SATES 를 활용한
소프트웨어 프로젝트 시작
(개발 실무 관점)

김석환 kim@seokhwan.net

차례

그림 차례

표 차례

0. 머리말

0.1. SATES 개발 배경

SATES 는 Spec And Test Engineering System 의 약자이다. 즉, Spec (사양) 과 Test (검증) 을 위한 시스템이다. 그렇다면 왜 사양과 검증을 위한 소프트웨어를 만들었는가? 그 개발 배경의 이야기는 다음과 같다.

나는 로봇을 위한 소프트웨어를 만드는 일을 하고 있다. 참 재미있는 일이다. 그런데, 이 즐거운 일에 안타까운 점 한 가지는 로봇 등 기계류에 들어가는 소프트웨어는 사람을 다치게할 수 있다는 점이다. 나는 그런 일이 일어나지 않았으면 한다. 그것이 SATES 를 개발한 가장 큰 이유이다.

로봇, 자동차 등 기계류에 들어가는 소프트웨어에게 가장 요구되는 한 가지는 바로 "안전" 이다. 로봇이든 자동차이든 기계 장치는 적게는 수십킬로그램 많게는 수천 킬로그램의 무게를 가지며 매우 빠른 속도로 움직이다. 그런 기계는 안타깝지만 사람을 다치게 할 수 있다. 사람을 다치지 않게 하기 위해 소프트웨어는 안전해야 한다.

"소프트웨어가 안전하다." 이것을 우리는 어떻게 증명할 수 있는가? 수학처럼 아주 멋진 방법으로 증명할 수 있다면 좋을텐데... 안타깝지만 그런 방법은 현실에서 극히 제한적이다. 소프트웨어의 안전은 요구사항을 기본으로하여 우리가 상상 가능한 모든 예외사항을 도출하고, 그 예외사항에서 소프트웨어가 의도한대로 동작하는 것을 테스트하고 그 테스트 결과와 예외사항에 대한 문서를 제 3 의 기관이 검토하여 증명한다. 사실 이것은 증명한다고 하기보다는 안전에 대한 최대한의 공학적 노력을 한다 라고 말할 수 있다.

SATES 시스템은 이러한 활동을 지원하기 위해 작성되었다. 위에 말하였듯이 요구사항 및 예외사항등을 문서화 하고, 이에대한 검증방법 (테스트케이스) 을 문서화 하며, 테스트 결과를 또한 문서화 한다. 사실 상 소프트웨어에 관한 모든 활동 (즉, 계획, 실행, 검증) 에 대한 문서화를 진행해야 하니, 문서화의 범위가 굉장히 넓다. 하지만, 그렇다고 이것이 불가능한 일은 아니다. 아니, 가능하다. 실제 많은 Safety Critical System (안전이 중요한 시스템 : 기차, 항공 SW 등) 들이 이러한 과정을 거쳐 시장에 출시된다.

문제는 툴이다. 물론 시장에는 이미 좋은 툴들이 많이 있다. 하지만, 이러한 툴들은 툴 자체가 하나의 복잡한 시스템으로 사용법을 익히고 제대로 사용하기 까지에는 많은 시간과 노력이

필요하다. 그리고, 일단 대부분 고가이다. 따라서 적은 규모의 프로젝트에서 그러한 툴을 사용하기는 어렵다.

내가 생각하는 툴이 갖춰야할 최소한의 기능은 아래와 같다.

- 요구사항 문서화
- 테스트 케이스 문서화
- 테스트 코드 실행
- 테스트 케이스 문서에 테스트 코드 실행 결과 반영
- 소스코드 문서화
- 그리고, 이들의 mapping 생성

위와 같은 기능을 갖추었으면서 무료인 툴을 찾아보았으나, 찾지를 못했다. 그래서 SATES 를 만들었다.

0.2. 글의 구성

이 글은 SATES 를 설명하기 위해 작성된 글이다. 최초 이 글은 본래 SATES 자체의 문서화를 염두에 두고 쓰여졌다. 문서화를 강조한 툴을 만들며, 그 툴에 대한 문서화를 제대로 하는 것은 너무나 당연한 일이지 않은가?

그런데, SATES 를 설명하려고 하니 SATES 자체에 대한 이야기도 많지만, 요구사항은 무엇인지, 테스트 케이스는 무엇인지 그리고 항상 염두에 두고 있는 V-Model 은 무엇인지 등에 대한 설명이 필요했다. 그러한 설명을 하려고 보니, 점점 SATES 보다는 그러한 일반적 사항에 대한 설명이 길어지게 되었다.

그래서 이 글의 구성은 전체적으로 다음의 두 항목을 중심으로 설명한다.

- V-Model, 요구사항, 테스트 케이스 등 일반적 SW 프로젝트 산출물 이야기
- SATES 를 활용한 V-Model, 요구사항, 테스트 케이스 등 작성의 지원 이야기

1. V-Model

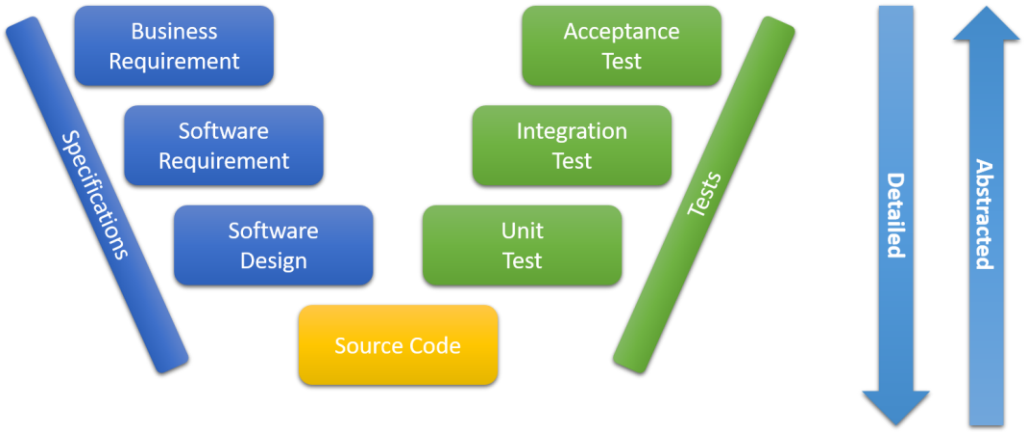

그림 1. V-Model 개념도

그림 1. V-Model 개념도는 유명한 V-Model 을 보여준다. 이 글의 제목은 "성공적인 소프트웨어 프로젝트 시작" 이다. 그리고, "성공적인 소프트웨어 프로젝트" 란 위의 V-Model 의 달성을 의미한다.

왜 갑자기 V-Model 을 말하는지 의아한 독자가 있을 것이다. V-Model 은 안전을 강조하는 소프트웨어 시스템에서 대부분 강조하는 방법론이다. 인터넷에서 Safety critical system, V-Model, V-Process, ISO 2626 등의 키워드를 검색하여 보면 수 많은 정보를 얻을 수 있다.

V-Model 의 왼쪽에는 Specifications (사양) 이 존재한다. 각각, 비즈니스 요구사항, 소프트웨어 요구사항, 소프트웨어 디자인이 위치한다. 이러한 Specification 은 결국 어떻게 소스코드를 작성해야 하는지를 정의한다. 그래서 사양이다.

소프트웨어 디자인은 소스코드들이 어떠한 구조로 구성되는지를 정의한다. 소프트웨어 요구사항은 소프트웨어가 충족시켜야할 기능적, 비기능적 요구사항을 정의한다. 그리고, 비즈니스 요구사항은 궁극적으로 소프트웨어 프로젝트가 달성해야할 비즈니스적 가치를 정의한다.

V-Model 의 오른쪽은 각각의 Specification 에 대응하는 테스트를 정의한다. 단위테스트 (Unit Test) 는 소스코드를 가장 근접거리에서 Software Design 대로 동작하는지 테스트한다.

통합테스트 (Integration Test) 는 각각의 소스코드를 조합하여 요구사항이 제시하는 시나리오별 테스트를 실행한다. 인수테스트 (Acceptance Test) 는 실제 프로젝트의 결과물을 납품하기 전에 실제 환경과 가장 비슷한 환경에서 (혹은 실제 동작과 동일한 환경) 테스트를 하여 최종 프로젝트의 결과물이 납품가능한지를 검증한다.

오른쪽 끝의 화살표가 의미하듯이 왼쪽의 Specification 과 오른쪽의 Test 는 위에 위치할 수록 추상적 (Abstracted) 이며, 아래에 위치할수록 구체적 (Detailed) 이다.

다음 장부터 본격적으로 프로젝트 시작을 위해 챙겨야할 것을 챙겨보자.

2. 사양(Specification) 문서화

성공적인 소프트웨어 프로젝트를 위해 가장 중요한 것은 바로 고객이 무엇을 원하는지를 명확히 알고, 또 그것을 내가 어떻게 만들지에 대해서 명확히 이해하는 것이다. 이것은 매우 당연한 말이다. 우리는 어떤 것을 만족할만한 수준으로 달성했을 때 "성공" 이라는 단어를 사용한다. 따라서, "성공"을 판별하기 위해서는 "우리는 어떤 것을 만드는지?", "만족할만한 수준은 무엇인지?" 에 대한 명확한 정의가 필요하다.

사실, 명확히 정의하지 않아도 사람은 직관적으로 "성공" 이라는 것을 느낄 수 있다. 가령 "오늘 하루 정말 내가 원하는 일들이 모두 잘되고, 우연히 저녁식사를 위해 찾아간 식당은 가격도 저렴한데 맛이 무척좋았다." 라면, 우리는 "오늘 참 성공적인 하루였다." 라고 말하곤 한다.

하지만, 그런 날이 반복되면 우리는 어느 순간 더이상 "성공" 이란 단어를 사용하지 않는다. 이상하게 매일같이 모든일이 잘되고, 매일 같이 가는 곳마다 저렴하고 맛있는 식당이라면, 그것은 더이상 "성공" 이 아닌 "일상" 이 되어버린다. 즉, 알게모르게 "성공" 의 판단 기준이 바뀌어 버린 것이다.

이렇게 본인 자신의 기준도 시간이 흘러감에따라 변하는데,수십, 수백명의 사람이 수개월에서 수년간 함께 하는 소프트웨어 프로젝트의 성공의 기준이 일관되기란 무척 어렵다. 따라서, 우리는 이를 명확히 정의해야 한다. 계속 바뀌는 혼돈스러운 현실속에서 길을 잃지 않을 단 한가지 방법은 바로 명확하게 정의하는 것이다.

그리고, 소프트웨어 프로젝트에서 우리는 이것을 조금 더 고급스러운 표현으로 "사양의 명확화" 라고 말하고, 그것을 문서에 기록하여 남겨두는 것을 "사양 문서화" 라고 말한다.

모든 프로젝트는 고객의 만족을 위해서 시작한다. 고객이 만족하면 프로젝트는 성공한 것이고, 고객이 만족하지 못하면 프로젝트는 실패한 것이다. 우리는 바로 이 고객이 원하는 것이 무엇인지 명확히 해야하고, 또 고객의 만족이 무엇인지 명확히 정의해야한다. 그리고, 그 만족을 최대화 시키는 결과물을 산출하는 것이 바로 우리의 소프트웨어 프로젝트이다.

따라서, 소프트웨어 프로젝트를 시작할 때, 소프트웨어 프로젝트의 사양 (즉, 성공의 기준) 을 명확히하는데 더이상 이의를 재기할수는 없다.

V-Model 의 오른쪽에 위치한 비즈니스 요구사항, 소프트웨어 요구사항, 소프트웨어 디자인은 바로 이러한 프로젝트의 사양 (성공의 기준) 을 정의한다. 바로 비즈니스 요구사항부터 살펴보자.

2.1. Business 요구사항

윗장부터 누누이 강조하고 있지만, 소프트웨어 프로젝트의 목적은 "고객의 만족" 이다. 그리고, 대부분 고객이 만족한다는 것은 고객의 비즈니스적 요구사항의 충족을 의미한다.

그렇다면, 비즈니스 요구사항이란 과연 무엇일까? 사실상, 대부분의 사장님께서 원하시는 비즈니스 요구사항은 한 단어로 말할 수 있다. 바로 "이익의 극대화" 이다. 공익사업을 제외한 모든 사업은 이윤추구를 목적으로 한다. 따라서 우리의 소프트웨어 프로젝트는 고객사가 조금 더 많은 돈을 버는데 일조하는 것이 목표이다.

하지만, 이 "이익의 극대화" 는 너무 추상적이다. "이익의 극대화"를 위한 소스코드를 작성하기는 참 어렵다. (내가 지금 쓰는 이 코드 1 줄이 어떻게 이익을 창출할지 어떻게 내가 알 수 있는가?)

항상 모든 것이 너무 추상적인 경우에는, 이를 구체화하는 것을 먼저 실행한다. 비즈니스 요구사항은 일반적으로 추상적인 "이익의 극대화" 를 달성하기 위해 소프트웨어 프로젝트가 달성해야 하는 목표를 제시한다.

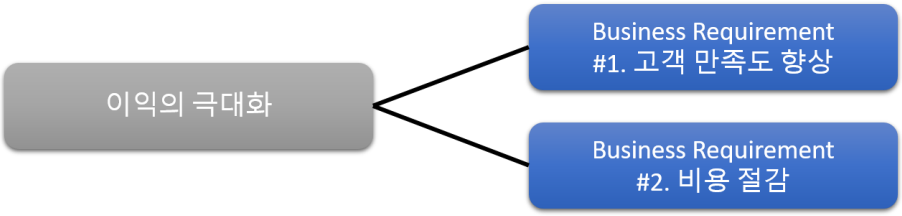

그림 2. 이익의 극대화를 위한 비즈니스 요구사항

그림 2. 이익의 극대화를 위한 비즈니스 요구사항은 바로 이러한 예를 보여준다. 고객의 만족이 향상되면 이익이 높아질 확률이 높아진다. 또한 비용이 절감될수록 이익이 높아질 확률 역시 높아진다. 우리가 수행하는 일에 절대적으로 성공이 보장되는 활동은 없다. 다만, 그 확률을 높이는 노력을 할 뿐이다.

자, 이제 조금 더 구체화 되었다. 그렇다 우리가 실행할 소프트웨어 프로젝트의 목표는 바로 "고객만족도 향상" 과 "비용절감" 이다. 그리고, 이 둘 중에는 "고객 만족도 향상" 더 높은 우선순위를 가진다고 가정하자.

2.1.1. 비즈니스 요구사항과 소스코드

언뜻 보기에 "고객만족도 향상", "비용절감" 과 같은 비즈니스 요구사항은 소스코드와 관계가 없어 보인다. 하지만, 사실은 그렇지 않다. 사실 생각보다 꽤 많은 순간 개발자는 비즈니스 요구사항을 참조한다. (그리고, 참조해야 한다.) 이를 아래의 예를 들어 설명하겠다.

	라이브러리 A	라이브러리 B
메모리 사용량	많음	적음
실행 속도	빠름	느림
런타임 라이센스 비용	5000 만원 / 년	1000 만원 / 년

표 1. 라이브러리 선택 예제

표 1. 라이브러리 선택 예제를 보면 라이브러리 A 와 B 는 그 특징이 명확하다. A 는 메모리는 많이 사용하지만, 속도는 빠르고 가격은 비싸다. 반면에 B 는 메모리는 적게 사용하지만, 느리고 가격은 저렴하다.

우리는 이러한 경우에 어떤 라이브러리를 선택해야 하는가? 이럴 때, 비즈니스 요구사항이 유용한 길잡이 역할을 한다. 위에서 정의한 비즈니스 요구사항에서 우리의 첫 번째 요구사항은 바로 "고객 만족도 향상" 이다. 실행 속도의 빠름은 고객만족도 향상에 도움이 될 것이 너무나 명확하다. 그렇다면, 우리의 선택은 바로 라이브러리 A 가 된다.

반대로, "비용절감" 이 첫번째 요구사항이었다면 우리는 라이브러리 B 를 선택해야했을 것이다.

이는, 비단 라이브러리 사용뿐만 아니라, 소스코드 작성에도 그대로 적용이 된다. 예를 들어, "고객만족도 향상" 이라는 가치를 위해서 우리는 똑같은 알고리즘이라고 해도 성능에 조금 더 초점을 맞춘 알고리즘을 구현하는 소스코드를 작성해야 한다.

사실, 이러한 비즈니스 요구사항은 소프트웨어 요구사항에 적용이 된다. 예를 들어, "고객만족도 향상" 이라는 가치를 위해, 소프트웨어 품질항목 중 ISO 25010 에 명시된 사용성 (Usability) 의 향상을 위한 요구사항이 포함될 수 있다. 하지만, 이러한 요구사항이 어떤 라이브러리를 선택해야한다. 또는 어떤 알고리즘을 선택해야 한다. 와 같은 아주 상세한 내용까지 정의할 수는 없다. 이런 경우에 우리는 비즈니스 요구사항을 참조하여, 판단하는 것이 가장 올바른 방법이다.

2.2. 소프트웨어 요구사항

비즈니스 요구사항이 사업적 성공을 위한 가치를 추상적으로 제시한다면, 소프트웨어 요구사항은 이를 달성하기 위해 구체적으로 소프트웨어가 어떤 기능을 제공해야하는지를 명확히 제시해야 한다.

소프트웨어 요구사항 문서가 비즈니스 요구사항에 비해 다른점은 바로 달성여부가 판단기준을 명확히 제시한다는 점이다. 즉 문서의 내용은 검증가능한 (testable) 내용이라는 것이다.

예를 들어, "고객만족도 향상" 이라는 것에 대해서 우리는 "작년대비 고객만족점수 10% 향상" 이라는 구체적인 목표를 제시할 수 있다. 그리고, 이 목표가 달성되었다고 가정하자. 하지만, 사실상 소프트웨어가 고객만족점수를 정말 10% 향상 시켰는지에 대해서는 검증이 불가능하다.

하지만, 소프트웨어 요구사항은 예를 들어 "고객 만족도 향상을 위해 시스템은 사용자 입력 후 0.1 초내에 결과를 화면에 보여준다." 와 같은 내용으로 작성된다. 이것은 스톱워치 혹은 소프트웨어 타이머등을 통해서 완벽히 검증이 가능하다.

소프트웨어 요구사항 문서는 그렇게 작성되어야 한다.

실무적 관점에서 소프트웨어 요구사항 문서가 갖추어야 하는 가장 중요한 것은 바로 두 가지이다.

1) 검증방법
2) 우선순위

2.2.1. 검증방법

검증방법이란 소프트웨어 작성 후, 요구사항을 만족시키는지 불만족시키는지를 명확히 기술한 것을 의미한다.

가령, "0.1 초 내에 화면의 결과를 보여준다" 라는 요구사항이 있다면, 이를 위한 검증 방법으로

- Windows OS 의 표준 타이머 API 를 활용하여 시간을 측정한다.
- 시간 측정의 오차를 감안하여 10%의 오차를 허용한다. (즉, 0.11 초까지 합격으로 간주한다.)
- 99.99% (즉, 1 만회 중 1 회) 달성을 목표로한다.

요구사항에 위와 같은 내용이 기술된다면, 테스트 프로그램은 Windows 의 표준 API 를 활용한 타이머를 활용하고, 0.11 초를 초과하는지 여부를 검사하며, 1 만회를 검사하여 1 회를 초과하는

경우가 있는지 여부를 검사한다. 그리고, 이를 만족하면 우리는 그 요구사항을 만족했다고 판단할 수 있다.

2.2.2. 우선순위

중요한일을 먼저 하는 것이 중요하다는 것에 대해서 이의를 재기할 사람은 없을 것이다. 가령, 오븐토스터기에 탑재되는 소프트웨어를 만들 때, 온도제어 기능과, 타이머 기능이 있다면 온도제어기능이 우선순위가 더 높을 것이다.

이 경우, 일단 온도제어 기능이 먼저 완성되었다면 중간에 프로젝트가 중단이 되더라도, 그 오븐토스터기는 사용 자체는 가능할 것이다. 타이머가 없는 것은 무척이나 불편하겠지만 스톱워치를 활용하여 대체가 가능하다. 반면에 타이머는 되지만, 온도제어가 되지 않는다면, 빵을 구울 수 없기 때문에 사용 자체가 불가능하다.

위에 기술한 검증방법과 우선순위는 어떤 요구사항 문서에서든지 필수적으로 포함되어야 하는 내용이다. 거기에 추가로 가능하다면, 아래의 사항을 추가하도록 한다.

3) 요구공수

2.2.3. 요구공수

요구공수라는 말은 소프트웨어 개발자에게는 익숙하겠지만, 일반인에게는 익숙하지 않은 개념일 것이다. 쉬운 말로, 해당 기능을 구현하기 위해 얼만큼의 노력 (몇명이 몇시간을 투입) 이 필요한지를 의미한다.

사실 어떤 기능을 개발할 때, 그 기능이 얼만큼의 노력을 요구하는지를 알기는 쉽지 않다. (사실상 불가능하다고 생각한다.) 다만, 시간이 흐른 후, 얼만큼의 노력이 투입되었는지 기록은 가능하다.

이 기록은 사실 나중에 요구사항 변경에 대한 비용산정에서 엄청난 위력을 발휘한다. 이는 뒷장에서 기술하도록 하겠다.

어찌되었든 명심할 것은 가능하면 요구공수를 꼭 작성하고, 프로젝트를 진행하면서 실제 투입된 공수를 꼭 기록하라는 것이다.

2.2.4. 소프트웨어 요구사항과 소스코드

소프트웨어 요구사항은 소스코드가 산출해야하는 내용을 명확히 기술한다. 따라서, 내가 어떤 소스코드를 작성할 때, 그 소스코드가 요구사항에 포함된 검증방법을 통해 OK 인지 Fail 인지 확인하며 작성하게 된다.

일단 소스코드는 요구사항의 검증방법을 만족시키면 개발실무 차원에서는 문제가 없다고 생각한다. 물론, 프로젝트가 진행됨에 따라 요구사항은 변경이 되고, 또 검증방법에는 오류가 발견된다. 이 이야기는 뒷장에서 더욱 자세히 하도록 하겠다.

2.3. 소프트웨어 설계

감히 말하건데 소프트웨어 설계는 소프트웨어 개발자로서 할 수 있는 일중에 가장 재미있는 일이다. 소프트웨어 요구사항을 만족시키기 위해 어떻게 모듈간의 관계를 맺으면 가장 효율적일지? 어떻게 모듈간의 관계를 끊으면 예상되는 변경에 유연하게 대처할 수 있을지? 에 대한 심도 있는 고민을 아주 즐겁게 하는 과정이다. 내가 생각하는 소프트웨어 설계는 그런 일이다.

소프트웨어 설계에 대해서는 수 많은 서적들이 존재하고 수 많은 방법론을 제시하면서 동시에 모든 서적들은 동시에 소프트웨어 설계에 정답은 없다고 말한다. 무척 동의한다. 다만, 아주 일반적으로 추천되어지는 방법론은 존재한다. 이 장에서는 그런 내용을 담고자 한다.

소프트웨어 설계란 결국은 소프트웨어가 요구사항을 만족시키기 위해 어떻게 동작을 해야하는지, 모듈들이 어떻게 관계를 맺어야 하는지를 소스코드보다 한 단계 상위의 추상적인 관점에서 기술하는 것이다. 이 추상적인 관점은 일반적으로 두 개의 관점으로 나누어진다. 그리고, 그것들은 바로 static view 와 dynamic view 이다.

static view 란, 일반적으로 패키지 다이어그램, 클래스 다이어그램과 같이 소스코드들이 서로 어떤 관계를 맺고 있는지를 표현한다. (어떤 클래스가 어떤 클래스를 상속받고, 어떤 클래스가 어떤 클래스를 멤버로 가지는지는 소스코드에서 표현이된다.)

dynamic view 란 이 소스코드가 실제 CPU 에서 어떻게 동작하는지를 설명한다. 어떤 클래스가 언제 인스턴스로 생성이되고 소멸되며, 그 어떤 클래스는 다른 어떤 클래스를 어떤 시점에 사용하는지를 표현한다.

이 이상의 긴 이야기는 별도의 소프트웨어 아키텍쳐 서적을 통해 공부하기를 바란다. 본격적 이야기를 하기에는 이 글은 소프트웨어 설계책이 아니고, 짧게 이야기하기에는 너무 어렵게 느껴질 것 같다.

3. 소스코드 작성과 테스트 실행

2 사양(Specification) 문서화장에서 우리는 사양을 정리하였다. 즉, 소프트웨어가 수행해야하는 일을 우리는 정의하였다. 개인적으로 프로젝트에서 가장 중요한 일이라고 생각한다. 사양이 명확히 정의되지 않은 프로젝트를 수행하는 것은 목적지를 입력하지 않고 네비게이션을 키는 것과 동일하다. 목적지를 입력하지 않은채로 네비게이션을 키고 운전을 아무리 해도 우리는 목적지에 다다를 수 없다. 목적지를 입력하지 않았는데, 어떻게 목적지에 갈 수 있겠는가?

잠깐 지면을 빌려 말하자면, 참으로 안타까운 일이지만 목적지를 입력하지 않고 운전을 하는 것과 같은 프로젝트가 세상에 너무나 많다. 국내에 많은 대기업들이 목적지 없는 프로젝트를 실행한다. 이러한 프로젝트의 결과물은 사실 상 회사에 어떤 도움도 되지 않는다. 그리고, 그 시간동안 투입한 인력과 비용은 고스란히 회사의 부담으로 남게 된다. 더욱더 안타까운 점은 그 부담은 그 회사가 부담하는 것이아니라 그 회사의 하청업체가 부담하게 되는 경우가 많이 있다. 나도 그렇고, 이 글을 읽는 여러분 그리고 여러분의 친구 가족 우리들은 대기업보다는 중소기업에서 일을 하고 있을 확률이 높다. 그리고, 우리는 알게모르게 그러한 목적지 없는 프로젝트의 실행의 낭비에 대한 비용을 부담해주고 있는 것이다. 너무나 안타까운 현실이다. 넋두리는 그만하자.

사양이 명확히 정의되었다면, 이제 그 다음일은 소스코드를 작성하고, 이를 검증하는 일이다. 이 장의 제목이 "소스코드 작성과 테스트 실행" 인것은 이 둘을 꼭 동시에 하자는 의미이다. 나는 나름 문서화, 소스코드 작성, 테스트 실행을 동시에하는 것을 무척 중요시하는 개발자라고 나름 자부한다. 그런 나 조차도 이들을 동시에 못하는 경우가 많다. 꼭 프로젝트의 바쁜 일정때문이 아니라 인간적 게으름으로 인해서 그렇다. 하지만, 그런 내게 항상 나는 실망한다. 그러지 말자. 꼭 소스코드 작성과 검증코드 작성, 검증의 수행은 동시에 진행하자.

3.1. 코드작성 사전준비

코드를 작성하는데 어떤 사전준비가 필요한가? 간단히 아래의 두 항목이 있다.

- 개발환경 준비

- 빌드/배포환경 준비

3.1.1. 개발환경 준비

개발환경 준비는 소프트웨어 산출물이 어떤 환경에서 동작하고, 어떤 빌드시스템을 통해서 빌드가 되는지를 명확히 하는 것이다.

예를 들어, 어떤 개발자가 "우리는 C++ 를 사용하고, Windows 10 에서 비쥬얼 스튜디오 2017 로 개발하는 것이 표준환경입니다." 라고 말한다면 그 개발자에게는 조금 더 구체적인 정보를 요구해야한다. 그리고, 개발자는

우리는 Windows 10 을 사용합니다만, 정확히 커맨드 프롬프트에 *[Version 10.0.17134.228] 로 명기된 버전을 사용합니다.* *그리고, C++ 11 에 명기된 문법을 사용하고, 그상위 문법은 사용하지 않습니다.* *컴파일러는 Visual Studio 에 내장된 컴파일러인데, 정확한 컴파일러 이름 및 버전은* *Microsoft (R) C/C++ Optimizing Compiler Version 19.15.26726 for x86* *입니다.* *개발자에 따라서 IDE 로 Visual Studio 2017 Professional 버전을 사용하고, 현재 표준 버전은* *15.8.2 입니다.* *아, 그리고 Windows 의 경우 회사의 IT 서비스에서 제공하는 업데이트 사이트를 통해서* *업데이트된 버전을 사용합니다. 기본적으로 Microsoft 의 정기 릴리즈를 그대로 업데이트 하는* *것은 권장하지 않습니다.*

표 2. 개발환경 시나리오 예제

와 같은 대답을 해주어야 한다. 그렇지 않다면 그 개발자와는 일을 지속하지 않을 것을 권장한다.

개발환경이란 이런 것이다.

물론 요즈음과 같이 Microsoft 의 Nuget, Deiban 계열의 APT, Node 의 npm 과 같이 패키지 메니저가 무척 훌륭한 환경에서 위와 같은 것을 강제하는 내게 편집증 환자라고 해도 딱히 할말은 없다. 하지만, 적어도 당신이 개발자라면, 당신이 작성하는 소스코드가 정확히 어떤 OS 의 어떤 버전에서 어떤 컴파일러의 어떤 버전이 빌드를 하고, 여기에 함께 링크되는 라이브러리는 어떤 버전인지 명확히 알고 있어야 하지 않겠는가? 이것을 정확히 모르면 아래와 같은 일이 발생한다.

2018 년 여름 나는 여느때와 같이 코드를 쓰고, 빌드를 하고 이를 배포했다. *그런데, 그 다음 날 아침 고객에게 전화가 왔다. 어제 당신의 PC 에서 잘 동작하던 것을* *함께 확인하였습니다만, 이상하게 우리 PC 에서는 정확히 동작하지 않네요.*

모든 메시지가 정상으로 올라오는데 갑자기 어느 순간 exception 발생 메시지가 뜨면서 동작하지 않습니다.

나는 다시 내 PC 를 키고 어제의 코드를 다시 빌드해서 돌려보았다. Debug 옵션, Optimization 옵션을 모두 변경해 보면서 실행해 보았다. 문제가 없었다.

다시 고객사에게 전화를 걸어 Windows 의 버전을 확인했다. 내 PC 와 동일하다. 아… 도대체 무슨일일까… 그렇게 괴로운 한달을 보내고 확인한 내용은 다음과 같았다.

내가 바이너리를 배포하기 전날, Windows 는 자동업데이트를 실행했고, KB4XXXX 를 업데이트 했다. 이 업데이트에는 내가 배포하는 바이너리가 사용하는 dll 하나가 업데이트 되었다. 그 업데이트는 순간적으로 network 이 다운되는 문제를 해결한 버전이었다.

내 PC 에는 이것이 설치되어 있었고, 고객의 PC 에는 그 업데이트가 설치되지 않아있었다. 휴 …

표 3. 잘못된 개발환경의 폐해 시나리오 예제

위와 같은 시나리오는 조금 극단적이기는 하지만, 약 2-3 년을 주기로 한번씩 겪게 되는 문제이다. 분명히 겪는다. 따라서, 이를 피하기 위해서는 아래의 사항을 명확히 통제해야한다.

1. OS 버전
2. 컴파일러 버전
3. 링크가 되는 모든 라이브러리의 버전

이를 우리는 개발환경이라고 부른다.

3.1.2. 빌드/배포환경 준비

개발환경의 준비를 보고 혹시라도 복잡하다고 생각하였다면, 그러면 안된다는 말씀을 드리고 싶다. 빌드/배포 환경의 준비는 조금 더 복잡하다. 빌드/배포 환경에서 가장 중요한 것은 바로

- Single Script File

이다.

이 Single Script File 은 아래의 일을 실행해야한다.

- Version 시스템으로 부터 Source Code 업데이트
- Source Code 의 빌드
- 테스트 케이스 전수 테스트
- 테스트의 결과가 OK 인 경우, 이의 로그와 함께 배포서버에 commit

조금 논란의 여지가 있을 수 있겠지만, 개인적으로 저기까지가 개발자 혹은 개발품질 부서가 해주어야 하는 일이다. 그리고, 위의 단계를 거친 binary 가 품질부서에 전달이 되고, 품질부서는 다시 이를 검증하여 품질부서가 OK 를 하면, 이게 최종 고객에게 전달이되어야 한다. 사실 이것이 흔히 이야기하는 CI (Continuous Integration) 서버의 역할이다.

생각보다 많은 회사에 (요즈음은 사실 모든 회사에) CI 서버를 운용한다. 그런데, 참으로 이상한 사실은 위의 활동을 모두 수동으로 하고 있다는 사실이다.

개발자는 Source Code 를 본인의 PC 에서 받아서 빌드하고, 이의 단위테스트, 통합테스트 정도를 본인이 직접 PC 에서 실행한다. 그리고, 문제가 없으면 이를 회사 시스템에 등록한다. 그리고, 품질부서는 그 파일을 받아 검증을 실행한다.

이 과정의 문제점은 다음과 같다.

1) 개발자가 실수로 (혹은 의도적으로) 버전시스템의 3345 버전이라고 하면서 2345 버전을 빌드한 바이너를 제출한 경우, 이를 확인할 방법이 없다.
2) 위의 (1) 을 확인할 방법이 없기 때문에, 개발자가 테스트를 전혀 진행하지 않고, 테스트 케이스를 모두 통과했다고 하여도, 이를 확인할 방법이 없다.

솔직히 실무에서 위와 같은 실수를 의도적으로 하는 개발자를 본적이 있다. 나는 그 개발자를 싫어한다. 그리고, 개발자가 그런일을 하더라도 이를 걸러낼 수 없는 회사의 시스템은 더욱 증오한다. 개발자도 나쁘지만, 뻔히 그런일을 방지할 수 있는 시스템이 있음에도 이를 실행하지 못하는 회사의 시스템은 더욱 비난 받아 마땅하다.

지면을 빌려 혹시라도 회사에서 SW 의 품질부서에 종사하고 계신다면 한 가지 부탁을 하고자 한다. 다른 것은 몰라도

소스코드 fetch / build 는 품질부서에서 실행해야 합니다.

아니면, 본인의 손에 입수되는 바이너리가 시스템 상에서 소스코드 -> 바이너리로 가는 과정이 완벽히 자동화되어 있고, 그 과정에 개발자가 소스코드 커밋 외에는 다른 어떤 활동도 할 수 없게 통제되어야 합니다.

가끔씩, 개발자들의 장난으로 인해 곤란한 일을 겪은 안좋은 기억을가지고 계신 품질 담당자를 만날때가 있다. 그 개발자도 나쁘지만, 개인적으로 그렇게 안좋은 일을 겪고도 아직도 개발자를 의심의 눈초리로 쳐다보며 개발자가 빌드해준 바이너리를 받아 테스트를 실행하는 그 품질 담당자를 보면 안타깝기 그지 없다. 적어도 빌드 스크립트의 실행법만이라도 배운다면, 그게 안된다면 CI 서버를 제대로 운용하게끔 회사에 요구하고 이를 관철시키더라도 더이상 빌드에 대한 개발자의 장난으로 부터는 자유로울 수 있을텐데...

빌드서버의 존재의 이유는 너무나 당연하다. 표 3. 잘못된 개발환경의 폐해 시나리오 예제 의 시나리오의 방지를 위해서이다. 아무리 개발 PL 이 표 2. 개발환경 시나리오 예제 와 같은 환경을 강제하더라도 어떤 사정에 의해서 모든 개발자들이 이런 표준환경을 가지고 있는것은 불가능하다. 하지만, 빌드서버 1 대에 대한 관리는 가능하다. 즉, 표 2. 개발환경 시나리오 예제 과 같은 표준환경을 구축한 환경이 필요하고, 빌드서버는 이의 실현예인 것이다.