 선언하지 마라.

호출되는 순서대로 배치 시켜라.

* 한 함수가 다른 함수를 호출한다면 두 함수는 세로로 가까이 배치한다.
* 가능하다면 호출하는 함수를 호출되는 함수보다 먼저 배치한다.
* 이렇게 배치하면 호출되는 함수를 찾기 쉽고 가독성이 높아진다.

유사성이 높은 코드는 가까이 배치 시켜라.

* 친화도가 높을수록 코드를 가까이 배치한다.
* 즉, 비슷한 동작을 수행하는 함수들은 가까이 배치한다.

3. 예외(Exception)

* 오류코드를 사용하기 보다는 예외를 던져라.
* 오류코드를 사용하면 호출자코드가 복잡 해진다.
* 오류코드가 변경되면 호출자코드도 변경되어야 한다.

4. 널(Null)

* Null로 함부로 초기화 하지 않는다.
* 되도록이면 Null을 반환하거나 전달하지 말아라.
  + Null을 반환하는 습관을 가지지 말아라.
  + Null을 인수로 전달하지 마라.
  + Null을 반환하면 모든 코드가 Null 확인으로 가득 차게 될 것이다.

5. 중복된 코드(Duplicated Code)

Extract Method

* 한 클래스의 서로 다른 두 메소드 안에 같은 코드

Extract Method, Pull Up Method

* 동일한 수퍼클래스를 갖는 두 서브클래스에서 같은 코드

Extract Method, Form Template Method

* 만약 코드가 비슷하기는 하지만 같지는 않다면, 비슷한 부분과 다른 부분을 분리한다.

Substitute Algorithm

* 메소드 들이 같은 작업을 하지만 다른 알고리즘을 사용한다면, 두 알고리즘 중 더 명확한 것을 선택

Extract Class

* 서로 관계없는 두 클래스에서 중복된 코드가 사용

6. 긴 메소드(Long Method)

* 메소드를 분해하는데 훨씬 더 공격적이어야 한다.
* 우리가 따르는 방법은, 어떤 것에 대한 주석을 달아야 할 필요를 느낄 때마다 대신 메소드를 작성한다.

7. 거대한 클래스(Large Class)

* 클래스가 지나치게 많은 인스턴스 변수를 갖는 경우
* 언제 인터페이스 클래스를 사용하는지?

8. 긴 파라미터 리스트(Long Parameter List)

* 대부분의 변경은 객체를 넘김으로써 없어진다.
* 이미 알고 있는 객체에 요청하여 파라미터의 데이터를 얻을 수 있으면 Replace Parameter with Method
* 한 객체로부터 주워 모은 데이터 뭉치를 그 객체 자체로 바꾸기 위해 Preserve Whole Object

9. 확산형 변경(Divergent Change)

* 한 변경으로 인해 클래스가 자주 변경되는 경우
* 어떤 클래스를 보고는 “새로운 데이터베이스를 추가할 때마다 항상 이 세 개의 메소드를 수정해야 하는 군” 또는 “새로운 어음이 있을 때마다 항상 이 네 개의 메소드를 변경해야 하는 군”하고 말한다면, 하나보다는 두개의 객체로 만드는 것이 더 좋은 상황에 있는 것이다.
* 특정 원인에 대해 변해야 하는 것을 모두 찾은 다음 Extract Class

10. 산탄총 수술(Shotgun Surgery)

* 한 변경으로, 많은 클래스를 조금씩 수정해야 하는 경우
* 이런 경우 Move Method와 Move Field를 사용하여 변경해야 할 부분을 모두 하나의 클래스로 몰아 넣는다.
* 만약 기존의 클래스 중에서 메소드나 필드가 옮겨갈 적절한 후보가 없다면 새로 만든다.
* 종종 Inline Class를 사용하여 모든 동작을 하나로 모을 수도 있다.

11. 기능에 대한 욕심(Feature Envy)

* 메소드가 자신이 속한 클래스보다 다른 클래스에 관심을 가지고 있는 경우
* 어떤 값을 계산하기위해 다른 객체에 있는 여러 개의 get메소드를 호출하는 경우 등 Move Method
* 메소드의 특정 부분만 이런 욕심으로 고통 받는데 이럴 때는 욕심이 많은 부분에 대해서 Extract Method를 사용한 다음 적절한 위치로 옮겨 주기 위해 Move Method를 사용한다.
* 종종 여러 개의 클래스의 기능을 사용하는 경우 데이터를 가장 많이 사용하는 클래스로 옮겨준다.
* 가장 기본적인 규칙은 같이 변하는 것을 모으는 것이다. 데이터와 그 데이터를 이용하는 동작은 보통 같이 변하지만, 예외도 있다. 이런 예외의 경우는 동작을 옮겨서 한 곳에서만 변하도록 한다.

12. 데이터 덩어리(Data Clump)

* 두 세 개에 중복되는 필드, 여러 메소드 시그니처에 있는 파라미터 등
* 함께 몰려다니는 데이터의 무리는 그들 자신의 객체로 만들어져야 한다.
* 첫 번째 필드로 나타나는 덩어리들이 있는 곳을 찾는 것이다.
* 이 필드들에 대해 Extract Class를 사용한다. 다음 관심을 메소드 시그니처로 돌려 Introduce Parameter Object, Preserve Whole Object를 사용하여 파라미터 리스트를 단순하게 한다.

13. 기본타입에 대한 강박관념(Primitive Obsession)

14. Switch문(Switch Statements)

* 코드 추가 시, 중복된 모든 switch 문을 찾아 바꿔 주어야 한다. 객체지향 개념 중 다형성이 이런 문제를 다루는 훌륭한 방법이다.

15. 평행 상속 구조(Parallel Inheritance Hierarchies)

* 중복을 제거하는 일반적인 전력은 한쪽 상속 구조의 인스턴스가 다른 쪽 구조의 인스턴스를 참조하도록 만드는 것이다.

16. 게으른 클래스(Lazy Class)

17. 추측성 일반화(Speculative Generality)

그 외에는 BwG\_Univ\_소프트웨어 품질 개론\_v1.8\_배포를 참고하도록 하자..