공학 트렌드

코드 품질 관리 Part 2
: 결함을 인정할수록 깨끗해지는 코드 품질의 비결

2013. 1. 28. (제41호)

목 차

소프트웨어공학센터
경영지원TF팀

Ⅲ. 탠덤의 역설

코드의 품질을 개선하기 위해서는 내가 만드는 코드에 문제가 없다는 것을 전제로 노력하기 보다는 오히려 내가 만든 코드에는 문제가 있다는 것을 인정하고 이를 어떻게 신속하게 처리할 것인가를 고민할 때 좀 더 나은 결과를 만들 수 있다. 문제가 생기면 이를 어떻게 빨리 해결할 것인가에 초점을 두는 것이 결함 수를 줄이려고 하는데 효과적이라는 것이다.

무정지 시스템의 대명사로 불리며 전 세계 최고의 품질을 자랑했던 탠덤(Tandem) 서버는 탠덤의 역설을 통하여 완성될 수 있었다. 탠덤은 99.999%의 시스템 가용성을 보장하는 무정지 시스템을 구축하였는데 99.999%의 시스템 가용성이란 1년 동안 겨우 5분 내외의 다운타임만을 허용하는 수준이다. 시스템이 일단 꺼지면 PC조차도 5분은 걸려야 재부팅 된다는 것을 감안한다면, 탠덤의 무정지 시스템은 절대 꺼지지 않는 시스템이라는 것을 의미한다. 이것이 어떻게 이루어냈을까? 탠덤의 무정지 시스템의 탄생에는 창립자의 독특한 사고방식의 전환이 있어 가능하였다. 탠덤의 창립자는 이렇게 생각하였다.

“컴퓨터 안의 모든 부품은 고장 난다, 반드시“

“오늘 안 고장 나면 내일, 내일이 아니면 1년 내에,

그것도 아니면 수 년 내에 반드시! 고장 난다“

탠덤은 메인보드 위의 IC까지 만드는 회사가 아님에도 불구하고 컴퓨터 상의 모든 부품의 고장까지를 커버하면서 99.999%이라는 가용성을 달성하기 위하여 모든 것을 이중화하였다. 프로세스(CPU)도 2개, 컨트롤러도 2개, 보드도 2개, 하드도 2개, ... 전부 다 두 개로 시스템을 구성하였다. 그럼 시스템이 고장 나면 어떻게 할까? 직관적으로 해결을 한다. 우선, 자가진단 도구와 모뎀을 통해서 실시간으로 지구상 세 곳의 탠덤 고객센터 중 가장 가까운 곳으로 자동 보고가 이루어진다. 첫 번째 것이 고장 나면 두 번째 것이 고장 나기 전에 A/S 요원을 보낸다. A/S를 하기 위해 필요한 절차는 매우 간단하다. 입사 후 30분 교육을 받은 정도의 신입 사원도 가방에 부품 한 개 가지고 가서 고객을 방문하여 탠덤 기계의 앞문 열고 고장 난

것을 뽑고(물론 계속 시스템 서비스 중) 가져온 부품을 꽂고 문 닫는 것으로 완료된다. 아무리 심각한 A/S라도 30분이 걸리지 않도록 처리하였다.

반면 다른 시스템 벤더들은 자기 것이 제대로 동작하려면 HDD가 에러가 없어야 하고 랜카드도 문제가 없어야 통신이 될 수 있어야만 했다. 어찌 보면 지극히 당연한 말 이지만 탠덤은 거꾸로 생각했다. HDD 에러 나도! 랜카드 에러 나도! 보드 에러 나도! CPU 불량 나도! 절대 안 서도록 하겠다. 자신들의 시스템은 고장 안 나고 품질 좋다고 주장하는 제품들이 장애 발생 시에 긴 다운타임과 어려움을 겪는 반면, 탠덤은 "우리 기계는 고장 납니다"라고 인정한 결과 역설적으로 서비스 장애를 일으키지 않았다. 이것을 바로 "탠덤의 역설"이다.

IV. 클린 코드(Clean Code)

리팩토링(Refactoring)의 저자인 마틴 파울러(Martin Fowler)는 "컴퓨터가 이해하는 코드는 어느 바보나 다 짤 수 있다. 훌륭한 프로그래머는 사람이 이해할 수 있는 코드를 짠다"라고 하였다. 사람이 이해할 수 있는 코드는 어떤 코드를 의미할까? 사람마다 정의가 다를 수 있겠지만 다음과 같은 특징을 지닌다고 할 수 있다.

- 가독성이 뛰어나다.
- 간단하고 작다.
- 의존성을 최대한 줄였다.
- 의도와 목적이 명확하다.
- 타인에 의해 변경이 용이하다.
- 중복된 코드가 없다.
- 클래스나 메소드가 한 가지 작업만 수행하도록 설계되어 있다.

기존 코드에서 나쁜 냄새가 나는 코드를 찾아 개선하는 리팩토링이 클린 코드를 만드는 좋은 방법이다. 혼란스러운 프로그램 내부의 구조를 개선하다보면 숨어있는 버그를 찾아내는 일도 쉬워지고 기능추가도 수월해진다. 나쁜 냄새가 나는 코드는

어떤 코드를 말하는 것일까? 일반적으로 개발자가 이해하고 수정 및 확장하기 어려운 경우에 해당한다. 마틴 파울러는 저서인 "리팩토링"에서 코드의 악취(Bad Smelles in Code) 22개를 다음과 같이 소개하였다. 이를 참조하여 리팩토링할 코드를 찾아볼 수 있다. 또한 각 리팩토링의 목적과 순서를 카탈로그화 하여 코드 예제를 웹사이트 (http://www.refactoring.com/)에서 살펴볼 수 있다.

<표 1> 코드의 악취(Bad Smelles in Code)

{Duplicated Code}	코드가 중복되거나 겹쳐 있다.
{Long Method}	메소드가 너무 길다.
{Large Class}	클래스의 파일이 크거나 메소드가 너무 많다.
{Long Parameter List}	메소드에 전달하는 인수의 수가 너무 많다.
{Divergent Change}	사양이 변경하면 수정할 곳이 흩어져 있다.
{Shotgun Surgery}	어떤 클래스를 수정하면 다른 클래스도 수정이 필요하다.
{Feature Envy}	클래스간의 속성과 조작의 관계가 부적절하다.
{Data Clump}	같이 다룰 수밖에 없는 여러 개의 데이터가 한 클래스에 정리되어 있지 않다.
{Primitive Obsession}	클래스를 만들지 않고 기본 데이터형만을 사용한다.
{Switch Statements}	switch문이나 if문을 사용해 동작을 분할하고 있다.
{Parallel Inheritance Hierarchies}	서브클래스를 만들면 클래스 계층에 별도로 서브클래스를 만들어야 한다.
{Lazy Class}	만들어 놓은 클래스에서 하는 일은 거의 없다.
{Speculative Generality}	추측하여 지나치게 일반화해 놓는다.
{Temporary Field}	임시로 사용하는 필드가 있다.
{Message Chains}	메소드가 호출하는 연쇄(chain)가 많다.
{Middle Man}	권리만 위임하고 하는 일은 없는 클래스가 있다.
{Inappropriate Intimacy}	불필요하게 쌍방향 링크가 걸려 있거나 상속 관계를 사용한다.
{Alternative Classes with Different Interface}	API가 부적절하다.
{Incomplete Library Class}	기존의 클래스라이브러리가 사용하기 힘들다.
{Data Class}	필드와 getter와 setter 메소드만 가지고 있는 클래스가 있다.
{Refused Bequest}	상속하고 있는 메소드를 호출하면 문제가 발생한다.
{Comments}	코드 자체 설명력이 떨어져 상세한 코멘트를 단다.

개발자가 시스템을 개발하는 경우 처음부터 모든 것을 다 만드는 것이 아니라 기존 프레임워크를 활용하거나 기존 코드에 새로운 기능을 추가하는 경우가 대부분이다. 시스템이 수정되고 기능이 추가되는 과정에서 기능 개발에만 초점을 두기 때문에 코드 품질의 개선은 점점 미비해 버리기 쉽다. 이 과정에서 개선되지 못한 기존 코드들이 점점 늙게 되고 시스템은 점점 스파게티처럼 꼬여 언제 터질지 모르는 폭탄이 되어간다. 코드 리뷰는 기계가 아니라 개발자가 코드를 읽고 검토를 하기 때문에 사람이 이해하는 코드를 만들어주는 좋은 방법이다. 그래야 코드가 건강하고 오래갈 수 있으며 생산성을 향상시킬 수 있다.

코드 리뷰는 개발 초기부터 시작하는 것이 필요하다. 시간이 지날수록 모듈간의 관계가 늘어가 복잡성이 증가하고 고참 개발자는 추가 기능을 개발하고 새로 들어온 개발자가 디버깅을 하다보면 비슷한 결함이 여기저기로 펼쳐지는 최악의 상황이 벌어질 수도 있다.

개발자가 결함을 발견하고 문제점을 5분 이내에 해결하지 못하면 이것 역시 결함으로 보고 해결해본다면 어떨까? 개발자가 만든 코드에는 반드시 결함이 있게 마련이다. 따라서 결함이 존재한다는 명제 하에 결함을 얼마만큼 빨리 처리할 것인가를 목표에 두고 코드 품질을 개선하는 것이 오히려 코드 품질을 높일 수 있는 비결이다. 탠덤의 역설을 통해 장애가 생긴다는 것을 인정하고 이를 얼마나 줄일 수 있는가를 고민하여 무정지 시스템을 완성했듯이 말이다. 지금 자신의 코드를 동료 개발자에게 자신 있게 보여줄 수 있는지 묻고 싶다. 종종 자신이 만든 코드는 자신만 알아보게 작성하고 이를 개선하기 보다는 다시 만드는 것을 선호하는 개발자들을 보게 된다. 아쉽게도 그런 개발자일수록 개발자로써의 성장은 더딘 경우가 많다. 자신이 개발한 코드를 다시 리팩토링 하며 현재 자신이 갖고 있는 지식과 통찰력을 반영해보며 반성하고 개선하는 과정이 부족하기 때문이다.

알고 있는 것과 행하는 것은 별개의 문제이다. 소프트웨어를 잘 하는 조직은 뛰어난 실력자로만 구성된 조직이 아니라, 자신의 환경에 맞는 방법과 훈련 체계를 갖추고 학습하는 조직에서 나오게 된다. 다 알고 있는 얘기라고, 혹은 현실은 힘들다는 핑계를 찾기보다 신뢰하는 동료와 함께 코드 품질을 개선시킬 수 있는 방법을 찾아서 시도해보면 어떨까?

<참고 자료>

1. Martin Fowler, Kent Beck, Refactoting: Improving the Design of Existing Code, Addsion-Wesley, 1999
2. Refactoting Home Page, http://www.refactoring.com/
3. 사람을 위한 자동화, 이클립스 플러그인으로 코드 품질 높이기, IBM DeveloperWorks
4. 로버트 마틴(저), 박재호, 이해영 공저, 클린코드(Clean Code) : 애자일 소프트웨어 장인 정신, 케이앤피스북, 2010
5. 소프트웨어 프로세스 이야기, http://swprocess.egloos.com